An approximate mirror from source document at
http://www.geocities.com/tablizer/oopbad.htm
made 2002-08-27 14:33:43 CDT (Aug Tue)

OOP Goop Image

Object Oriented Programming Oversold!

OOP criticism and OOP problems
The emperor has no clothes!
Reality Check 101
Snake OOil

Updated: 6/23/2002

OOP Myths Debunked:

- **Myth: OOP is a proven general-purpose technique**
- **Myth: OOP models the real world better**
- **Myth: OOP makes programming more visual**
- **Myth: OOP makes programming easier and faster**
- **Myth: OOP eliminates the "complexity" of "case" or "switch" statements**
- **Myth: OOP reduces the number of places that require changing**
- **Myth: OOP increases reuse (recycling of code)**
- **Myth: Most things fit nicely into hierarchical taxonomies**
- **Myth: Sub-typing is a stable way to model differences**
- **Myth: Self-handling nouns are more useful than self-handling verbs**
- **Myth: OOP does automatic garbage-collection better**

- **Myth: Procedural cannot do components well**
- **Myth: OO databases can better store large, multimedia data**
- **Myth: OODBMS are overall faster than RDBMS**
- **Myth: C and Pascal are the best procedural can get**
- **Myth: OOP would have prevented more Y2K problems**
- **Myth: OOP "does patterns" better**
- **Myth: Only OOP can "protect data"**
- **Myth: Implementation changes significantly more often than interfaces**
- **Myth: Procedural/Relational ties field types and sizes to the code more**
- **Myth: Procedural cannot extend compiled portions very well**
- **Myth: No procedural language can re-compile at the routine level**
- **Myth: Procedural/Relational programs cannot "factor" as well**
- **Myth: OOP models human thought better (Which human?)**
- **Myth: OOP is more "modular"**
- **Myth: OOP divides up work better**
- **Myth: OOP "hides complexity" better**
- **Myth: OOP better models spoken language**
- **Myth: OOP is "better abstraction"**
- **Myth: OOP reduces "coupling"**
- **Myth: Most programmers prefer OOP**

- **Fact: OO fans have repeatedly failed to demonstrate OO superiority. They can talk up a good storm, but cannot walk their talk.**

See The Guide To Myths for specific information on each myth, including a disclaimer.

*** Jump to Summary ***

SCOPE DISCLAIMER: I have been programming small and medium custom business applications for most of my career. Most of my complaints against OO are related to this rather large niche. Perhaps OO is good for other niches; however, I cannot really answer for other niches. (Nevertheless, the critique may apply to some aspects of other niches.) The problem is that OO is permeating

my niche where it probably does not belong. Its few, rare benefits there do not justify the added complexity. Academics tend to ignore business applications, and instead promote paradigms optimized for other niches more favored by them.

The GUI Link

OOP became popular primarily because of GUI interfaces. In fact, many non-programmers think that "Object" in OOP means a screen object such as a button, icon, or listbox. They often talk about drag-and-drop "objects". GUI's sold products. Anything associated with GUI's was sure to get market and sales brochure attention, regardless of whether this association was accurate or not. I have even seen salary surveys from respected survey companies that have a programming classification called "GUI/OOP Programming".

> **Some OO proponents have strongly disagreed with a popularity connection between GUI's and OOP. It is difficult to really know for sure why society moves one way or another on such things without detailed surveys or neuron tracing, but my personal observation is that GUI's did impact the impression of OOP. Bertrand Meyer appears to confirm this in Chapter 1 of OOSC2.**

Screen objects can correspond closely with OOP objects, making them allegedly easier to manipulate in a program. I do not disagree that OOP works fairly well for GUI's, but it is now being sold as the solve-all and be-all of programming in general.

Some argue that OOP is still important even if not dealing directly with GUI's. In my opinion, much of the hype about OOP is faddish. OOP in itself does NOT allow programs to do things that they could not do before. OOP is more of a program organizational philosophy rather than a set of new external solutions or operations.

The OOP organizational philosophy allegedly makes software more change-friendly. It is also often claimed that OOP makes software more reusable, bringing less re-invention of the same software wheels. However reuse claims have lost steam of late, putting

change-friendly at the top claim spot. (Individuals' brag-lists may vary.)

Although GUI's revolutionized the industry, there is more to computers than GUI's. Also, there may be other ways to handle GUI's besides OOP. Nobody ever bothered to do a proper study. There is more proof to the existence of UFO's than to OOP being the best way to do GUI's or any other programming topic. (Many of the bad impressions of procedural GUI's stem from the fact that 1980's languages and API's used for GUI's were often optimized for the hardware, not for the programmer. They also did not have the benefit of prior experience. They were pioneers.)

Beating Up Straw Men

I am not saying OOP is useless, per se; many are just frustrated with the fact that OOP has slowed or even reversed programming progress in other areas. I have debated OO fans that appear ignorant to some nifty techniques available in old-fashioned procedural programming. Often times someone will compare C to C++ and conclude that the differences are paradigm differences.

For example, Sometimes programmers complain that subroutine parameters are too sensitive to positional ordering changes or quantity changes. However, named parameters, dictionary arrays, and/or optional parameters can often alleviate these issues if a language supports them. (Named parameters can optionally make routines very Smalltalk-like. Some people erroneously think that OOP invented named parameters.)

In other words, their bashing of the procedural/relational (p/r) paradigm is really an attack on one specific language, a bad vendor, or their own lack of p/r skills. Even people who have been using p/r for several years often never try certain useful p/r features or techniques before they switch to OOP. There is more to p/r than C and Pascal. The fault is usually the bathwater, not the baby.

My pet paradigm, Table Oriented Programming, is another area that was blunted by OO hype and product penetration. OOP has perhaps contributed some good ideas to programming, but is certainly no holy grail. I am saddened to see non-OO research, training, and tools stop advancing because everybody focuses on OOP instead. OOP's alleged advantages are becoming a self-fulfilling-prophecy because of this. They should prove that OO is better for everything first before pulling the plug on the others.

Object Oriented Technology is a subjective philosophical paradigm only. As we shall see, it is an organizational religion that has not (so far) graduated into a proven general-purpose benefit.

Bait-and-Switch

One of the reasons for the popularity and management acceptance of Object Oriented Programming is clever little examples that demonstrate the alleged power of OOP.

Most experts realize that these examples are not very representative of "good" real world OO programming. The actual implementation often involves fairly complex arrangements that make real OO messy and more confusing than its competitors. OO fans defend the simple ones as "just training examples," but there is rarely a disclaimer of such near the examples.

If you are new to OOP, please don't be fooled by simplistic examples. These bait-and-switch examples often take the form of geometric shapes, animal categories, vehicle taxonomies, vehicle parts, employee types, Y2K dates, stacks, device drivers, clothing, or bank account examples.

These examples often assume the world can usually be divided into clean, never-changing (or hierarchically-changing) categories or "chunks", in which groups of features always stay together or change in a lockstep dance within generally non-divisible chunks.

The truth is messier, and OO is no better optimized to deal with dynamic feature relationships and changes than competitor paradigms, and in many cases seems to be messier in the end.

Also note that in many cases classifications are given by the users via an interface, and not done by programmers via programming code. Accountants build and maintain accounting code categories, for example, and not the programmers. (The internal result is usually database cross-references, and not OOP classes.)

Land of Confusion

OOP technology has generated more confusion than almost any other computer technology. For example, many OOP "experts" claim that most companies are either not using OOP properly, or are not taking advantage of OOP. Experts also say that there is a long learning curve before many people grasp the power of OOP; that its benefits can't really be taught, at least not understood, from a book. It has almost become like Particle Physics, in which only a small elite group appears to understand it properly, and everybody else needs years of meditation and practice. Even OOP's goals are not clear.

Ironically, OOP is sometimes billed as "better fitting the way people think". Years of meditation and study to learn how to "think naturally"? I am thinking of setting up a $60-per-hour consultancy to teach sports fans to drink beer and belch in order to "optimize their recreational satisfaction".

Persistent storage (databases) is especially in turmoil in the OOP world. Some experts say that a company cannot get the benefits of OOP without using Object-Oriented Database systems; others say that companies need to hire "middle-layer" experts to convert conventional databases into virtual OO databases for other programmers. However, there are very few books on database connections to OOP (relative to other OOP titles), and probably

fewer experts. Is it a neglected field? Is a layer expert not really needed? Are current OOP programmers doing it wrong because they have no layer experts around?

Further, little is known or agreed on about whether OO's benefits are mostly limited to large projects, or all project sizes. I get very mixed opinions when I ask OO proponents about project size and benefits. (See also: OOSC2 Critique section on project size.)

There is also a lot of mixed opinions about when, where, why, how, and if OO works better for some domains (industries) over others. One OO proponent will say it works well for games but not embedded systems, and another will say the opposite. Some proponents say it works well for nearly all domains.

Further, there is a lack of consistency in modeling techniques by OOP celebrities. Methodology-of-the-week is commonplace. The lack of consistency makes it tough to make any generalizations about how OOP goes about modeling useful business applications. An OOP consultant may have to be well-versed in dozens of OO methodologies to be able to walk into a shop and perform any useful work any time soon.

Not Table-Friendly

OOP often does not map well to relational databases. Even though OOP languages are common, object-oriented databases (known as "OODMS") are not. OODMS's are in their infancy and are still not selling as well as the "traditional" RDBMS. In business applications, most object instances are actually equivalent to fields and records. A good data dictionary (table-oriented technique) is similar to using OOP constructors, destructors, and attributes associated with fields and tables.

Using relational tables with OOP requires converting (mapping) fields into objects and visa-verse when putting them back into the tables. This is a painstaking process and can waste a lot of

programming time. It reminds one of the luggage and visa inspection process on international plane flights.

Further, the popular OOP languages like Java and C++ require the use of API's or API-like constructs to manipulate databases. API's are fine for occasional or special use, but if 80 percent of your program manipulates tables, API's become a bottleneck.

See Table Oriented Programming for more about OOP, TOP, and tables. (There is a little bit of topic overlap in this document.)

People read this section and keep mentioning "wonderful" products that allow automated mapping of relational tables to OOP objects. However, these appear not to be an improvement over traditional or table-oriented paradigms, but simply a tool to close the gap a bit. Why have an extra layer dedicated to converting one paradigm to another?

There is nothing wrong with layers as long as the purpose is to hide low-level details from the programmer or code maintainer, but these mapping tools are translating from one high-level paradigm (relational) to another high-level paradigm (OOP). Thus, they are not improving the system, only unnecessarily complicating it by spending complexity on translating between different philosophies instead of adding something helpful. (This philosophy collision is sometimes called the Impedance Mismatch. It is a controversial topic among OO proponents.)

There is also talk that OO systems are not very good at ad-hoc queries, which where made popular with SQL in relational systems. Part of the theory behind this claim is that OOP requires most object behavior to be declared in advance. In other words, a predetermined set of methods are associated with a data item (object). This is part of "encapsulation" in OOP lore. One is not supposed to "go outside of" these predetermined methods. Ad-hoc queries often require that unanticipated operations be applied to any and all data items. This appears to violate basic encapsulation principles.

Rather than pick sides on this complicated query issue, just be warned to be careful if ad-hoc queries are an important part of your business. The OOP proponents usually agree that most current OOP systems are not optimized for ad-hoc queries, but claim that it will improve with time.

See Also:
Double Dipping
Data Protection

Melding Can Be Hazardous

Spock mock

OOP pushers like to talk about how great it is to meld the methods (procedures) and the data together. Data in traditional relational form is relatively easy to transfer to different systems as technology and vendors change. Procedures are NOT easy to transfer. If you mix the data in with methods, then you are STUCK with the current OO programming language to interpret your data. You cannot easily read the data except with the OOP language and/or program that generated it.

Even if you stay with one OOP language or protocol, one has to spend a fair amount of time building OO wrappers and mappers for incoming and outgoing data that does not use the same (or no) objects as the internal system. OO systems are in essence Xenophobic.

When a new language fad replaces OOP, how do you convert legacy Java objects (or their state/data) into Zamma-2008 objects? (I will eat a week's pay if OOP is still in vogue in 2015.) How many times do different programming languages and systems have to share data? Very often in the real world. Thus, it is usually safer to keep large data pools separate from methods. Everybody clamors for open systems; why not open data which is not tied to a particular programming paradigm?

Tools like Crystal Reports and data mining systems are much easier to work with if behavior and data are kept more or less isolated.

Some have suggested that object service interfaces like CORBA or COM will satisfy this need. However, these still rely on execution of some language or executable. They also require new interfaces be built for unanticipated requests. They may be satisfactory for external transfers, but not for the more highly-shared internal or close-partner information. Plus, there are non-OO equivalents of CORBA and COM. (DDE, EDI, and CGI + HTTP are some examples.)

OOP is best used for situations where an OOP program or language should knowingly and exclusively have nearly complete control over the birth, processing, and death of data items (objects). I call this the "cradle-to-grave limit" of OOP. OOP should not be used where sharing is important.

See Also:
Chapter 31 Review - OOSC2
Control Panel Analogy (T.O.P.)

Why OOP Reminds Me of Communism

Cliche-Oriented Programming

Communism

Economic communism spread like wildfire in the first half of the 20th century because it had such appealing ideals and ideas. Like OO, these ideals were very seductive on paper. Intellectuals all over the world were drawn in by its concepts in droves. However, the complexities and dynamism of human nature proved not to favor economic communism as a productive model when it hit the road of the real world. It was the test of the real world that deflated the socialism hype, not intellectual analysis for the most part.

> **Perhaps economic communism allows for more equality, but mostly by making everybody equally poor. Further, it forms a kind of "social currency" based on schmoosing and favoritism that is nearly impossible to objectively measure and tax. Thus, it is still not really class-less, but simply makes classes harder to measure because the "currency" is**

> mostly social in nature. In other words, it does not eliminate inequality, but instead makes it harder to identify.

I am frankly jealous of OOP's cliches; or to be more specific, jealous of OOP's "cliche-ability". The cliches sound so convincing on first impression, yet take many pages and many tedious change-scenario analyses to de-hype. Like socialism, bashing capitalistic greed is easy to do and makes for great, gut-level sound-bytes. A counter-argument to such is rather long, and assumes a decent education and attention span. Take this quote from a top-ranking comment from a *slashdot.com* discussion about an earlier version of this very website:

> "[OOP] allows you to specify several functions, abstractly, that are required for a class to implement the Interface. The implementation of these functions is class-specific: for example, all clothes implement the Wearable interface, but you would not want underwear and shoes to have the same implementation of Wear(). However, in [OOP], you may specify a function to take a Wearable object, and [may] not need to specify any further. This abstraction level is why OOP does, in fact, better model the real world."

I believe the catchy sound-byte quality of this analogy is part of the reason why the message was ranked so high by the readers. It is hard to compete with a paradigm that is so photogenic from a cliche standpoint.

> Here is a summarized debunking of the clothing example: Modeling (clothing) products using hierarchies instead of sets is often problematic on a large scale, and OOP has nothing better than p/r for set management. Sets are more general-purpose than trees, and thus more adaptable to unexpected changes and orthogonal trait management. Further, most information about the products would be parameterized (factored into attributes) and stored in a database and set up so that operators (non-programmers) could key in data about clothing without having to write OOP classes.

Protocol Coupling

I once went to apply for a business license. Before the license could be evaluated and granted, I had to fill out layers of other forms that had almost nothing to do with my planned business. It was like reeling in one fish to find out that there were ten other fish attached to the first fish via ten other fishing lines, and that I could not end the fishing trip unless I either aborted the whole trip or finished pulling in all ten fish. (If you like fishing you may enjoy such effort. However, don't forget that many lakes have quotas.)

This reminds me of the tendency of OO to rely too heavily on Protocol Coupling. In OOP this is where one class depends too heavily on the interface of another class. One cannot understand the first class until they understand the second. Further, understanding the second class may require understanding yet another, third class. Thus, you have a "grok-breakdown" chain-reaction in the heavier cases, and mild confusion in lighter ones.

Further, the class depends on the other protocol to work. If you change or remove the other class, you risk ruining the existing one.

It is true that protocol coupling is indeed not necessary in OOP, as OO apologists happily point out. However, removing it from OOP programs makes them look less and less like OO and more like procedural programming. This pattern pops up in other ways also, such as reducing reliance on hierarchies to avoid the myriad pitfalls of modeling with trees (discussed below).

Note that protocol coupling can also happen in procedural/relational programming in various ways (see link). However, in practice it is either less common, uses "lighter wires" such as tables, or the interface itself hides the unnecessary details from the user of a routine.

OOP apologists keep saying, "Well, if doing X creates problems, then don't do X". However, following such advice would leave most OOP programs stripped of characteristics that make OOP programs different from procedural/relational ones. It repeatedly seems that the features which make OOP unique are also the ones that cause the most problems.

In one debate I kept finding problems with the OOP version presented using various change scenarios. So, the OO fan kept re-arranging the methods and structure to reduce the places that needed changing in case variations on the given scenarios popped up again. Finally the OO fan said, "See, it can now handle all of your changes well." My response: "Do you realize that what you now have is a nice procedural module? Toss a few 'self' or 'this' references, and it would almost compile as procedural code." (Paraphrased)

For more details, please see:
Black Boxes and Protocol Coupling (with examples)

Hierarchy-Happy

Hierarchies, Taxonomies, and Nesting Carried Too Far

The inheritance and aggregation model of OOP tends to assume the world can be cleanly modeled as hierarchical classification and/or nested structures, or at least into mutually-exclusive divisions (a flat tree), when in fact this is often not the case. (Yes, it can be forced into such structures, but the results are not pleasant.) Even a company management hierarchy is often defined in a nonhierarchical way when management plays political games or decides to use the matrix approach.

taxonomy styles

Several times I have seen what seemed like simple hierarchical structures morph into something a bit different when new government regulations or management changes come along. Although it is true that OOP does not have to use hierarchies and "IS-A" relations, it would lose one of its alleged great selling points (on paper) without them.

Hierarchies can be divided into (at least) two categories: static and dynamic. Static hierarchies are classifications that do not change much over time. These include animal classifications, chemistry, physics, and geometric shapes (circles, squares, cones, etc.). Static hierarchies are most often found in nature or occasionally from slow-moving standards bodies.

Dynamic hierarchies are much more common in business applications. These include company structure, product classifications, employee classifications, plant operation specifications, and so forth. It is my contention that the inheritance model works quite poorly on dynamic classifications.

Some of the bigger problems with inheritance and IS-A modeling are:

1. **There are often multiple orthogonal candidates for subclass divisions.**

2. **The features that make up the potential subclass divisions are often recombined in non-tree ways. Catdogs are real in the business world.**

3. **The range of variation often does not fall on existing method boundaries. For example, only 1/3 of a new variation may be different for a given method. In other words, how do you override 1/3 of a method? This may end up requiring altering many sibling methods in a domino-like "polymorphic splitting cascade".**

4. **Inheritance Buildup - Rather than alter the root or base levels of the tree (which risks unforeseen side-effects), the programmers often end up extending the inheritance tree to subclass the changes. Over time, you get a mess. (Or spend your time rearranging code, which has been given the convenient euphemism "refactoring" in some fan circles.)**

5. **Users often maintain real-world hierarchies, such as product categories and accounting codes, via hierarchy edit interfaces, and not programmers writing subclasses. In other words, the hierarchy nodes are stored in a database, and not in program code.**

6. **Hierarchies are often just one of many possible views of relationships. Consider an invoice model with a "header" portion and a "detail" portion for line items. ("Header" may be**

misleading here, but it is common terminology for a general invoice record.)

In OOP one often would have 2 classes; one for the header and one for line items. To model the line item relationships, you would perhaps have an array/collection of all the line items for a given invoice.

```
class line_item {.....}

class invoice_header {
   private line_items is array of line_item
   ...
}
```

However, the flaw in this is that you have to go through the invoice header to get at the detail. Sometimes you want to look at the detail regardless of its "parent." For example, you may want to count or sum all of product X that was shipped in a given month. In that case, you don't care about the parent invoice. (SQL: `"select sum(price) from line_items where product_id=X and month(shipDate)=Y"`)

> **Note that the different views may be needed for more than just ad-hoc queries. There could be regular reports on sales by product category, delivery delays (shipping orders are not always 1-to-1 with invoices), sales-by-vendor, sales-by-weight, sales-by-quantity, etc. In my opinion, it is risky to assume that one view aspect will always be the strongest or only aspect (such as by-invoice in this case).**
>
> **Another example of this is time-sheet detail lines. In one consultancy company, the project worked on for a given detail line was a more commonly-used relationship than the time-sheet or employee entity. Thus, modeling via a code-wired header-detail hierarchy could have been a rather large burden (code-wise and CPU-wise) if every project-oriented data request had to first go through the header (parent).**
>
> **Note that both of these examples do not involve OOP inheritance. Abuse of hierarchies is not just limited to explicit inheritance. The "hierarchy" here is the hard nesting of structures. Also note that some languages can map array syntax to external or shared data sources. In such cases, some of the criticisms above may not apply.**

These kinds operations make me hesitant to "hard-wire" a hierarchy into a model. Often one does not want to travel through trees to get stuff that does not relate to the tree. (File directories also often suffer from this flaw.) Relational thinking is more adaptable this way because it tends not to say that one relationship is more important than another; thus, less artificial relationship favoritism. A tree is only one of many possible views/relationships of data. (Note that the different views may be simultaneous in many cases.)

In business you often look at or use the same data in many different ways. Getting too "taxonomy happy" is a hindrance to this.

Some will argue that overuse of inheritance or hierarchies is the fault of the programmers, and not the fault of the OOP paradigm. However, why build in a complex organizational paradigm like inheritance when its real-world use is quite limited? It just invites misuse. Inheritance is probably the most misused OO feature. Complexity should be "spent" with great care. The bottom line is that OOP's brand of inheritance should not be a base part of a language that claims to have a general purpose focus. It is abused by inexperienced or ignorant programmers far more often than the times that it actually helps software organization for the long run.

Some might also contend that inheritance should be kept in a language for the rare times when static inheritance does occur. I disagree because the very fact that static hierarchies don't change very often implies that an organization paradigm, such as OOP inheritance, is not an important issue. If something rarely changes, then change-management of it is not something to spend too much time or language bloat on. Language complexity is better spent on stuff that occurs often.

See Also:
A Banking Example
More on Inheritance
Subtype Proliferation Myth
Fixing Bad Taxonomies

Alternatives to Trees
Nesting and Abstraction

Six Polymorphism Killers

Problems with or movements away from polymorphism usually fall into one of these categories:

1. **Mutual-exclusiveness of the polymorphic divisions (taxonomy) is often fleeting or falsely assumed. See the Bank Fee example for an illustration.**
2. **Multiple competing (orthogonal) or changing division/taxonomy aspects. (See "Noun Shuffle" below.)**
3. **Changing boundaries of influence.**
4. **Discrete distinctions may turn out to be continuous, such as a number instead of a "type".**
5. **Some divisions morph together to become one.**
6. **Subclass differences are parameterized and turned into instances and/or data records because of the volume involved.**

IF statements, case-blocks, and data query expressions are usually more flexible in dealing with changing dispatching (rule triggering) criteria; even multiple aspects can participate in an IF expression. It is very tricky and messy to achieve the same with polymorphism. Changing the IF criteria does not require moving the IF block.

Polymorphism is just too narrow a concept to be flexible. It requires a single, clean, stable taxonomy/division-criteria to be effective, and those are hard to find except perhaps in nature (geometry, physics, etc.).

OO fans often act proud when they can force ("find") a taxonomy or division of things which are best left independent. It is true that almost anything can be forced into taxomies if one tries hard

enough. However, contrived taxonomies/divisions are not very change-friendly in the long run. Many are falsely taught that forcing taxonomies/divisions results in "better abstraction". One should use great care when basing a design on mutually-exclusive groups or sub-types.

See Also: People Types Experiment

IS-A-Mania

Related to inheritance (above), IS-A-Mania is the excessive promotion of one aspect/criteria above other candidates.

aspect inflation

In certain "natural" domains like geometry and chemistry this may be appropriate because God (or Mother Nature) does not change those very often . However, my experience is that classification criteria or "usage paths" of items often change in the business domain.

HAS-A relationships are simply more flexible. I agree that there are some potential compiler checking features and syntactical shortcuts that IS-A can provide, including popular forms of polymorphism. But adaptability is usually more important. IS-A is a stronger modeling statement, and thus is harder to undo than HAS-A.

The bottom line is that HAS-A relations can easily serve as IS-A relations with little or no change, but not the other way around. And, OO loses much of its differences under HAS-A. Classes become little more than procedural "modules" under HAS-A.

Some places where IS-A can cause problems:

- **Business modeling, where classifications are often very dynamic, overlapping, and capricious.**

- Collection Protocol Taxonomies. Trees, stacks, queues, etc., can easily morph into or share aspects of other collection "types". Thus, it is best not to hard-wire your code with the assumption of a narrow collection "type". Network and GUI taxonomies also often seem to suffer this disease.

- Containment and Aggregation force most accesses and queries to go through the parent object, regardless of whether the parent is significant or not for a given operation. (See also the invoice-line-item example above.) In essence you are hard-wiring in one access path.

- The over-use of code patterns to limit the code structure to one of many other possible orthogonal pattern views. IOW, something "has-a" pattern, but some see it as "is-a" pattern, limiting other legitimate pattern perspectives.

- Code Management - Document management systems have learned that searching and classifying under multiple aspects/criteria is very useful (think about using web search engines). However, OOP still clings to the old-fashioned notion that one aspect, the class, is sufficient.

- Forced Single Entity Association - Fitting every operation into one and only one entity (class), when in fact many operations involve multiple entities and can easily need to be changed around.

- Hard Partitioning - Hard-wiring divisions by one aspect when in fact multiple division aspects (or views) may eventually be needed.

The Real World is Multifaceted and Relative

See Also:
Aspects
Cleaning up Taxonomania in API's
Abstraction

Granularity Problem

There is a problem pattern to much of OO philosophy. It depends too heavily on units or "chunks" that are too large to handle spot-level differences, variations, and changes. I call this the "granularity problem".

My favorite political analogy for this is having to vote for a party or candidate when what you really want to do is vote on specific issues like abortion, defense spending, social programs, etc. The "granularity" of voting is often not fine enough. A given candidate will rarely support all the same issues that you do. An example manifestation of this is the need to override only one line instead of a whole method.

One can say that sub-typing can be unnecessary "coupling of features". OO fans often brag about how OO allegedly reduces "coupling", but forcing things into groups that may not belong there in the long run is also a "coupling sin".

This kind of "lump thinking fallout" can be found in hierarchical/divisional taxonomies (A.K.A. "large sub-types"), method boundaries, noun-based "encapsulation", and strategy isolation. (Most of these were already briefly mentioned above. The hyperlinks go into more detail.) There are OO "workarounds" to some of these problems, but the cure is often worse than the disease. I usually find procedural techniques more friendly with regard to many kinds of changes that don't fit the boundaries or coverage of typical OO divisions. Changes that would only require the simple insertion or changing of an IF statement expression in procedural/relational programming may require moving OO methods from one sub-class to another, or splitting a method and all it's same-named polymorphic "cousins" into smaller chunks. In other words, more work and more code changes. Too much "shuffling around".

I suppose that the intellectual appeal is that you can manipulate and manage things in larger or more "structured" chunks. You know,

"hiding details", "encapsulation", and other buzz-phrases. I am not against the goal of this in any way. It is just that OO's approach to these goals falters when the real world does not fit into or change within the confines of the boxes OO lays out.

Do the Noun Shuffle

Encapsulation or capsule-hopping?

One of the biggest differences between OO and procedural/relational is that OO tends to group operations around nouns (entities), while p/r tends to group around tasks (at least the code part). I tend to find task grouping more "invariant" (stable) than noun grouping.

For example, suppose we have a certain feature in an invoice process:

```
sub processInvoice(rec)    // rec = record handle
  ....
  if rec.hasFeatureX then
     foo bar
  end if
  ....
end sub
```

Suppose feature-X is originally associated with customers (that is, specified per customer). Most OO designs would put the "foo bar" block of code into the Customer class(es); probably as a method.

Suppose later that customer is not fine-grained enough so it is moved to the invoice level. In the p/r version, either the SQL query statement would have to be changed, or the IF statement criteria. However, the OO version would have to then move the "foo bar" block (method) to the Invoice class(es).

Later on, it might be discovered that invoice-level is still not fine enough, and that the feature has to be moved to the invoice detail

level (line items). Again, the p/r version would only likely require a change to the IF criteria or the SQL query. Yet, the OO version will likely require moving the "foo bar" block to the InvoiceDetail class(es).

One OO fan claimed that moving code around is not a big problem because his "wonderful OOP IDE tool" made it easier. However, any paradigm can build crutch tools around its particular weaknesses. Regardless, moving is a costlier change than altering a few lines of code in-place. Plus, moving makes it harder to find code because you forget which "noun" to look in.

An even bigger monkey wrench is criteria that involves multiple nouns:

```
  if noun2.a OR noun3.b OR noun4.c then .....
```

Darts or dice to decide which noun class gets this? (Some OO fans will say to make Feature-X it's own class or method. However, that is a rather procedural decision.)

Even if you think I am exaggerating, task-oriented code grouping is at least not objectively inferior to noun-oriented grouping and taxonomy-based dispatching. Changing the criteria that "triggers" business rules is quite common in my experience. The p/r approach better decouples the dispatching criteria from the behavior, since the dispatching criteria often changes.

For more examples, see also:
Aspects (gov. invoice example)
Tasks and Modeling (banking example)
Modeling English Sentences

Assuming Idealism

Chaos vs. Order

Many OO books and even some OO fans suggest that OO only shines under fairly ideal conditions. These conditions may include:

- **Sufficient training and mentoring**
- **Sufficient incentives for long-term planning and/or good OO (see Planning article)**
- **Sufficient understanding of the domain/task**
- **OO-friendly database system**
- **Low employee turnover**
- **Good code viewer/editor tools**
- **Sufficient budget for good planning**
- **Good project management**

It may be time to focus on languages and paradigms optimized for the less-than-ideal conditions. There is a complexity cost to OO. Thus, if its payback is not sufficient, it may be a drain.

Note that this point of view is not necessarily suggesting that OO is "bad" per se, but that it has a different area of optimization than other paradigms.

Research Lacking

Furthermore, there is very little research on whether non-OOP languages can be used such that reuse and flexibility can be increased to the same levels as properly-used OOP. In other words,

the industry keeps adding new languages and features to solve the same problems instead of improving use of existing languages. Many are coming to believe that most of the computer industry is led by hype instead of careful research. OOP spread like a weed before there was any proof that it could or would be beneficial. (And the current proof is highly conditional). We could not find any comparative studies of OOP and other methods in the real world.

A decent study should have these aspects:

1. **Compares OOP to other techniques and languages, not just self-standing OOP successes.**

2. **Sponsored by an organization that has no vested interest in OOP technology.**

3. **Uses a wide variety of applications.**

4. **Considers factors like existing training levels, training curves, and investment return timing expectations.**

5. **Uses unbiased sampling techniques that do not bypass case successes or failures due to embarrassment, fear of obsolescence, PR agendas, or other social factors. (A tough but necessary step.)**

6. **Is carefully reviewed by secondary entities, and preferably open to the public.**

Some studies that are often cited were very limited in scope. For example, a large project was converted from C to C++. Allegedly, the new codebase was much smaller.

I consider this a poor test because C is a fairly low-level language, while C++ can be given higher-level constructions. I have, for example, converted a C music sequencer into XBase with a great improvement in productivity and a great reduction in code size. (Almost any language is an improvement over C. Its strengths are in

OS portability and speed; and little else. However, C fans may disagree.)

Note that I must reject anecdotes as evidence because:

1. **There are plenty of anecdotes not favorable to OO (although companies are often too embarrassed to admit to them publicly.) There are not even good "anecdote surveys" around to know the ratio of pro-OO to anti-OO anecdotes.**
2. **The participants may simply have an "OO-mind" (see below), which may not be representative of the rest of the programming population.**
3. **They may be comparing good OO to bad procedural/relational programming.**
4. **The details are usually not available for open inspection.**

See Also: Goals and Metrics

One Mind Fits All?

It seems that paradigms are like colors, cars, and clothes--everybody has their favorite. What works for one person, culture, or application may not work well for another. To say that object paradigms are the best organization tool in a general sense strikes us as arrogant. It is like saying, "Yellow is the best color for everybody."

desk

What if somebody walked up to your desk and started telling you how to organize your desk's contents? Most would say that this is silly. However, this is almost exactly what is being done with OOP. Paradigms are about organizing software information to make it easier for developers to work with it. It has nothing to do with users nor the outside world (as long as specifications and good UI design principles are followed). It is all about modeling the developer's mind, and no OO proponent knows my mind better than me.

You can tell a wood craftsman how you want the end result, but you are foolish to tell him how to construct it. If you are also a wood crafter, then you can offer tips, but only each individual knows what works best for themself in the end. (There are issues of maintenance teams, but this is too involved to discuss here.)

There are other paradigms that are not heard about in the press and brochures, or perhaps not yet fully explored, that could be an equal or better approach for an entity (person, culture, or application) than object paradigms. Even procedural programming is under-explored in some cases. Also, many programmers mistakenly use C as representative of procedural languages and are glad to move on to C++, which does offer more power simply because it has more features and vendor tools for it.

The OOP prophets out there promoting OO may indeed find objects better for themselves or their teams. However, they should not extrapolate this to everyone. Just because they think in terms of OOP objects does not mean everyone else does. I find that software designers and builders think very differently from each other. The human mind and business cultures come in many flavors, so perhaps our paradigms should do the same. (This makes for an interesting debate topic: should one change the paradigm to fit one's mind or change one's mind to fit the paradigm?)

I often hear OO proponents say something along the lines of, "I always built my software in an OO-ish way, I just did not have a name for it. Then OOP formally came along and I was happy to see it built into the language."

I, on the other hand, think best in terms of tables and like to organize software projects using control tables. I have seen others do the same without ever knowing that they were using a "table paradigm." While OOP attempts to keep you on course by putting up edge barriers, control tables keep you on course by giving you a better view of the course. They both have similar goals, but use very different approaches to achieve these goals. If Table Oriented Programming were simply given a recognized identity, other tablers might also come forward. Another table fan had this to say:

> *...I was always told that OO was an intuitive way of thinking. Although I develop daily in OO languages, I do find OO rather unintuitive (although I manage). ... If I analyze a problem, I draw tables not objects and classes, I think in tables, those are much more intuitive to me. Even when I program in OOP, I rely as much as possible on tables. ...it's nice to see that I'm not the only one who 'thinks' in tables. ...*

See Also: Physical vs. Mental Metrics

Cult Oriented Programming May Just Work

In chapter 3 of *The Rise & Resurrection of the American Programmer*, page 63, Edward Yourdon gives some statistics from studies about OOP projects. It seems that early Smalltalk projects, for example, were quite successful, but later ones did not score higher than the average language score. Yourdon's speculation for this change is that early projects had more Smalltalk (language-specific) "fanatics" on them, but later ones had a more mixed crowd. Smalltalk became the "in" language for a while and people rushed into it because that is where the money was. The earlier Smalltalkers chose Smalltalk mostly because they liked it.

This implies that a programmer's personal preference is more important to getting results than the language or paradigm itself. In other-words, if a programmer can work with the tools and languages that he or she most prefers, then productivity and quality may be much higher than from passing down edicts based on a blanket choice of tool, language, or paradigm.

My personal experience tends to confirm this. When given tools that allow me to maximize my use of Table Oriented Programming, for example, I seem to get things done much faster and am more satisfied with the results. I have heard fans of other languages and techniques say similar things.

Thus, if you really want to get things done, hire a zealot or a team of like-minded zealots. "The best tool for the mind" might be just as

important as "best tool for the job". Regardless, Zealot Oriented Programming is an area ripe for more research.

> One caveat is the some zealots focus too much on following a certain methodology rather than getting something done. Thus, you have to be careful about distinguishing ideological zealots from practical zealots. It just may be that different forms of zealotry are needed for different situations. For example, the Eiffel language and philosophy involve building many "safety-checks" into the software to assure compliance with specifications. This philosophy and it's corresponding followers may not do well if fast turnaround time is required, but may be ideal for life-support hospital applications or missile applications, where mistakes can be deadly. (See also Long-Term Issues.)

Split Ends

Did Betsy Ross ever worry about split ends in her hair? Probably not. However, if she was alive today she probably would. Advertisers found a solution to a small problem, used promotional techniques to make it into a big problem, and are laughing all the way to the bank. Thanks to commercials whose media techniques are outstanding, even guys now recognize a woman's split ends. Before, most guys had no clue. (Ironicly, split ends are often caused by excessive use of other beauty techniques.)

OOP appears to be doing the same thing. It takes small or imagined problems and magnifies them to the point of irrational paranoia. Global variables*, case statements, and other straightforward procedural techniques are now officially banned by the Church of Objects.

It does not matter how many OO wrappers, mappers, meta structures, meta classes, adapters, or converters it takes, those evils must be stamped out at all costs. It does not matter that the resulting OOP program is bloated, tangled, and confusing; as long as the Great Evils are eliminated.

Sure, global variables sometimes cause name conflicts and other problems, but is sure bloat and roundaboutness better than a relatively slim possibility of a name conflict?

> *** There are procedural ways to group routines and scope to reduce or eliminate what would be global variables, but these rarely get any attention. Perhaps because name conflicts are not common enough to justify the extra syntax and expert proceduralists know this. Often times globals can be reduced or eliminated by using tables to store shared values or having more flexible access to tables. I am not promoting mass use of globals. Used judiciously, they are rarely a problem in small and medium applications in my experience.**

Bloat creates errors because of confusion and distracting clutter. Why are the errors introduced by having too many airbags somehow better than errors introduced by having none? Double Standard!

It reminds me of the cartoons where the main character destroys his entire house trying to kill a single mosquito. Perhaps that is where the term "buzzword" came from.

Another variation on this theme is the "Meteor Insurance" game. Many of the "best" OO demos actually are solving patterns of problems that do not occur very often at all, at least not in my niche.

See also:
Subtype Proliferation Myth
The Driver Pattern

Case of the Hidden Case (or Switch) Statements

Many OOP proponents brag about OOP's alleged elimination of "case" or "switch" constructs. Case statements are a favorite (straw-man) whipping boy for OO textbooks and fans. In reality

what is being done is that the case statements are being broken up and spread around the program so that they are not immediately visible. Their equivalent is still there, but simply morphed and divided.

This is similar to a child's trick of spreading the unfinished food around the plate so that it looks like the child finished his or her food. Thus, the parents may accuse the child of being messy instead of not finishing his or her food.

The equivalent of the Case decisions are still there in OO programs. They are simply spread around to such places as object assignment and subclass comb structures. OOP simply inverts the nesting and grouping structure that is found in traditional (procedural) programming. Inverting is not the same as eliminating a structure. Further, putting methods together based on variation (subtype) often pulls them apart with regard to behavioral grouping. Behavioral grouping is at least as important. For some undefined reason, OO fans tend to dismiss the importance of behavioral grouping.

Examples which show sub-classing "reducing the quantity of spots that need updating" over case statements are misleading because they tend to ignore the fact that sub-classing increases the quantity of changes for the addition of new operations. In other words, sub-classing favors subtype-oriented changes at the expense of operational changes. It is more or less a zero-sum tradeoff, and not the free lunch that many OO proponents suggest. I see the OO side of the story all the time, but rarely are the down-sides mentioned. Further, real world occurrences of case statements usually don't match the OOP textbook pattern of repeated case-lists (except in poorly-engineered code).

For more details, please see:
Shapes Example (Case Statements)
Aspects
Meyer's Single Choice Principle

Wrong Credit

OOP sometimes takes credit for ideas that are not necessarily part of OOP. For example, some criticize the variable scoping rules of procedural languages, saying that OOP improved it. However, some procedural languages like Pascal already allowed multiple levels of variable and procedure scoping before OOP became a mainstream fad.

Having variable parameter types and quantities has been part of many interpreted procedural languages a good time before OOP became a mainstream fad. For example, in XBase you can use the Type() function to query a parameter type. (It lacked formality, but it was there.)

Many of the table oriented approaches that I recommend are sometimes called "OO in disguise" by some OO fans. However, these have been in use in various forms long before OOP was invented in 1967. (Improvements in relational, indexing, and table browsing UI technology make them easier to do now.)

Memory Recovery

At least twice I have heard OOP fans say that OOP allows programs to automatically recover memory, also known as "garbage collection." However, OOP does not have a monopoly on this.

They are usually making the mistake of comparing C to C++. C is just too low-level to serve as a decent representative of the potential of procedural programming.

Garbage collection in procedural programming usually comes about by having the end of a routine or end of the program automatically close out the structures and mark the memory as available for other uses. Example:

```
sub X {
   local  Y
   global Z
   Y = openTable("foo")
   Z = openTable("bar")
   doSomething_with(Y)
   doSomething_with(Z)
}
```

In this example, the table referenced by handle Y is automatically closed and de-allocated when routine X finishes. However, Z is not closed because it is connected to a global variable. It will automatically close at program end, though. (There would also usually be something like a "close(Z)" operation that could be used if disired.)

This is one of many approaches. There are different techniques for implementing this. In some languages the programmer can define or access "scope-end" triggers for variables (including structure "handles"), in others it only comes about when using built-in structures.

> **The issue of "de-allocating" external resources versus internal resources can get quite involved. Java's "Finalize" method is sometimes cited as a way to automatically de-allocate external resources automatically, but is nearly useless in practice because of the nature of Java. Another sticky issue is why/if external resources should be managed differently than external ones in the application code. The idea of swappable engines seems counter to this idea. I am planning on writing more about these issues in the future.**

Further, if you use tables instead of arrays and linked lists, then memory management is often handled by the table engine and not the application programmer. A decent collection system can handle and "have ready" many different operations. It reduces a lot of wheel reinventing.

Lost Art?

I am beginning to get the feeling that many people are forgetting how to do good procedural programming and blaming the paradigm for their shrinking knowledge. The above claim about memory recovery is just one example.

I just heard someone say that they found an old procedural program of theirs that used too many global variables and too many parameters. Rather than blame his bad programming or lack of knowledge about procedural/relational organization techniques, he blamed the paradigm and used it as a sorry excuse to proceed with OOP.

OO fans see no problem with reading 1,700 page books about "proper OOP", yet never touch a one-page guide to "proper Procedural/Relational programming" (if they make such a thing anymore). In my opinion good procedural/relational (p/r) techniques are much easier to learn than OOP because they are less abstract.

However, nobody is championing procedural/relational anymore because it is out of style and a risk to career face right now. Americans have a habit of switching IT fads before perfecting the current one.

Also, many bad procedural/relational memories came from nasty languages like C. C is about as primative as one can get above assembler.

A common misconception is that one has to change all the procedural code if a DB field type or size changes. For example, if a percentage rate changes from an integer to a float/real, then OO fans often claim that it causes a procedural code change cascade. However, this usually only matters in strong-typed languages, like C, Java, and Pascal. (Weaker typing is better for p/r usage in my opinion.)

The fact that these claims come up time and time again testifies to the p/r ignorance that is floating around out there. These OOrban

legends spread around like bad gossip via greedy snake OOil salespersons.

It is also sometimes stated that procedural/relational software cannot factor as well. I have found that this is not true. Given a few dynamic features and flexible variable scoping control, I can factor any procedural/relational program as small as an equivalent OO program. (The result may not always have the same built-in protection against forgetful maintenance programmers, but it is not more code.)

Another false bash of procedural/relational (P/R) is given by Bertrand Meyer (Ch. 5 of OOSC2). He sets up a false dichotomy of top-down versus OO. I have many times written P/R software without locking the units into the kind of arbitrary top-down ordering he describes. His dichotomy is purely fiction. Perhaps he likes OO so much because he is such a crappy proceduralist. Just because you cannot ride a bicycle does not mean that a horse is better for all.

See Also: Procedural/Relational Tips

Too Many Fingers?

As a general rule, OO modeling appears to work best where most or all of the information for a given activity comes from a single entity (class). But, that is a rare thing in business processes in my experience.

Perhaps this is why components are much more popular than OO business modeling in actual applications being produced.

Components tend to have self-contained sources of state or data. However, business processes tend to need state (data) from a wide span of sources (other entities or classes). In other words, business processes have their tentacles all over the place. Things like month-end processing or calculating customer and volume discounts can often involve references to 4 or more entities in many systems. Even

processes that start out under one entity often expand to multiple entities over time. (Accessors excepted. That is another issue.)

This is a major reason that OO business modeling is nearly hopeless. There is no one entity to associate primary behaviors with unless you arbitrarily pick one based on current references (which may change tomorrow). The remaining choices are either a bunch of confusing middle-men classes, or a class that is a behavioral class. A behavioral class is otherwise equivalent to a procedural module or subroutine.

By the way, I think that components can be done well in procedural contexts, just not in the C and Pascal-like languages often associated with procedural programming.

For more details, please see:
Aspects
Components
Business Modeling

Modeling the Real World

The modeling section has been moved to its own page.

Another Fad?

OOP is a lot like those idealistic development fads that failed to prove themselves in the real world. Expert systems, CASE, and 3-tier client/server technology are three other examples of technology that failed in the real world or took on a niche status. After the market place realizes the shortcomings of these technologies, the die-hard supporters always say the same thing: "They WOULD work if people simply used them properly." We are already hearing this from OOP apologists.

They are also using the line that it will improve to acceptable levels over time. We would prefer that it be perfected and proven before it's shoved down our throats, not after.

It now seems like IBM-based mainframes and big servers are back in style. Everyone thought they were dead. They became so unfashionable in the early 90's that some companies got rid of them out of embarrassment, not out of any well-studied need.

One of the reasons mainframes are almost vogue again is because the GUI interface (user's screen) of web-based systems is no longer directly defined by the operating system or the hardware. Generating HTML output or talking to a Java applet (Web technology) is almost as easy on a mainframe as it is on a desktop PC.

Who knows, perhaps DOS will be the next "thing" for web server programming platforms. After all, Linux is making command lines "sexy" again (even though it usually comes with a GUI for most tasks.)

None of our Business?

Even though regular business software development is probably the largest "niche" of all software development, it often at the bottom of the totem pole when it comes to software organization research and ideas. Most of the focus seems to be on scientific programming, systems programming, and other areas often associated with large university research.

In short, business programming does not get the respect that it deserves. Just because it has failed to produce the aura of "rocket science" that other niches have, does not mean that business programmers should have to put up with technologies and paradigms better tuned for other niches.

It is time to stand up and be heard.

> **Note that despite its reputation, business development can get very tricky. Often this is because of the complex, multi-aspect, dynamically-changing rules that can affect anything and everything. It is tough to find isolated units (encapsulation, black box, etc.) in order to narrow your focus. (See "Fingers" above.)**

See Also:
Business Modeling
Subtype Proliferation Myth
The Driver Pattern

Summary

It is hard to summarize such a complex, involved topic; but here goes an attempt anyhow. Most problems with OOP can be summed up in a handful of general principles.

1. **The real world does not change in a hierarchical way for the most part. You can force a hierarchical classification onto many things, but you cannot force change requests to cleanly fit your hierarchy. Just because a structure is conceptually simple does not necessarily mean it is also change-friendly.**

2. **There are multiple orthogonal aspect grouping candidates and the ones favored by OOP are probably not the best in many or most cases. OO literature is famous for only showing changes that benefit the aspects favored by OO. In the real world, changes come in many aspects, not just those favored or emphasized by OO. Encapsulating by just a single dimension is often a can of worms.**

3. **OOP's granularity of grouping and separation is often larger than actual changes and variations. OOP's alleged solutions to this, such as micro-methods and micro-classes, create code management headaches and other problems.**

4. **OOP has the habit of hard-wiring the model structure and relationships into the larger program code structure, as opposed to using more change-friendly and multi-view-friendly formulas or expressions (such as relational and Boolean). I lean toward virtual, local, as-needed structures rather than global ones. OO has failed to grasp the importance of relativism.**

5. **There is no decent, objective, and open evidence that OOP is better. It may just all be subjective or domain-specific. Software engineering is sorely lacking good metrics.**

6. **There is a large lack of consistency in OO business design methodologies. Procedural/relational approaches tend to be more consistent in my experience. (Group code by task, and use database to model noun structures and relations.)**

7. **Many of the past sins that OOP is trying to fix are people and management issues (incentives, training, etc.), and not the fault of the paradigms involved. Until true A.I. comes along, no paradigm will force good code. If anything, OOP simply offers more ways to screw up.**

A Challenge

Do you think I am full of horse droppings? I have offered a challenge to produce realistic business examples demonstrating OO's alleged superiority. So far a few attempts have failed. I often like to say to brochurish braggers, "put your code where your mouth is."

Communism also looked good in theory, but its base assumptions about human nature and change patterns were flat wrong!

"The good things about OOP [concepts] are often the information hiding and consistent underlying models which derive from clean

thoughts, not linguistic cliches." (from S. Johnson, see links below)

End the Hype and Start the Thinking!

Links and Related Documents

Internal Links

- To Main
- OOP Questions & Answers
- Table Oriented Programming
- Critique of Bertrand Meyer's OOSC2
- Code Challenge to OO Fans
- Short-Term Versus Long-Term and Planning Issues
- Tabled GUI's (alternative to OOP GUI's)
- Buzz-Words (Incoherentance, Entrapsulation, Polydwarfism, and others)
- An OOP Forum
- Competing Paradigms
- OOP's Goals and Metrics
- Aspects
- Why More Don't Speak Up
- Subtype Proliferation Myth
- The Driver Pattern A Narrow Niche?
- OOP Criticism Part 2 (includes Black Box and Component issues)
- OOP Questions & Answers
- Java Criticism
- Business Modeling
- Abstraction
- General Software Engineering Notes (not really OO-related)
- Procedural/Relational Patterns
- Change Patterns
- OO Myth Guide
- Code and scenario examples: Shapes, Bank, Publications, Data Translation, Challenge List

External Links

- **OOP Reuse Not High according to Dr. Dobb's - 5/7/1998, by L. Finch**
- **Other Comments on Reuse and OOP (Wikiwiki forums)**
- **Nuts to OOP (The "emperor clothes" reference was made independently)**
- **More on "Nuts to OOP"**
- **Objecting To Objects, by Stephen C. Johnson (A C++ perspective)**
- **OOP Paradigm Critique by Shajan Miah (It is a long article and I have not fully reviewed it yet.)**
- **Critique of OOP by James M. Coggins**
- **Reuse Is Tough (An InfoWeek article not really about OO, but a good reality check)**
- **No Silver Bullets by Warren Keuffel**
- **BlueTail's "Why OO Sucks" - Quote: *If a [paradigm] is so bad that it creates a new industry to solve problems of its own making, then it must be a good idea for the guys who want to make money.***
- **Paul Graham Excluding OOP From New Language - Paul co-started a very successful e-stores company using LISP. To my chagrin, however, he does not like databases either. LISP shares my philosophy of treating larger-scale data organization similar to code organization. I just think that nested lists are less "grokkable" and flexible than (good) tables.**