October 7th, 2014

In **Technology**

6 Comments

Object Oriented Programming is an expensive disaster which must end

(written by **lawrence**, however indented passages are often quotes)

The **No True Scotsman fallacy** leads to arguments like this:

Person A: “No Scotsman ever steals.”

Person B: “I know of a Scotsman who stole.”

Person A: “No True Scotsman would ever steal.”

Person A is thus protected from the harmful effects of new information. New information is dangerous, as it might cause someone to change their mind. New information can be rendered safe simply by declaring it to be invalid. Person A believes all Scotsman are brave and honorable, and you can not convince them otherwise, for any counter-example you bring up is of some degraded Untrue Scotsman, which has no bearing on whatever they think of True Scotsman. And this is my experience whenever I argue against Object Oriented Programming (OOP): no matter what evidence I bring up for consideration, it is dismissed as irrelevant. If I complain that Java is verbose, I’m told that True OOP Programmers let the IDE take care of some of the boilerplate, or perhaps I am told that Scala is better. If I complain that Scala involves too much ceremony, I’m told that Ruby lacks ceremony. If I complain about the dangers of monkey-patching in Ruby, I’m told that True OOP Programmers know how to use the meta-programming to their advantage, and if I can’t do it then I am simply incompetent. I should use a language that is more pure, or a language that is more practical, I should use a language that has compile-time static data-type checking, or I should use a language that gives me the freedom of dynamic typing. If I complain about bugginess, I’m told that those specific bugs have been fixed in the new version, why haven’t I upgraded, or I’m told there is a common workaround, and I’m an idiot if I didn’t know about it. If I complain that the most popular framework is bloated, I’m told that no one uses that framework any more. No True OOP Programmer ever does whatever it is that I’m complaining about.

There are many beautiful ideas that people associate with OOP. I am going to show 2 things:

1.) compared to other languages (lisps, functional languages, etc) OOP languages have no unique strengths

2.) compared to other languages (lisps, functional languages, etc) OOP languages inflict a heavy burden of unneeded complexity

Those features which are potentially good (data hiding, contract enforcement, polymorphism) are not unique to OOP and, in fact, stronger versions of these things are available in non-OOP languages. Those features that are unique to OOP (dependency injection, instantiation) are awful and exist only because OOP is awful.

I am taking an ecumenical, universalist approach to OOP. Below I will refer to all of these languages as OOP: C++, Java, Scala, PHP, Ruby, and Javascript. Is that fair? I know, from personal experience, some proponents of Java will complain that Ruby and PHP lack compile time data-type checking and therefore should not be considered OOP. And I know, from personal experience, some proponents of Ruby will argue that in Ruby everything is an object, whereas Java still has non-object primitives (such as integers), and therefore Ruby is more of an OOP language than Java. I know that some critics of PHP will argue that OOP features were bolted on to PHP and it should not be taken seriously as an OOP language. I know some people will point out that Scala is multi-paradigm and it is as easy to work in the “functional paradigm” with Scala as it is easy to work with the object oriented paradigm.

Given the diversity of the languages, and the lack of a standard definition, is it meaningful to talk about OOP at all? I would say yes. The need is urgent. OOP may be a poorly defined, amorphous concept, but it absolutely dominates the tech industry. Many software developers, and many companies, feel that OOP is the only reasonable way to develop software today. Any one who argues against OOP is immediately made conscious of the fact that they are arguing against the "conventional wisdom" of the industry.

I run into this when I go to a job interview. It does not matter if I interview for a Ruby job, or a Java job, or a PHP job, the job interviewers ask me if I know what OOP is. They ask me to define "encapsulation" and "polymorphic dispatch". These are standard questions, to which I am expected to give the standard answers. And when they ask me "What are the benefits of OOP?" I find myself wanting to give an awkwardly long answer, which consists "These are the 12 things that are supposed to be the benefits of OOP, but really OOP has no unique strengths." And so I am writing this essay, and in the future, when I'm asked questions like this at a job interview, I'll simply directly people to what I have written here.

Does any of this really matter? You could argue that I'm wasting my time, that I am writing a very long essay that merely engages in a bunch of semantic hair-splitting that benefits no one. But I would suggest that muddled definitions lead to muddled thinking, **in the manner that Orwell described**:

> A man may take to drink because he feels himself to be a failure, and then fail all the more completely because he drinks. It is rather the same thing that is happening to the English language. It becomes ugly and inaccurate because our thoughts are foolish, but the slovenliness of our language makes it easier for us to have foolish thoughts. The point is that the process is reversible. Modern English, especially written English, is full of bad habits which spread by imitation and which can be avoided if one is willing to take the necessary trouble.

On that basis, I would like to think that I do some good, to the extent that I'm able to take on the broad range of ideas associated with OOP.

This essay is long, and it would be even longer if I carefully qualified every sentence about OOP. Please note that, below, when I refer to a multi-paradigm language, such as Scala, as an OOP language, I am specifically referring to the OOP qualities in that language. And I would like you to ask yourself, if you use a multi-paradigm language to write in the "functional" paradigm, are you actually gaining anything from the OOP qualities of that language? Could you perhaps achieve the same thing, more easily, using a language that is fundamentally "functional", rather than object oriented?

On tech blogs and forums, there are a great many people who defend OOP, and who feel certain that they know what they are defending, despite the lack of any standard definition. **Consider this remark by "millstone" on Hacker News**:

> This article, like many that cheer functional programming, falls into a certain cognitive bias, that prevents it from seeing what OO is good at.

What is OO good at? Apparently millstone thinks that OO is good at being dynamic. millstone then criticizes the fact that Haskel has static type checking, ignoring the fact that Java, C++, C# and many other OO languages all have static type checking:

> Alan Kay wrote "The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be."
>
> To start to see what this means, consider the annoying String / Data.Text split in Haskell. String is very much in the "leave data alone" mindset, baring its guts as a [Char]. Now you're stuck: you can't change its representation, you can't easily introduce Unicode, etc. This proved to be so rigid that an entirely new string type had to be introduced, and we're still dealing with the fallout.
>
> Great and growable systems! The large scale structure of our software, decomposed into modules, not just at a moment frozen in time, but in the future as well. We are tasked with thinking about relationships and communication.

millstone then quotes the original article, and then makes clear they are really talking about " truly dynamic languages" and not OOP:

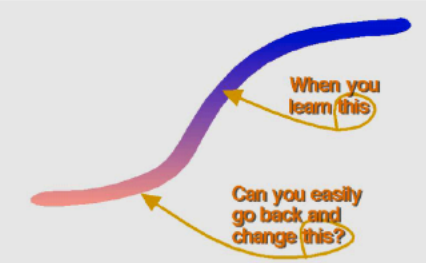
> *To come up with a better solution [for dispatching], Haskell and Clojure take very different approaches, but both excel what any OO programmer is commonly used to.*
>
> "Any OO programmer?" No way! OO as realized in truly dynamic languages exposes not just a fixed dispatch mechanism, but the machinery of dispatch itself, i.e. a metaobject protocol

There are plenty of OOP languages that have static data-type checking, and there are non-OOP languages that are dynamic, so millstone is not really talking about OOP at all, and yet millstone is certain that they know what OOP is. This is a problem that I run into fairly often: a fierce advocate of OOP who is using some idiosyncratic definition of OOP, which takes me completely off-guard.

millstone then quotes this part of an essay by Alan Kay:

Late binding

Until real software engineering is developed, the next best practice is to develop with a dynamic system that has extreme late binding in all aspects. The first system to really do this in an important way was LISP, and many of its great ideas were used in the invention of Squeak's ancestor Smalltalk -- the first dynamic completely object-oriented development and operating environment -- in the early 70s at Xerox PARC. Squeak goes much further in its approach to latebinding.



Late binding allows ideas learned late in project development to be reformulated into the project with exponentially less effort than traditional early binding systems (C, C++, Java, etc.)

One key idea is to keep the system running while testing and especially while making changes. Even major changes should be incremental and take no more than a fraction of a second to effect. Various forms of "undo" need to be supplied to allow graceful recovery from dangerous changes, etc.

Please note the irony here: millstone is quoting a passage that is critical of Java, and yet millstone is claiming this passage, about "late binding", shows what OO is good at. By this definition, Java is not an OOP language, which would surely be a surprise to Java programmers.

Again, there are OOP languages that have don't have late binding, and there are non-OOP languages that do have late binding. But for millstone, No True OOP Programmer would ever use a language with static data-type checking. To the extent that millstone is inventing a wholly idiosyncratic definition of OOP, their comments about OOP are wholly irrelevant to anyone else who wants to talk about OOP. And yet, in some sense, millstone is very common: I commonly run into software developers who have wholly idiosyncratic definitions of OOP. This can make it difficult to have a meaningful conversation.

How should we talk about a concept as amorphous as OOP? There is no standard definition, so the best we can do is survey a few different sources and gather up the main ideas. How should we conduct this survey? Two ways: first, a trip through history, listening to Alan Kay describe the roots of OOP, and then a look at what some current introductory materials are teaching beginners about the core ideas of OOP.

Alan Kay started with some brilliant observations about the changing nature of computing, and this fed directly into the beautiful ideas that OOP began with:

> One would compute with a [laptop] "Dynabook" in a way that would not be possible on a shared mainframe; millions of potential users meant that the user interface would have to become a learning environment along the lines of Montessori and Bruner; and needs for large scope, reduction in complexity, and end-user literacy would require that data and control structures be done away with in favor of a more biological scheme of protected universal cells interacting only through messages that could mimic any desired behavior.
>
> …Somewhere in all of this, I realized that the bridge to an object-based system could be in terms of each object as a syntax directed interpreter of messages sent to it. In one fell swoop this would unify object-oriented semantics with the ideal of a completely extensible language. The mental image was one of separate computers sending requests to other computers that had to be accepted and understood by the receivers before anything could happen. In today's terms every object would be a server offering services whose deployment and discretion depended entirely on the server's notion of relationship with the servee.

That is a beautiful vision, but is that the same thing as what we now call "object oriented programming"? In the above quote, **Alan Kay seems to be describing something close to what we would now call the Actor Model**.

> An actor is a process that executes a function. … Actors never share state and thus never need to compete for locks for access to shared data. Instead, actors share data by sending messages that are immutable. Immutable data cannot be modified, so reads do not require a lock. Messages are sent asynchronously and are buffered in an actor's mailbox. A mailbox is essentially a queue with multiple producers (other actors) and a single consumer. A particular actor is driven by receiving messages from the mailbox based on pattern matching.

Alan Kay's beautiful description of OOP bears no relation to anything that I have ever seen in the real world. Whenever I point out that what we ended up with is very far

from what we were promised, I am often greeted with a No True Scotsman defense: if only I did things better, or used a purer language, then I would experience Enlightenment, and suddenly I would gain all the benefits of True Object Oriented Programming. And yet, even Alan Kay seems aware of the fact that what we ended up with is far from what he started with. He is, of course, aware that his own language, SmallTalk, never became popular. **But what does he think of the languages which popularized object oriented programming**?

> Sun Microsystems had the right people to make Java into a first-class language, and I believe it was the Sun marketing people who rushed the thing out before it should have gotten out.
>
> …If the pros at Sun had had a chance to fix Java, the world would be a much more pleasant place. This is not secret knowledge. It's just secret to this pop culture.

Alan Kay himself has never been a blind, ideological defender of OOP. **He borrowed many ideas from Lisp, and he is open about his admiration of Lisp**:

> Kay characterizes SIMULA as a great transitional set of ideas. With SIMULA, Algol blocks could be used as independent things that could carry data and behavior. In 1966, Kay had just learned Sketchpad when he read an early paper on SIMULA by Nygaard and Dahl. Kay put a biological twist on what he would later call object-oriented programming. "Everything is a cell," he explains. "The main thing I added is that everything could be an object. There is an interface algebra that today might be called polymorphism. There was a collision of these ideas and LISP." Kay admires the great set of ideas present in LISP and refers to it as the "greatest single programming language ever designed."

My own experience with OOP involves long meetings debating worthless trivia such as how to deal with fat model classes in Ruby On Rails, refactoring the code into smaller pieces, each piece a bit of utility code, though we were not allowed to call it utility code, because utility code is regarded as a bad thing under OOP. I have seen hyper-intelligent people waste countless hours discussing how to wire together a system of Dependency Injection that will allow us to instantiate our objects correctly. This, to me, is the great sadness of OOP: so many brilliant minds have been wasted on a useless dogma that inflicts much pain, for no benefit. And worst of all, because OOP has failed to deliver the silver bullet that ends our software woes, every year or two we are greeted with a new orthodoxy, each one promising to finally make OOP work the way it was originally promised.

Though it is meant as parody of SOAP, **Peter Lacey offers what I regard as an accurate description of object oriented programming in the real world**:

> Dev: So it's simple?
>
> SG: Simple as Sunday, my friend.
>
> Dev: Okay, lay it on me.
>
> SG: Well, just like it says in the name, SOAP is used for accessing remote objects.
>
> Dev: Like CORBA?
>
> SG: Exactly like CORBA, only simpler. Instead of some complex transport protocol that no one will let traverse a firewall, we use HTTP. And instead of some binary message format we use XML.
>
> Dev: I'm intrigued. Show me how it works.
>
> SG: Sure thing. First there's the SOAP envelope. It's pretty simple. It's just an XML document consisting of a header and a body. And in the body you make your RPC call.
>
> Dev: So this is all about RPCs?
>
> SG: Absolutely. As I was saying, you make your RPC call by putting the method name and its arguments in the body. The method name is the outermost element and each sub-element is a parameter. And all the parameters can be typed as specified right here in Section 5 of the specification.
>
> Dev: (reads Section 5) Okay, that's not too bad.
>
> SG: Now, when your service is deployed, you specify the endpoint.
>
> Dev: Endpoint?
>
> SG: Endpoint, the address of the service. You POST your SOAP envelope to the endpoint's URL.
>
> Dev: What happens if I GET the endpoint's URL?
>
> SG: Don't know. Using GET is undefined.

Dev: Hrrm. And what happens if I move the service to a different endpoint? Do I get a 301 back?

SG: No. SOAP doesn't really use HTTP response codes.

Dev: So, when you said SOAP uses HTTP, what you meant to say is SOAP tunnels over HTTP.

SG: Well, 'tunnel' is such an ugly word. We prefer to say SOAP is transport agnostic.

Dev: But HTTP isn't a transport, it's an application protocol. Anyway, what other "transports" does SOAP support?

SG: Well, officially none. But you can potentially support any of 'em. And there's lots of platforms that support JMS, and FTP, and SMTP.

Dev: Does anyone actually use these other transports?

SG: Uhm, no. But the point is you can.

Dev: Fine. How 'bout this SOAPAction HTTP header, what's that for?

SG: To be honest, no one's really sure.

Dev: And these 'actor' and 'mustUnderstand' attributes, does anyone use those?

SG: No. Not really. Just ignore those.

Dev: All right, let me give it a shot.

(time passes)

Dev: Well, I could mostly make things work, but only if I stick with one SOAP stack. Also, I can't say I like the idea of remote procedure calls and serializing objects.

SG: Remote procedure calls! Serialized objects! Where did you get the impression that SOAP was about RPCs? SOAP is all about document-based message passing, my friend.

Dev: But you just said —

SG: Forget what I said. From here on in we pass around coarse-grained messages — you like that term, 'coarse-grained?' Messages that conform to an XML Schema. We call the new style Document/Literal and the old style RPC/Encoded.

Dev: XML Schema?

SG: Oh, it's all the rage. Next big thing. Take a look.

Dev: (Reads XML Schema spec). Saints preserve us! Alexander the Great couldn't unravel that.

SG: Don't worry about it. Your tools will create the schema for you. Really, its all about the tooling.

Dev: How are the tools gonna do that?

SG: Well, they will reflect on your code (if possible) and autogenerate a compliant schema.

Dev: Reflect on my code? I thought it was all about documents, not serialized objects.

SG: Didn't you hear me? It's all about the tools. Anyway, we can't expect you to write XML Schema and WSDL by hand. Besides, its just plumbing. You don't need to see it.

Dev: Whoa, back up. What was that word? Wizzdle?

SG: Oh, haven't I mentioned WSDL? W-S-D-L. Web Services Description Language. It's how you specify the data types, parameter lists, operation names, transport bindings, and the endpoint URI, so that client developers can access your service. Check it out.

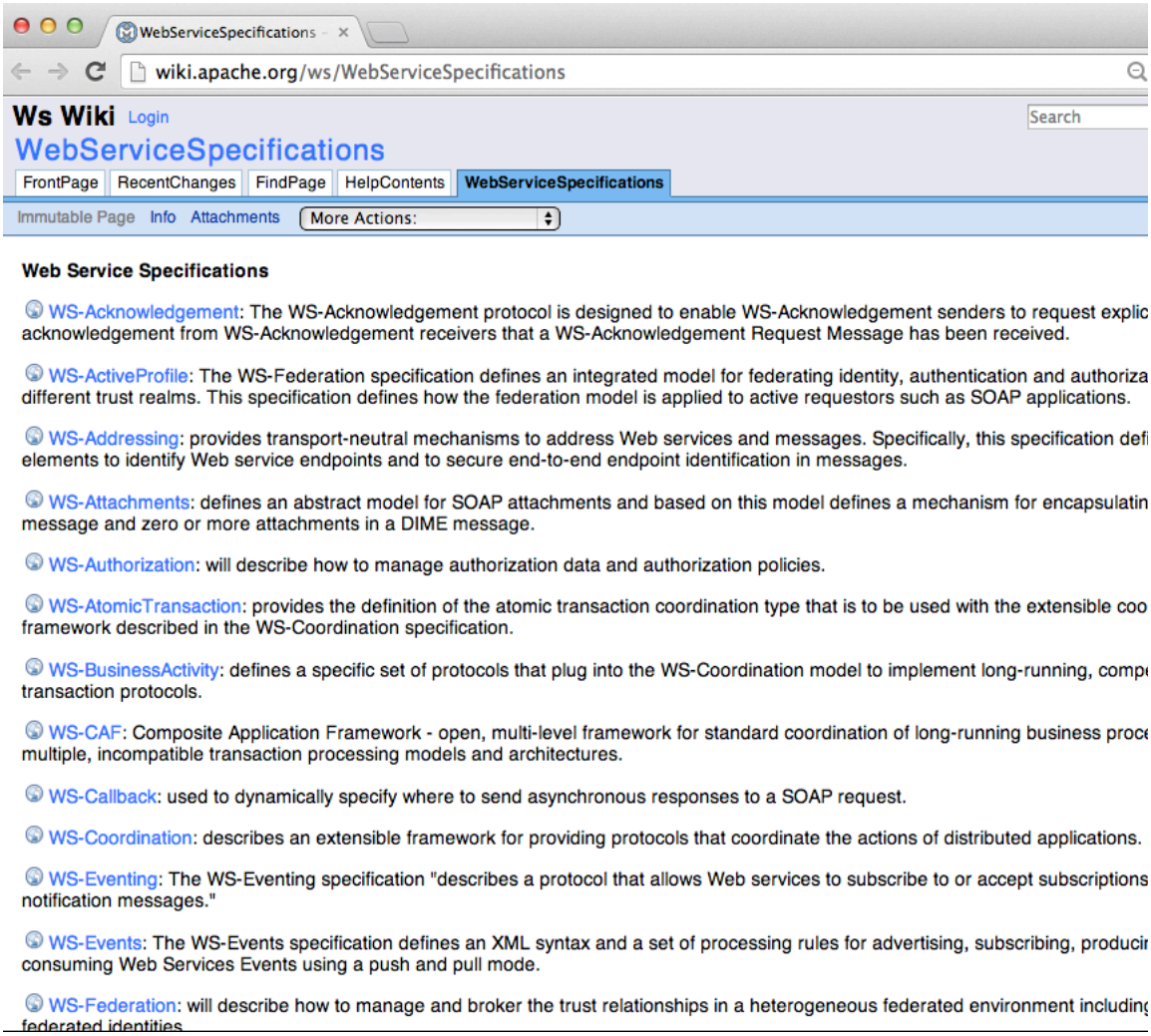
Dev: (Reads WSDL spec). I trust that the guys who wrote this have been shot. It's not even internally consistent.

The culture that grew up around industry standards such as WSDL lead James Lewis and Martin Fowler to complain about "**a complexity that is, frankly, breathtaking**":

> Certainly, many of the techniques in use in the microservice community have grown from the experiences of developers integrating services in large organisations. The Tolerant Reader pattern is an example of this.

> Efforts to use the web have contributed, using simple protocols is another approach derived from these experiences — a reaction away from central standards that have reached a complexity that is, frankly, breathtaking. (Any time you need an ontology to manage your ontologies you know you are in deep trouble.)

Here is an ontology to manage your ontologies (good luck!):



But of course, although umpteen millions were invested in these systems, thousands of conferences and MeetUps were held, dozens of books published, and numerous tech companies got rich as Croesus, when I bring up these expensive disasters, I am told that No True OOP Programmer does any of that nonsense any more. But of course, they do: maintaining these fragile, verbose legacy systems makes up a huge percentage of the work that OOP programmers do.

Where did Alan Kay think OOP should go?

> Smalltalk is not only its syntax or the class library, it is not even about classes. I'm sorry that I long ago coined the term "objects" for this topic because it gets many people to focus on the lesser idea. The big idea is "messaging" … The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be. Think of the internet – to live, it (a) has to allow many different kinds of ideas and realizations that are beyond any single standard and (b) to allow varying degrees of safe interoperability between these ideas. If you focus on just messaging – and realize that a good metasystem can late bind the various 2nd level architectures used in objects – then much of the language-, UI-, and OS based discussions on this thread are really quite moot.

Does anyone really think that OOP is the best way to give us "messaging"? We have, at this time, numerous technologies that help with messaging. An advocate of the functional paradigm might say something like "Pure functions combined with the Actor Model go much further toward giving us what Alan Kay seems to think best" but I'll point out, you could restrict yourself to using PHP, and only use those features that were available in PHP4, back in 2004, and then add in one modern ingredient, for instance, maybe integration with ZeroMQ, and with that you would be able to achieve something closer to Alan Kay's ideal than what most of the modern, bloated OOP frameworks give us. (Indeed, for this reason I am sad that **Photon** seems semi-dead, as the integration of ZeroMQ/Mongrel2 seemed like the beginning of an exciting rebirth for PHP.)

If anyone wants to respond with “but that is a completely different kind of messaging!”, that is exactly my point, that a different kind of messaging would give us something much closer to Alan Kay’s original vision: “The mental image was one of separate computers sending requests to other computers that had to be accepted and understood by the receivers before anything could happen.” Something very basic, such as PHP4 and a good networking library, would allow a kind of messaging that would naturally allow the decoupling and composability that OOP originally promised.

(In passing, let us also note that the tendency of the tech industry to re-use certain words (“message”, “class”, “hierarchy”) in radically different contexts makes it difficult for us to discuss these concepts with clarity. If we simply invented new words for new concepts, as Isaac Newton did when he invented the word “gravity”, we would have a better vocabulary for discussing the radical innovations brought to us by computers.)

When I get into a conversation with a proponent of OOP, I try to point out that the reality does not live up to Alan Kay’s original dream, at which point the original dream is usually dismissed. At this point in the conversation I typically hear some variation of “Modern OOP has evolved into something much more powerful and flexible than its original vision.” Which is fair enough. Time goes by, and software development, as a field, has undergone many changes since the early 1980s. So what are some modern understandings of OOP? Let’s review a few of the sites that try to teach OOP to beginners — what do they view as the core ideas of OOP? **CodeBetter focuses on this list of features**:

> Encapsulation
> …encapsulation is the hiding of data implementation by restricting access to accessors and mutators.
>
> Accessor
> An accessor is a method that is used to ask an object about itself. In OOP, these are usually in the form of properties, which have, under normal conditions, a get method, which is an accessor method.
>
> Mutators are public methods that are used to modify the state of an object, while hiding the implementation of exactly how the data gets modified. Mutators are commonly another portion of the property discussed above, except this time its the set method that lets the caller modify the member data behind the scenes.
>
> Abstraction
> Software developers use abstraction to decompose complex systems into smaller components. As development progresses, programmers know the functionality they can expect from as yet undeveloped subsystems. Thus, programmers are not burdened by considering the ways in which the implementation of later subsystesm will affect the design of earlier development. The best definition of abstraction I’ve ever read is: “An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of object and thus provide crisply defined conceptualboundaries, relative to the perspective of the viewer.”
>
> Inheritance
> Objects can relate to each other with either a “has a”, “uses a” or an “is a” relationship. “Is a”is the inheritance way of object relationship. The example of this that has always stuck with me over the years is a library (I think I may have read it in something Grady Booch wrote). So, take a library, for example. A library lends more than just books, it also lends magazines, audiocassettes and microfilm. On some level, all of these items can be treated the same: All four types represent assets of the library that can be loaned out to people. However, even though the 4 types can be viewed as the same, they are not identical. A book has an ISBN and a magazine does not. And audiocassette has a play length and microfilm cannot be checked out overnight.
>
> Polymorphism
> [This] manifests itself by having multiple methods all with the same name, but slighty different functionality. Many VB6ers are familiar with interface polymorphism. I’m only going to discuss polymorphism from the point of view of inheritance because this is the part that is new to many people. Because of this, it can be difficult to fully grasp the full potential of polymorphism until you get some practice with it and see exactly what happens under different scenarios.

That is fairly basic, and a bit naive. A good OOP programmer would offer some qualifiers distinguishing between best and worst practice. **Wikibooks offers many of those qualifiers and therefore blunts some of my criticisms**:

> Object oriented programming can be traced back to a language called Simula, and in particular Simula 67, which was popular during the 1960s. It was Simula that first instituted “classes” and “objects,” leading to the term “object oriented” programming. By the early 1990s, enough experience had been gained with large OOP projects to discover some limitations. Languages such as Self, ideas like interface programming (also known as component or component-oriented programming), and methodologies such as generic programming were being developed in response to these difficulties. Although often derided by OOP purists, it was the standardization of C++ in 1998 — including generic programming facilities — that really ushered in the modern era of OOP, which we also refer to as Multi-Paradigm programming.

> …[offering an example about "chairs":]
>
> …It is easy to drift off into abstruse philosophical debates over objectness; in some areas like knowledge representation and computational ontology they are very relevant. However, for a computer programmer, it is only necessary to figure out what your application needs to know about and do with chairs.
>
> …Structs (structures), records, tables, and other ways of organizing related information predated object oriented programming. You may be familiar with something like the following Pascal code:
>
> TYPE chair = RECORD
> model : integer;
> weight : integer;
> height : integer;
> color : COLOR;
> END;
>
> This doesn't actually create a chair variable, but defines what a chair variable will look like when you create one. You could proceed to create arrays of chairs and so forth, and as we hope you've discovered for yourself, this kind of thing is quite indispensable for keeping your programs understandable. Object oriented programming wants to push this advantage and milk it for every ounce of understandability, correctness, and simplicity it can.

When they mention "correctness" they are conflating static-typing with OOP. There are OOP languages that do not offer type enforcement (PHP, Ruby, etc). But let's continue listening to them, because static data-types are a common argument in favor of OOP:

> A fundamental problem solving technique is divide and conquer — you just have to figure out how to partition your problem into sub-problems. OOP innovators realized that we already had figured out ways to partition the problem, and it was reflected in the way we organized our data, like above. If you looked at an application that had that chair RECORD in it, you would surely find lots of code for doing things with chairs. Why else would you bother to define it? So, if you were to extract all of that code from all over the application and put it together with the chair definition, the reasoning goes, you should have an easier time ensuring that:
>
> all chair code is correct
>
> all chair code is consistent with each other
>
> there's no duplicated chair code
>
> overall, you have less spaghetti because chair code is no longer tangled up with sofa code etc
>
> So you take that chair definition and that code extracted from all over the application, and call that a class. Take a chair variable and call it an object, and you're off to the races.

Much of the above quote confuses OOP with static data-type checking, but there are many languages that offer type-checking without resorting to OOP. I'll offer some examples of Haskell down below, but here I want to mention the very interesting **Qi/Shen language**:

> Qi makes use of the logical notation of sequent calculus to define types. This type notation, under Qi's interpretation, is actually a Turing complete language in its own right. This notation allows Qi to assign extensible type systems to Common Lisp libraries and is thought of as an extremely powerful feature of the language.

Qi/Shen has a type system that is far more powerful than anything offered by any OOP language, and yet Qi/Shen is not an OOP language. I could say the same thing about Haskell.

I have the impression that Wikibooks is doing the best that it can to defend OOP, though the writers are aware of all the many failures of OOP. And in fact, they can not think of many good things about OOP, so they instead talk about type-checking and consistency, which are available, with much less effort, in non-OOP languages. In this next quote, we can all agree that it would be bad if "Applications could easily set invalid or nonsensical values" but let's remember that protection from invalid or nonsensical values is not unique to OOP:

> Encapsulation is about risk management, reducing your maintenance burden, and limiting your exposure to vulnerabilities — especially those caused by bypassed/forgotten sanity checks or initialization procedures, or various issues that may arise due to the simple fact of the code changing in different ways over time. Technically, encapsulation is hiding internal details behind an opaque barrier so as to force external entities to interact through publicly available access points.
>
> Think about it in the context of an OS kernel, like the Linux kernel. In general, you don't want a common user level application modifying any internal kernel data structures directly — you want applications to work

> through the API (Application Programming Interface). Hence encapsulation is the general term we use for giving varied levels of separation between any core system elements and any common application elements. Otherwise, “unencapsulated code” would be bad for a number of obvious reasons:
>
> 1.) Applications could easily set invalid or nonsensical values, causing the whole system to crash. Forcing the application to use the API ensures that sanity checks get run on all parameters and all data structures maintain a consistent state.
>
> Internal data structures could be updated and change (even drastically so) between seemingly minor kernel updates. Sticking to the API insulates application developers from having to rewrite their code all the time.

We should note the irony that they are using Linux to explain OOP concepts, even though Linux is written in C, which is not an OOP language. They acknowledge that OOP is not necessary for encapsulation when they write “Hence encapsulation is the general term we use for giving varied levels of separation between any core system elements and any common application elements.” They are applying this definition of encapsulation to a program written in C, so clearly, encapsulation is more about writing code intelligently than writing code with an OOP language.

As an aside, since they mention Linux, **let us consider what Linus Torvalds had to say when someone suggested that Linux should be re-written with the OOP language C++**:

> C++ is a horrible language. It’s made more horrible by the fact that a lot of substandard programmers use it, to the point where it’s much much easier to generate total and utter crap with it. Quite frankly, even if the choice of C were to do *nothing* but keep the C++ programmers out, that in itself would be a huge reason to use C.
>
> C++ leads to really really bad design choices. You invariably start using the “nice” library features of the language like STL and Boost and other total and utter crap, that may “help” you program, but causes:
>
> – infinite amounts of pain when they don’t work (and anybody who tells me that STL and especially Boost are stable and portable is just so full of BS that it’s not even funny)
>
> – inefficient abstracted programming models where two years down the road you notice that some abstraction wasn’t very efficient, but now all your code depends on all the nice object models around it, and you cannot fix it without rewriting your app.

If you have time, you can read the conversation **this sparked on Hacker News, and please note how many of the responses fall into the category of “No True Scotsman”**:

> Linus may be a very experienced C programmer, but that doesn’t mean his opinion on C++ carries much weight… I’d be more interested on what someone who actually has a lot of experience in using C++ says. Especially with modern C++ and recent tools, libraries etc, which are very different from what was around five or ten years ago.

A True OOP Programmer only uses the most recent tools, libraries, etc, which are very different from what was around five or ten years ago.

> Linus’s objections seem centered on the fact that it makes it easier to generate bloated code. While this may be true, there’s nothing a little self-discipline can’t control.

A True OOP programmer has the self-discipline to avoid writing bloated code.

The worst accusation I can make about OOP encapsulation is that it fails to give us the protection from unwanted change that it promises us. Consider this classic example:

```
class UniqueId {
private i = 0;
function getUniqueId() {
return i++;
}
}
```

Let us assume that multiple threads are doing some work, perhaps resizing images, and to ensure a unique name for each image, they want a function that will return a unique number that they can use as part of the name. The var i is “encapsulated” because we have declared it to be private — the only way to access it is using the accessor function getUniqueId(). But sadly, it turns out that this is not an atomic operation:

```
i++
```

This is actually 3 operations:

1.) read the value from memory

2.) increment the value

3.) write the new value to memory

What happens if 2 threads call this method at the same instant? Potentially they will both get the same number returned to them — this object can not ensure that its private variable is unique.

To get around this problem, some OOP languages offer different methods of acquiring locks on objects or object fields, and some OOP languages even offer automatic ways of handling this specific problem. For instance, **Java offers AtomicInteger**. But AtomicInteger is a mere convenience, it simply automates, in the background, the fetching, and then the releasing, of a lock. Lot's of non-OOP language also automate the fetching and releasing of locks (Clojure, for instance), so why should we bother with the complexity of OOP?

Whenever I make this point in conversation, someone will tell me that I am confused — locks have nothing to do with encapsulation, I am making a big mistake by running these 2 ideas together. But what is the goal of encapsulation? Encapsulation is a form of "data hiding". Why do we want "data hiding"? We want it because we want protection from unexpected changes to state, and OOP encapsulation rarely gives us this — we have to combine this kind of encapsulation with other tools, such as locks, to achieve the goal. This suggests weakness in the way that OOP enables data hiding.

Here is one of the best passages in the Wikibooks:

> Something that we don't see reiterated enough: State (as opposed to change) is evil! Or, (perhaps better said) maintaining unnecessary state is the root of all (er… many) bugs.

No disagreement from me. This is a point that is often made by those who favor the "functional" programming style. Immutable data is safe, mutable data is unsafe. And the whole world of OOP has been moving towards favoring immutable data whenever possible. In his book "**Effective Java**", Joshua Bloch makes this one of his 52 tips: favor immutable data. But then, this raises the question, why not go all the way with this, and give up OOP, and embrace the "functional" style of programming? Why not go with a language where immutable is the default and mutable is unusual?

Here is another excellent passage from the Wikibooks, about inheritance, but it again raises the question, why bother with OOP at all? We can all agree that we need our data-types to exist in a hierarchy, but we can do this more easily outside of OOP languages, so why add all the burdens of OOP?

> In many books, inheritance and OOP are made to seem synonymous, so it may seem strange that we deferred this discussion so far. This is a reflection of the diminished role of inheritance over time. In fact, one of the primary distinctions between Classic and Modern OOP lies in the usage of inheritance. As the old adage goes, if all you have is a hammer, then everything looks like a nail. And so it happened that often times, inheritance was the only tool available to the erstwhile OOP programmer, and so every concept under the sun was crammed into inheritance. This lack of conceptual integrity and separation of concerns led to over-intimate dependencies and many difficulties. In some languages, programmer technique evolved to make the concepts clearer using the same limited language functions, while other languages explicitly developed features to address these concerns. Because you are almost certain to be exposed to some of this misguided advice at some point in your OOP learning, we'll try and explain some of the problems to you.
>
> …The most over-used and rather worthless discussion on inheritance that you will see revolves around the "Is-A vs Has-A" discussion. For example, a car is-a vehicle but has-a steering wheel. The idea these authors are chasing is that your car class should inherit your vehicle class and have a steering wheel as a member. You might also run across examples with shapes, where a Rectangle is-a Shape. The point is, once again, abstraction. The goal is to identify operations that operate only on the abstract notion of vehicle or shape and then write that code only once. Then, through the magic of inheritance, you can pass in cars or rectangles or what-have-you to the generic code, and it will work, since the derived classes are everything the parent classes are, "plus more".
>
> The problem here is that inheritance is mixing together several things: you inherit "typeness", interface, and implementation all at the same

> time. However, all of the examples focus on interface while talking about “typeness”. The abstract code doesn’t care that a car “is-a” vehicle, just that the objects respond to a certain set of functions, or interface. In fact, if you want to give your chair class accelerate(), brake(), turn_left() and turn_right() methods, shouldn’t the abstract code be able to work on chairs then? Well, of course, but that doesn’t make a chair a vehicle.

I want to offer some examples from a “functional style” language, and Clojure is the functional language I know best, so I will use that. **In Clojure, inheritance is simple**:

```
(derive ::rect ::shape)
nil

(derive ::circle ::shape)
nil

(isa? ::circle ::shape)
true

(isa? ::rect ::shape)
true
```

Here, I get to define my data-type hierarchy independently of my functions and independently of any state. We do not need OOP to have inheritance. Someone might look at this example and complain that this, by itself, does not give us compile-time data-type checking, to which 4 responses are necessary:

1.) data-type hierarchies and compile-time checking are 2 different subjects which should be addressed separately. Some OOP languages have compile-time checking and some don’t, and some non-OOP languages have compile-time checking, and some don’t.

2.) Clojure now has a project (**core.typed**) that allows you to annotate your code with type checks, and the compiler issues warnings on those checks.

3.) even if you don’t use core.typed, you can use pre and post assertions on your functions which will give you run-time data-type checking

4.) don’t let the specifics of any one example distract you from the overall idea: we can have a data-type hierarchy that is independent of OOP. Please try to notice the gestalt of this essay, a gestalt that can not be communicated in any one example.

Before I sum up the supposed strengths of OOP, I will link to 2 more articles that describe the so-called strengths of OOP. **Here is one from Drexel University**:

> Code Reuse and Recycling: Objects created for Object Oriented Programs can easily be reused in other programs.
>
> Encapsulation (part 1): Once an Object is created, knowledge of its implementation is not necessary for its use. In older programs, coders needed to understand the details of a piece of code before using it (in this or another program).
>
> Encapsulation (part 2): Objects have the ability to hide certain parts of themselves from programmers. This prevents programmers from tampering with values they shouldn’t. Additionally, the object controls how one interacts with it, preventing other kinds of errors. For example, a programmer (or another program) cannot set the width of a window to -400.
>
> Design Benefits: Large programs are very difficult to write. Object Oriented Programs force designers to go through an extensive planning phase, which makes for better designs with less flaws. In addition, once a program reaches a certain size, Object Oriented Programs are actually easier to program than non-Object Oriented ones.
>
> Software Maintenance: Programs are not disposable. Legacy code must be dealt with on a daily basis, either to be improved upon (for a new version of an existing piece of software) or made to work with newer computers and software. An Object Oriented Program is much easier to modify and maintain than a non-Object Oriented Program. So although a lot of work is spent before the program is written, less work is needed to maintain it over time.

Also, there are the SOLID principles, which describe good architectural ideas for software, and which assume that OOP programming is the best way to implement them:

> Single responsibility principle
> a class should have only a single responsibility (i.e. only one potential change in the software’s specification should be able to affect the specification of the class)
>
> Open/closed principle
> “software entities … should be open for extension, but closed for modification.”

> Liskov substitution principle
> "objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program." See also design by contract.
>
> Interface segregation principle
> "many client-specific interfaces are better than one general-purpose interface."[8]
>
> Dependency inversion principle
> one should "Depend upon Abstractions. Do not depend upon concretions."[8]
> Dependency injection is one method of following this principle.

So, to recap, these are 12 of the things that proponents of OOP regard as the strengths of OOP:

Encapsulation

Polymorphism

Inheritance

Abstraction

Code re-use

Design Benefits

Software Maintenance

Single responsibility principle

Open/closed principle

Interface segregation principle

Dependency inversion principle

Static type checking

Some people will read this and decide that their favorite thing about OOP is not on this list. Some people will insist that the best thing about OOP is "design by contract" or testability or dependency management or some other thing that they love. This is one of the difficult things about arguing over OOP: there is a great diversity of definitions. All the same, when I go to a job interview, the things I get asked about the most often are encapsulation and polymorphic dispatch and inheritance, so simply by having those 3 on the list, I think I am covering the core ideas that people associate with OOP.

The critics of OOP have many concerns that are not on this list, for instance, **object-relational impedance mismatch**, and the inability of OOP to handle concurrency. And yet, even if we ignore what the critics say, we can still make clear that OOP is a terrible paradigm for writing software. (All the same, I'll mention concurrency issues below.)

Encapsulation

I've already talked about encapsulation above, and I don't have much more to say about it. OOP gives us a type of data encapsulation (that is, a type of data hiding), but it fails at the goal of protecting its "private" data from unwanted changes, as multiple threads can all read the same variable at the same instant, leading to the kinds of unpredictable changes that encapsulation is suppose to protect us from. In most OOP languages (Java, C++, etc) we only get real protection from unwanted change by acquiring a lock (that is, we only achieve the goal we are after with a combination of data hiding and locks) or some other tool of synchronization, but non-OOP languages also give us the same thing.

There are some OOP languages that are typically single threaded:

MRI Ruby

PHP

Javascript

Does OOP offer encapsulation in a single-threaded OOP language? Mostly, though in Ruby there are other ways of getting around the limits that are supposed to be imposed by the "private" or "protected" keywords. And Javascript gives us data hiding the same way any Lisp would: via closures. (**Douglas Crockford was the first to show how to implement private members in Javascript by using closures.**)

Whether you use an OOP language or a functional language, your enemy is state, and the goal of any kind of data-hiding is to limit the ways that state can change. **John Barker sums up the enemy**:

> State is not your friend, state is your enemy. Changes to state make programs harder to reason about, harder to test and harder to debug. Stateful programs are harder to parallelize, and this is important in a

> world moving towards more units, more cores and more work. OOP languages encourage mutability, non determinism and complexity.
>
> As someone who was initially hostile to the idea that state is the root of all problems, I initially greeted this idea with skepticism. Mutating state is so easy and fundamental in OOP that you often overlook how often it happens. If you're invoking a method on an object that's not a getter, you're probably mutating state.

In fact, OOP gives us a vast graph of mutable objects, all of which can mutate each other, with a change in any one object possibly setting off a cascade of mutations that propagate out through the graph in ways that are often too complicated for the human mind to comprehend. **Rich Hickey (inventor of Clojure) has made this point more clearly than anyone**:

> Even if you don't have concurrency, I think that large objected-oriented programs struggle with increasing complexity as you build this large object graph of mutable objects. You know, trying to understand and keep in your mind what will happen when you call a method and what will the side-effects be.

It's also worth noting that sometimes with OOP we are forced to engage in data-hiding more than we would like, since having generic data structures, and a wealth of functions to operate on them, offers some convenience. **Colin Jones sums this up nicely**:

> More generic data types generally have well-known and useful functions associated with them. We could enumerate the keys or values on a map, filter an array by a function, or reduce across it. Alan Perlis [allegedly] said "It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures," and I tend to agree. One benefit of using a more generic data structure instead of hiding that data behind a class is that it removes an extra step in applying well-known functions to that data. In some situations this could be considered a downside. Consider encapsulation, where we might insulate against changes to the underlying data structures. But in many cases when choosing to encapsulate our data behind a class, the tradeoffs in ease of use and reuse are not worth it. If we decide we need encapsulation, of course, most functional languages do provide that ability using closures.

Polymorphism

Polymorphism in OOP is weak, and yet, paradoxically, polymorphism is often mentioned as a strength of OOP. **Here, truly, we run into the mental weaknesses of the Blub programmer**:

> Programmers get very attached to their favorite languages, and I don't want to hurt anyone's feelings, so to explain this point I'm going to use a hypothetical language called Blub. Blub falls right in the middle of the abstractness continuum. It is not the most powerful language, but it is more powerful than Cobol or machine language.
>
> And in fact, our hypothetical Blub programmer wouldn't use either of them. Of course he wouldn't program in machine language. That's what compilers are for. And as for Cobol, he doesn't know how anyone can get anything done with it. It doesn't even have x (Blub feature of your choice).
>
> As long as our hypothetical Blub programmer is looking down the power continuum, he knows he's looking down. Languages less powerful than Blub are obviously less powerful, because they're missing some feature he's used to. But when our hypothetical Blub programmer looks in the other direction, up the power continuum, he doesn't realize he's looking up. What he sees are merely weird languages. He probably considers them about equivalent in power to Blub, but with all this other hairy stuff thrown in as well. Blub is good enough for him, because he thinks in Blub.

Yukihiro Matsumoto (inventor of Ruby) put the same idea in a more polite form:

> Every language or system has its own culture. In the background of every language or system are some central ideas. Most of these ideas are good, but they are different. By learning many languages and systems, you get exposed to different ideas — and that enhances your point of view. If you don't know Prolog, for example, you may not know the power of goal directed programming — programming by describing the problem to solve through specifying rules to apply. This is a very interesting concept. It is a very different way of thinking. And if you don't know Prolog, or the predicate logic, it's very difficult to discover this way of thinking by yourself. Knowing other systems and paradigms expands the world inside your brain. That's why I advise learning multiple languages.

Why do we want polymorphism? Because we want flexibility in the way we dispatch execution. But OOP languages generally only give us flexible dispatch based on the signature of a method, and the signature is almost always evaluated in terms of the data-type of the parameters being handed into the method. There are more expansive ways to think of flexible dispatch. For instance, **multimethods in Lisp languages give us a more powerful method of dispatch**:

> A Clojure multimethods is a combination of a dispatch function and one or more methods, each defining its own dispatch value. The dispatching function is called first, and returns a dispatch value. This value is then matched to the correct method. Lets take a look at our previous example refactored into a multimethod.
>
> (defmulti convert class)
>
> (defmethod convert clojure.lang.Keyword [data]
> (convert (name data)))
>
> (defmethod convert java.lang.String [data]
> (str "\"" data "\""))
>
> (defmethod convert nil [data]
> "null")
>
> (defmethod convert :default [data]
> (str data))
>
> Awesome! We have our first polymorphic solution. Now we can add more data types without altering the existing functions. Let's add a method for vectors as well.
>
> (defmethod convert clojure.lang.PersistentVector [data]
> (str "[" (join ", " (map convert data)) "]"))
>
> Now we can also convert vectors into JSON.

In the above example, they are using "class" as the dispatch function (which returns the class of the parameter), so it works just like traditional OOP, and it even falls back on the underlying Java OOP. But we can use any function for dispatch in a multimethod.

Bozhidar Batsov offers an example using an anonymous function:

> Object oriented programming in most programming languages is based on a single dispatch message passing. The object on which we invoke a method (poor choice of words, but easier to comprehend) is the receiver, the method name and it's arguments are the message. The method's invoked solely on the base of the type of the receiver object.
>
> Lisps have traditionally implemented OOP with generic methods, that don't have a receiver and are dispatched on the basis of the types of all of their arguments. In the world of multiple dispatch the more traditional single dispatch is just a special case in which only the type of the first method argument matters. Here's a taste of multimethods in Clojure:
>
> (defmulti my-add (fn [x y] (and (string? x) (string? y))))
>
> (defmethod my-add true [x y]
> (str x y))
>
> (defmethod my-add false [x y]
> (+ x y))
>
> user=> (my-add 3 4) ; => 7
>
> user=> (my-add "3″ "4″) ; => "34″
>
> Here we defined a multi-method that behaves differently for string and numeric arguments — strings args are concatenated and numeric args are added together.

We all want the power that real polymorphism gives us, but OOP fails to deliver.

Inheritance

Inheritance is crucial to the software that I write. Inheritance is one of the most important ideas in software development. We all the want the power to create hierarchies of data types, and up above I quoted an example of how simple this can be in a language like Clojure. There is no limit to inheritance in a functional language such as Clojure, nor do complex hierarchies inflict any pain on one's program. **But in OOP programming, inheritance is dangerous**:

> Although introductory texts on Object Oriented Programming (OOP) are quick to tout the benefits of inheritance they typically fail to teach the lessons learned outside of academia detailing the shortcomings of this aspect of OOP. Disadvantages of using object inheritance include the following
>
> Large Inheritance Hierarchy: Overuse of inheritance can lead to inheritance hierarchies that are several levels deep. Such large inheritance hierarchies typically become difficult to manage and maintain due to the fact that the derived class is vulnerable to changes made in any of the derived classes which often leads to fragility. There are also performance considerations in that instantiating such classes involves calling constructors across the entire inheritance hierarchy as well as above average memory requirements for such objects. An

> example of such a class is the javax.swing.JFrame class in the Java swing library which has an inheritance depth of six levels.
>
> Fragile Superclasses: Classes that have been subclassed cannot be altered at will in subsequent versions because this may negatively impact derived classes. In C++ this is especially problematic because changes in a superclass typically end up involving a recompile of the child classes. Java utilization of dynamic resolution prevents the need for recompilation but does not entirely lessen the need to avoid making significant changes in base classes.
>
> Breaks Encapsulation: Inheritance in OOP is primarily a mechanism for reusing source code as opposed to a mechanism for reusing binary objects. This transparent nature of OOP inheritance relies on the author of the derived class being the author of the base class or having access to its implementation details. This violates one of the other tennets of Object Oriented Programming; encapsulation.

Joshua Bloch, in his book “Effective Java” says, “Favor composition over inheritance”. The problems with inheritance, in OOP, have become clearer over time. **It is tragic that OOP has had to retreat from such a powerful idea, to the extent that composition is now the favored strategy**:

> Composition over inheritance (or Composite Reuse Principle) in object-oriented programming is a technique by which classes may achieve polymorphic behavior and code reuse by containing other classes that implement the desired functionality instead of through inheritance. Some languages, notably Go, use type composition exclusively.

It is absolutely tragic that this advice needs to be given. Back in the 1960s, OOP introduced the concept of inheritance, and yet now we know OOP is a terrible way to implement inheritance. And yet, we often want inheritance, not composition.

Inheritance is good. Nearly all data-types belong to a hierarchy, and my code should be able to represent that hierarchy without fear. If I write software for a store, I will probably have a “sell price” and a “buy price”, both of which are children of “price” which is a child of “transaction metric” which is a child of “decimal” (or “number”). I don’t want to model this with composition, nor do I want to worry about brittle base classes and tight-coupling between the interface and the implementation. To escape from these worries, I want my data-types declared in a hierarchy that is all-together separate from my code’s behavior (the functions that I write). Functional languages like such as Shen, Haskell or Clojure allow for data-type definitions that are separate from the behavior of my code. Java does not. Ruby does not. Python does not.

Consider the case where “SimpleProductManager” is a child of “ProductManager”:

```
public class SimpleProductManager implements ProductManager {
private List products;

public List getProducts() {
return products;
}

public void increasePrice(int percentage) {
if (products != null) {
for (Product product : products) {
double newPrice = product.getPrice().doubleValue() *
(100 + percentage)/100;
product.setPrice(newPrice);
}
}
}

public void setProducts(List products) {
this.products = products;
}
}
```

There are 3 behaviors here:

getProducts()

increasePrice()

setProducts()

Is there any rational reason why these 3 behaviors should be linked to the fact that in my data hierarchy I want “SimpleProductManager” to be a child of “ProductManager”? I can not think of any. I do not want the behavior of my code linked together with my definition of my data-type hierarchy, and yet in OOP I have no choice: all methods must go inside of a class, and the class declaration is also where I declare my data-type hierarchy:

```
public class SimpleProductManager implements ProductManager
```

This is a disaster.

At this point in the conversation, someone might be tempted to say something like “You can write bad code in any language, and you can write good code in any language.” That is true, but there is the question, does the language help us write

good code? Does the language help with code re-use? **You can adopt the functional style while still writing Java code**, but is this easy or fun?

Abstraction

Let us reconsider the quote above:

"Software developers use abstraction to decompose complex systems into smaller components."

This is absolutely true, and this has nothing to do with OOP.

This part is specific to OOP:

"An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of object and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer."

This definition of "abstraction" leads to the advice of "program to an interface, not to an implementation". **Let us consider Chad Myers reflections on this advice, as he gleaned it from the book "Design Patterns: Elements of Reusable Object-Oriented Software"**:

> In the book, the author says:
>
> *[Manipulating objects solely in terms of their interface and not their implementation] so greatly reduces implementation dependencies between subsystems that it leads to the following principle of reusable object-oriented design:*
>
> *Program to an interface, not an implementation.*
>
> *Don't declare variables to be instances of particular concrete classes. Instead, commit only to an interface defined by an abstract class.*
>
> This point is profound and if it isn't already something you religiously practice, I suggest you do some more research on this topic. Coupling between types directly is the hardest, most pernicious form of coupling you can have and thus will cause considerable pain later.
>
> Consider this code example:
>
> ```
> public string GetLastUsername()
> {
> return new UserReportingService().GetLastUser().Name;
> }
> ```
>
> As you can see, our class is directly new()'ing up a UserReportingService. If UserReportingService changes, even slightly, so must our class. Changes become more difficult now and have wider-sweeping ramifications. We have now just made our design more brittle and therefore, costly to change. Our future selves will regret this decision. Put plainly, the "new" keyword (when used against non-framework/core-library types) is potentially one of the most dangerous and costly keywords in the entire language — almost as bad as "goto" (or "on error resume next" for the VB/VBScript veterans out there).
>
> What, then, can a good developer do to avoid this? Extract an interface from UserReportingService (-> IUserReportingService) and couple to that. But we still have the problem that if my class can't reference UserReportingService directly, where will the reference to IUserReportingService come from? Who will create it? And once its created, how will my object receive it? This last question is the basis for the Dependency Inversion principle. Typically, dependencies are injected through your class' constructor or via setter methods (or properties in C#).

So, we want to GetLastUsername(). Before we introduce the complexity of OOP, lets go back to basics for a minute and think about the easiest way to get a last name, and then we will think about why the simplest way of doing this does not work for us.

We want to get a user's last name. If we keep all data about a user in a hashmap, then we might have a key called "last_name" and we can call that key and get the last name. That is very basic and simple. What is wrong with this approach?

Well, the simplest approach doesn't (always) work, because we probably have a contract that we want to enforce where the User is concerned. So let's think about contracts, and compare contracts in OOP and non-OOP languages.

I think we can all agree that contracts are important, and we want our contracts enforced. "Design by contract" is an excellent methodology for developing software. I am sure we have all written software that had the concept of a User. We want a contract for User, which perhaps enforces these rules:

"first name" must be a string

"last name" must be a string

"date of birth" must be between 1890 and 2014

Up above there was an example of Pascal code that offered this struct:

```
TYPE chair = RECORD
model : integer;
weight : integer;
height : integer;
color : COLOR;
END;
```

Most programming languages that I am aware of offer something like a struct or a record or an enum or a class — we can use these to enforce contracts. However, whatever data structure we use, do we need the complexity that is apparent in Chad Myers example? A simple record can give us contract-enforcement, but a class comes with a whole extra set of baggage that programmers would be wise to avoid. Please ask yourself what is really needed here. Do we need a Service object to fetch the User for us? Do we need Dependency Inversion to set our current object with a Service object so we can fetch a last name? Do we need OOP?

At most, we need 2 things for our User:

1.) contract enforcement

2.) data hiding

We do not need OOP for this, and if we do use OOP then it inflicts on us a very high cost in terms of ceremony, setup, and complexity. How bad is that cost? Myers continues:

> It's also the case that the act of creating (new()'ing) an object is actually a responsibility in and of itself. A responsibility that your object, which is focused on getting the username of the last user who accessed the system, should not be doing. This concept is known as the Single Responsibility Principle. It's a subtle distinction, and we usually don't think of the "new" keyword as a responsibility until you consider the ramifications of that simple keyword. What if UserReportingService itself has dependencies that need to be satisfied? Your class would have to satisfy them. What if there are special conditions that need to be met in order for UserReportingService to be instantiated properly (existing connection to the database/open transaction, access to the file system, etc). The direct use of UserReportingService could substantially impact the functioning of your class and therefore must be carefully used and designed. To restate, in order to use another class like UserReportingService, your class must be fully responsible and aware of the impacts of using that class.

Please note the irony that Myers is arguing in favor of OOP, though his words can very easily be read as a criticism of OOP. He continues:

> The Creational patterns are concerned with removing that responsibility and concern from your class and moving it to another class or system that is designed for and prepared to handle the complex dependencies and requirements of the classes in your system. This notion is very good and has served us well over the last 15 years. However, the Abstract Factory and Builder pattern implementations, to name two, became increasingly complicated and convoluted. Many started reaching the conclusion that, in a well-designed and interface-based object architecture, dealing with the creation and dependency chain management of all these types/classes/objects (for there will be many more in an interface-based architecture and that is OK), a tool was needed. People experimented with generating code for their factories and such, but that turned out not to be flexible enough.

In other words, the costs of this approach were so awful that even the proponents of OOP nowadays shrink away in horror. OOP was once seen as the silver bullet that was going to save the software industry. Nowadays we need a silver bullet to save us from OOP, and so, we are told (in the next paragraph), the Inversion of Control Container was invented. This is the new silver bullet that will save the old silver bullet.

The next 4 paragraphs are simply amazing, and not in a good way:

> To combat the increasing complexity and burden of managing factories, builders, etc, the Inversion of Control Container was invented. It is, in a sense, an intelligent and flexible amalgamation of all the creational patterns. It is an Abstract Factory, a Builder, has Factory Methods, can manage Singleton instances, and provides Prototype capabilities. It turns out that even in small systems, you need all of these patterns in some measure or another. As people turned more and more of their designs over to interface-dependencies, dependency inversion and injection, and inversion of control, they rediscovered a new power that was there all along, but not as easy to pull off: composition.
>
> By centralizing and managing your dependency graph as a first class part of your system, you can more easily implement all the other patterns such as Chain of Responsibility, Decorator, etc. In fact, you could implement many of these patterns with little to no code. Objects that had inverted their control over their dependencies could now benefit from that dependency graph being managed and composited via an external entity: the IoC container.

> As the use of IoC Containers (also just known as 'containers') grew wider and deeper, new patterns of use emerged and a new world of flexibility in architecture and design was opened up. Composition which, before containers, was reserved for special occasions could now be used more often and to fuller effect. Indeed, in some circumstances, the container could implement the pattern for you!
>
> Why is this important? Because composition is important. Composition is preferable to inheritance and should be your first route of reuse, NOT inheritance. I repeat, NOT inheritance. Many, certainly in the .NET space, will go straight for inheritance. This eventually leads to a dark place of many template methods (abstract/virtual methods on the base class) and large hierarchies of base classes (only made worse in a language that allows for multiple inheritance).

Remember, we really only need, at most, 2 things for our User struct/record:

1.) contract enforcement

2.) data hiding (maybe)

We also very badly want one other thing:

3.) hierarchies of data-types

Proponents of OOP, such as Chad Myers, to get #1 and #2 are willing to sacrifice #3, even though #3 is very important, and even after that sacrifice, achieving #1 and #2 involves a mind-numbing degree of complexity. I am astounded that intelligent people actually defend these practices.

Please note that in the above paragraphs "dependencies" has a meaning that is specific to OOP. We are not talking about merely including libraries that your code will call — any good package manager will take care of that for you, and in 2014 we are blessed with a great abundance of good package managers. But Chad Myers is talking about "dependencies" in the sense of providing objects to an object at the time of instantiation — a problem that is inflicted on us by OOP. This kind of "dependencies" is not forced on us by computers, or programming, or logic, or whatever problem we are trying to solve, it is a problem that only exists in the world of OOP.

When I get into a discussion of this issue with proponents of OOP, I am often misunderstood as wanting a compromise between inheritance and composition, or arguing for multiple inheritance, at which point someone will tell me that mixins are the answer. But that misses the point: I want to have inheritance among my data-types, because data-types tend to have natural hierarchies that I want to model in my code, but I don't want my data-types to be bound to specific behavior — I want something more flexible than that. But, even though the point is irrelevant to my argument, **I will point out that, as Michele Simionato says, mixins come with their own set of problems, including namespace pollution**:

> Have a look at the hierarchy of the Plone Site class. Between square backets you can see the number of methods/attributes defined per class, except special attributes. The plot comes from a real Plone application I have in production. The total count is of 38 classes, 88 names overridden, 42 special names and 648 regular names: a monster.
>
> To trace the origin of the methods and to keep in mind the hierarchy is practically impossible. Moreover, both autocompletion and the builtin help facility become unusable, and the self-generated class documentation becomes unreadable since it is too big.
>
> …My hate for mixins comes from my experience with Zope/Plone. However the same abuses could be equally be done in other languages and object systems – with the notable exception of CLOS, where methods are defined outside of classes and therefore the problem of class namespace pollution does not exist – in the presence of huge frameworks.
>
> A consequence of namespace pollution is that it is very easy to have name clashes. Since there are hundreds of methods and it is impossible to know all of them, and since method overriding is silent, this is a real problem: the very first time I subclassed a Plone class I ran into this issue: I overrode a pre-defined method inadvertently, causing hard-to-investigate problems in an unrelated part of the code.

I would emphasize this:

"with the notable exception of CLOS, where methods are defined outside of classes and therefore the problem of class namespace pollution does not exist"

This is exactly what we want: to define our data-type hierarchy "outside of classes", which is to say, independent of any functions. CLOS is the Common Lisp Object System and, as with Clojure (another Lisp) and Shen (another Lisp) and Haskell, the data-type definitions are independent of any behavior. This makes them more flexible, and this facilitates code re-use. **Joe Armstrong (the inventor of Erlang) says this beautifully:**

> Objects bind functions and data structures together in indivisible units. I think this is a fundamental error since functions and data structures belong in totally different worlds. Why is this?

> *Functions do things. They have inputs and outputs. The inputs and outputs are data structures, which get changed by the functions. In most languages functions are built from sequences of imperatives: “Do this and then that …” to understand functions you have to understand the order in which things get done (In lazy functional programming languages (FPLs) and logical languages this restriction is relaxed).
>
> Data structures just are. They don’t do anything. They are intrinsically declarative. “Understanding” a data structure is a lot easier than “understanding” a function.
>
> Functions are understood as black boxes that transform inputs to outputs. If I understand the input and the output then I have understood the function. This does not mean to say that I could have written the function.
>
> Functions are usually “understood” by observing that they are the things in a computational system whose job is to transfer data structure of type T1 into data structure of type T2.
>
> Since functions and data structures are completely different types of animal it is fundamentally incorrect to lock them up in the same cage.

Code Re-use

Chad Myers has some quotes from the book “Design Patterns: Elements of Reusable Object-Oriented Software” which are relevant here:

> [Inheritance] can cause problems when you’re trying to reuse a subclass. Should any aspect of the inherited implementation not be appropriate for new problem domains, the parent class must be rewritten or replaced by something more appropriate. This dependency limits flexibility and ultimately reusability.
>
> …Ideally you shouldn’t have to create new components to achieve reuse. You should be able to get all the functionality you need just by assembling existing components through object composition (via the container — Chad). But this is rarely the case, because the set of available components is never quite rich enough in practice. Reuse by inheritance makes it easier to make new components that can be composed with old ones. Inheritance and object composition thus work together.
>
> Nevertheless, our experience is that designers overuse inheritance as a reuse technique and designs are often made more reusable (and simpler) by depending more on object composition. You’ll see object composition applied again and again in the design patterns.

Again, OOP actively undermines the very thing it is suppose to promote: code re-use.

There are some OOP languages that are famous for enabling a high level of code re-use, and, among these, Ruby stands out as exceptional. But it is worth noting, the things that facilitate code re-use in Ruby are exactly those ideas that were drawn from non-OOP languages. OOP does not help with code re-use in Ruby. **Yukihiro Matsumoto, who invented Ruby, credits Lisp with inspiring those parts of Ruby which help with code re-use**, including the meta-programming and features such as “closures”:

> Bill Venners: What makes a block a closure?
>
> Yukihiro Matsumoto: A closure object has code to run, the executable, and state around the code, the scope. So you capture the environment, namely the local variables, in the closure. As a result, you can refer to the local variables inside a closure. Even after the function has returned, and its local scope has been destroyed, the local variables remain in existence as part of the closure object. When no one refers to the closure anymore, it’s garbage collected, and the local variables go away.
>
> …Bill Venners: OK, but what is the benefit of having the context? The distinction that makes Ruby’s closure a real closure is that it captures the context, the local variables and so on. What benefit do I get from having the context in addition to the code that I don’t get by just being able to pass a chunk of code around as an object?
>
> Yukihiro Matsumoto: Actually, to tell the truth, the first reason is to respect the history of Lisp. Lisp provided real closures, and I wanted to follow that.
>
> Bill Venners: One difference I can see is that data is actually shared between the closure objects and the method. I imagine I could always pass any needed context data into a regular, non-closure, block as parameters, but then the block would just have a copy of the context, not the real thing. It’s not sharing the context. Sharing is what’s going on in a closure that’s different from a plain old function object.
>
> Yukihiro Matsumoto: Yes, and that sharing allows you to do some interesting code demos, but I think it’s not that useful in the daily lives of programmers. It doesn’t matter that much. The plain copy, like it’s done in Java’s inner classes for example, works in most cases. But in Ruby closures, I wanted to respect the Lisp culture.

(An aside: some people argue “Ruby proves that it is possible to have a low-ceremony OOP language.” Nowadays most languages have drawn inspiration from multiple sources, and many languages are multi-paradigm, so when I attack an OOP language, I am attacking those parts of the language that draw ideas from the OOP tradition. **As Paul Graham has pointed out, over the years, nearly all languages have borrowed more and more features from Lisp**. My point is, the features that allow Ruby to be low-ceremony are drawn from non-OOP languages.)

Ruby also allows code re-use through other means, such as Gems, and the Gems are able to plug into Ruby frameworks in part due to Ruby’s meta-programming features. Ruby allows a very high level of meta-programming, almost on par with the Lisps, but this comes at a cost: Ruby’s meta-programming features make it difficult to enforce type constraints or visibility guarantees. **For instance, a method might be marked as “private” and yet you can still access it, thanks to meta-programming**. In other words, to get all of the advantages of Ruby, you have to undermine its OOP features.

I admire Ruby very much: I think it is a beautiful language. It’s meta-programming makes it easy to compose functionality, thus allowing high levels of code re-use. Ruby is very slow, and monkeypatching can introduce bugs that are hard to trace, but, as **Douglas Crockford said of Javascript**, this is a Lisp with C syntax. But that raises the question, why don’t we just use Lisp? Apparently some people prefer C syntax, and feel that a Lisp with C syntax is something worth fighting for. Eric Kidd’s blog post “**Why Ruby is an acceptable LISP**” set off an intense debate, with dozens of people arguing that the world is a better place having Lisp features with a C syntax. What I find interesting is that you can see a debate in the comments, with dozens of intelligent people making hundreds of insightful remarks, and not one of the strengths mentioned go back to the OOP nature of Ruby. One of the strongest arguments made for Ruby was by “Eleanor”:

> So to summarize: Ruby is a powerful language with a syntax that is easy even for non-programmers to get to grips with. It includes concepts to do with block closures, object-orientation and meta-programming which are usually absent from languages accessible to novice programmers and provides idioms for their usage which are easy to remember and use. There are without a doubt architectural techniques which can be utilized more easily in Lisp, but as to whether or not those are techniques which are actively useful for general software development is something that I think is open to debate.

I agree with this: Ruby is an easy way to learn most of the ideas that grew out of Lisp. But that implies that at some point, programmers hit a level of maturity where they should graduate from Ruby and move on to Lisp.

I also very much like this comment by “KristofU”:

> I’m a C++ programmer, and was looking for a new language to learn and do stuff more elegantly. Ruby just sucked me right in. There is almost nothing to learn, it’s just there. You can slice and dice and juggle and the result is always a working program. You like for-statements? Well, you can use for-statements. You like iterators? Well, you can use iterators. Make object functors or lambda’s, whichever you prefer. Ruby doesn’t force anything upon you, there is time to learn to appreciate the finer features, while still churning out working apps in the meantime. I’ve also briefly tried Haskell, Scheme and Erlang, but I have to say, I couldn’t get anything done. Compared to Ruby, quite an anti-climax.

Ruby is dynamic, low-ceremony and does not need to be compiled, so you can skip past the complexity of the data-type checking that is mandatory in Haskell, and you (mostly) don’t have to worry about organizing your code for “supervisory trees” like you do in Erlang. Of course, Ruby is incapable of doing most of the stuff that you can do with Haskell or Erlang. Below I’ll quote from Joe Armstrong’s work on Erlang, where he points out that **Ericsson builds telephone switches (with Erlang) that are expected to have only 2 hours of downtime per 40 years**!!! You could never build anything that reliable with Ruby — what makes it easy also limits its reliability. All the same, I think Ruby has great strengths. But those strengths have nothing to do with OOP.

And having said all that, it has to be noted that Ruby has more ceremony than what it should have, especially considering its reputation. I’ll say more about that later, but also, Stuart Halloway mentions Ruby while he makes the point that **code re-use, in any language, is killed off by the amount of ceremony that is necessary to get anything done**:

> Good code is the opposite of legacy code: it captures and communicates essence, while omitting ceremony (irrelevant detail). Capturing and communicating essence is hard; most of the code I have ever read fails to do this at even a basic level. But some code does a pretty good job with essence. Surprisingly, this decent code still is not very reusable. It can be reused, but only in painfully narrow contexts, and certainly not across a platform switch.
>
> The reason for this is ceremony: code that is unrelated to the task at hand. This code is immediate deadweight, and often vastly outweighs the code that is actually getting work done. Many forms of ceremony come from unnecessary special cases or limitations at the language level, e.g.
>
> factory patterns (Java)

> dependency injection (Java)
>
> getters and setters (Java)
>
> annotations (Java)
>
> verbose exception handling (Java)
>
> special syntax for class variables (Ruby)
>
> special syntax for instance variables (Ruby)
>
> special syntax for the first block argument (Ruby)
>
> High-ceremony code damns you twice: it is harder to maintain, and it needs more maintenance. When you are writing high-ceremony code, you are forced to commit to implementation approaches too early in the process, e.g. “Should I call new, go through a factory, or use dependency injection here?” Since you committed early, you are likely to be wrong and have to change your approach later. This is harder than it needs to be, since your code is bloated, and on it goes.

What we want, at all times, is the least amount of ceremony that still allows us to get our job done. Getting our job done tends to mostly involve data transformations (some would prefer to say “managing state safely”) and enforcing data-type contracts. Depending on your programming goals (reliability? speed of development?) there are functional programming languages that deliver what we want much better than OOP. I’ll offer some insights from the world of Clojure, as I know it better than any other functional language.

Clojure promotes code re-use through flexible functions that can work on many data-types:

> Many types, one interface
>
> One of Clojure’s core features is its generic data-manipulation API. A small set of functions can be used on all of Clojure’s built-in types. For example, the conj function (short for conjoin) adds an element to any collection, as shown in the following REPL session:
>
> user> (conj [1 2 3] 4)
> [1 2 3 4]
>
> user> (conj (list 1 2 3) 4)
> (4 1 2 3)
>
> user> (conj {:a 1, :b 2} [:c 3])
> {:c 3, :a 1, :b 2}
>
> user> (conj #{1 2 3} 4)
> #{1 2 3 4}
>
> Each data structure behaves slightly differently in response to the conj function (lists grow at the front, vectors grow at the end, and so on), but they all support the same API. This is a textbook example of polymorphism — many types accessed through one uniform interface.
>
> Polymorphism is a powerful feature and one of the foundations of modern programming languages. The Java language supports a particular kind of polymorphism called subtype polymorphism, which means that an instance of a type (class) can be accessed as if it were an instance of another type.
>
> In practical terms, this means that you can work with objects through a generic interface such as java.util.List without knowing or caring if an object is an ArrayList, LinkedList, Stack, Vector, or something else. The java.util.List interface defines a contract that any class claiming to implement java.util.List must fulfill.

“Many types, one interface” promotes code re-use, whereas the cornucopia of interfaces in OOP languages undermines code re-use. As Rich Hickey (inventor of Clojure) has said “**All that specificity [of interfaces/classes/types] kills your reuse!**”

Of course, if you are very careful, you can achieve fairly high levels of code re-use in any language, including any OOP language. But there tends to be trade-offs involved: achieving one goal means sacrificing another. To achieve high levels of re-use in OOP languages, one is often forced to write very small classes, which leads to an explosion in the number of classes in your system. **John Barker makes this point**:

> The typical college introduction to OOP starts with a gentle introduction to objects as metaphors for real world concepts. Very few real world OOP programs even consist entirely of nouns, they’re filled with verbs masquerading as nouns: strategies, factories and commands. Software as a mechanism for directing a computer to do work is primarily concerned with verbs.
>
> OOP programs that exhibit low coupling, cohesion and good reusability sometimes feel like nebulous constellations, with hundreds of tiny

> objects all interacting with each other. Sacrificing readability for changeability. Many of OOP best practices are in fact encouraged by functional programming languages.

Joe Armstrong (the inventor of Erlang) indirectly makes the point that plain data structures are more reusable than objects:

> Consider "time". In an OO language "time" has to be an object. But in a non OO language a "time" is an instance of a data type. For example, in Erlang there are lots of different varieties of time, these can be clearly and unambiguously specified using type declarations, as follows:
>
> ```
> -deftype day() = 1..31.
> -deftype month() = 1..12.
> -deftype year() = int().
> -deftype hour() = 1..24.
> -deftype minute() = 1..60.
> -deftype second() = 1..60.
> -deftype abstime() = {abstime, year(), month(), day(), hour(), min(), sec()}.
> -deftype hms() = {hms, hour(), min(), sec()}.
> ```
>
> Note that these definitions do not belong to any particular object. They are ubiquitous and data structures representing times can be manipulated by any function in the system.
>
> There are no associated methods.

Design Benefits

I'll repeat the quote above:

"Object Oriented Programs force designers to go through an extensive planning phase, which makes for better designs with less flaws."

This is a surprising appeal to the hated Waterfall development method:

> The waterfall model is a sequential design process, used in software development processes, in which progress is seen as flowing steadily downwards (like a waterfall) through the phases of Conception, Initiation, Analysis, Design, Construction, Testing, Production/Implementation and Maintenance. The waterfall development model originates in the manufacturing and construction industries; highly structured physical environments in which after-the-fact changes are prohibitively costly, if not impossible. Since no formal software development methodologies existed at the time, this hardware-oriented model was simply adapted for software development.

The waterfall method is risky because you only gain benefits from your software at the end of a long process — you might have to wait months, or even years, before the users of your software get to use it, and only then can they give you feedback about whether it actually helps them. **To reduce risk, many organizations have moved to lean and agile methods of development**:

> Agile is adaptive, not predictive.
> Do you remember what "waterfall" was like back in the day? We spent months gathering business requirements, writing specs, and designing, and then spent the next 10 months coding. Since we spent the first few months trying to predict what the next 10 months would entail, we could never accurately estimate how much work a task was supposed to be, and heaven forbid the requirement changed half way through! Agile is an attempt to shorten that cycle so we don't have to waste 10 months before find out something was wrong.

The waterfall method would be worth it if it lead to better software, but the opposite is true: because the development receives no feedback till the end of the process, the software produced by this method tends to be worse. **Agile Builds Trust with Fast Feedback**

> Agile development addresses this problem by iterative delivery of system functionality to meet requirements that people know they need today. Because people aren't asked to think of everything they could ever want, the number of requirements is much more manageable. And even within existing requirements, some requirements are always more important than others so if IT can deliver on those requirements right away (in 30 day cycles for instance), then business people get more value more quickly and the process builds mutual trust based on fast feedback and shared success.

More so, the idea that fast iterations lead to better outcomes is grounded on solid research, going all the way back to **John Boyd** who came up with **the OODA loop**:

> According to Boyd, decision-making occurs in a recurring cycle of observe-orient-decide-act. An entity (whether an individual or an organization) that can process this cycle quickly, observing and reacting to unfolding events more rapidly than an opponent, can thereby "get inside" the opponent's decision cycle and gain the advantage.

> …The same cycle operates over a longer timescale in a competitive business landscape, and the same logic applies. Decision makers gather information (observe), form hypotheses about customer activity and the intentions of competitors (orient), make decisions, and act on them. The cycle is repeated continuously. The aggressive and conscious application of the process gives a business advantage over a competitor who is merely reacting to conditions as they occur, or has poor awareness of the situation…
>
> The approach favors agility over raw power in dealing with human opponents in any endeavor.

About this:

"In addition, once a program reaches a certain size, Object Oriented Programs are actually easier to program than non-Object Oriented ones."

Basic math tells us this is untrue: since OOP depends on a graph of objects that mutate each other's state, the number of possible mutations increases exponentially with the number of objects, minus whatever limits can be imposed through contract enforcement and data hiding. But state is everywhere in an OOP program, so it takes extreme efforts to keep the number of possible mutations to a manageable level in a large program. When state is independent of classes, it becomes easier to centralize all the state in an app, and thus it becomes easier to protect it, but when state is inside of classes, as it must be with OOP, then it is difficult to protect, in part because it is spread out everywhere. Ask a Java programmer how many non-constant (mutable) variables exist in their current project and they would have to do some careful research to find out the answer. As a point of contrast, a typical Clojure app centralizes all state in a few global vars, which are then worked upon by all the functions in the program — the idea of "few nouns, lots of verbs" works to keep the state at a manageable amount. In large Clojure apps, like any software, the amount of state can eventually reach a scale where changes become complicated, but even then it is helped by the fact that the state is separate from the behavior.

As to large scale applications, it is worth noting that one of the Ericsson phone switches that Joe Armstrong worked on contains 1.7 million lines of Erlang code, and has a very high reliability requirement: no more than 2 hours of downtime every 40 years. This would be difficult or impossible using an OOP language.

Software Maintenance

About this:

"An Object Oriented Program is much easier to modify and maintain than a non-Object Oriented Program. So although a lot of work is spent before the program is written, less work is needed to maintain it over time."

I won't waste much time pointing out how wrong this is. The ability to maintain code over the long term depends crucially on the ability of programmers to understand the code, and no programmer can correctly reason about a graph of mutating state once that graph grows large enough — too many interactions begin to occur.

Also, OOP tends to be bloated, containing a lot of useless boilerplate code, so for the exact same amount of functionality, software written in a different style would be smaller and easier to think about.

Single responsibility principle

Uncle Bob argues for SRP in this way:

> The Single Responsibility Principle (SRP) says that a class should have one, and only one, reason to change. To say this a different way, the methods of a class should change for the same reasons, they should not be affected by different forces that change at different rates.
>
> As an example, imagine the following class in Java:
>
> class Employee
>
> {
> public Money calculatePay() {…}
> public void save() {…}
> public String reportHours() {…}
> }
>
> This class violates the SRP because it has three reasons to change. The first is the business rules having to do with calculating pay. The second is the database schema. The third is the format of the string that reports hours. We don't want a single class to be impacted by these three completely different forces. We don't want to modify the Employee class every time the accounts decide to change the format of the hourly report, or every time the DBAs make a change to the database schema, as well as every time the managers change the payroll calculation. Rather, we want to separate these functions out into different classes so that they can change independently of each other.
>
> Of course this seems to fly in the face of OO concepts since a good object should contain all the methods that manipulate it. However, decoupling tends to trump ideals like this. We don't want business rules coupled to report formats, and so we need to put these functions in different classes.

This is another case of a problem that only exists because of the ceremonies forced on us by OOP. Perhaps we should have global vars that are atoms, one for pay and one for hours, or perhaps we'd like to keep this in a hashmap inside of a ref (the ref giving us transactional guarantees so we don't have to worry about stuff like locks and synchronization) then we might have 100 functions, or 200 functions, that modify these vars, but at no point would we have a class about which we would have to ask "Am I obeying the Single Responsibility Principle?"

Open/closed principle

Up above I quoted John Barker:

"OOP programs that exhibit low coupling, cohesion and good reusability sometimes feel like nebulous constellations, with hundreds of tiny objects all interacting with each other."

This tendency toward having too many classes is exemplified in examples of the Open/closed principle:

A simple example of the Open/Closed Principle

> Now our customer, Aldford, wants us to build an application that can calculate the total area of a collection of rectangles.
>
> That's not a problem for us. We learned in school that the area of a rectangle is it's width multiplied with it's height and we mastered the for-each-loop a long time ago.
>
> ```
> public class AreaCalculator
> {
> public double Area(Rectangle[] shapes)
> {
> double area = 0;
> foreach (var shape in shapes)
> {
> area += shape.Width*shape.Height;
> }
>
> return area;
> }
> }
> ```
>
> We present our solution, the AreaCalculator class to Aldford and he signs us his praise. But he also wonders if we couldn't extend it so that it could calculate the area of not only rectangles but of circles as well.
>
> That complicates things a bit but after some pondering we come up with a solution where we change our Area method to accept a collection of objects instead of the more specific Rectangle type. Then we check what type each object is of and finally cast it to it's type and calculate it's area using the correct algorithm for the type.
>
> ```
> public double Area(object[] shapes)
> {
> double area = 0;
> foreach (var shape in shapes)
> {
> if (shape is Rectangle)
> {
> Rectangle rectangle = (Rectangle) shape;
> area += rectangle.Width*rectangle.Height;
> }
> else
> {
> Circle circle = (Circle)shape;
> area += circle.Radius * circle.Radius * Math.PI;
> }
> }
>
> return area;
> }
> ```
>
> The solution works and Aldford is happy.
>
> Only, a week later he calls us and asks: "extending the AreaCalculator class to also calculate the area of triangles isn't very hard, is it?". Of course in this very basic scenario it isn't but it does require us to modify the code. That is, AreaCalculator isn't closed for modification as we need to change it in order to extend it. Or in other words: it isn't open for extension.
>
> In a real world scenario where the code base is ten, a hundred or a thousand times larger and modifying the class means redeploying it's assembly/package to five different servers that can be a pretty big problem. Oh, and in the real world Aldford would have changed the requirements five more times since you read the last sentence :-)
>
> One way of solving this puzzle would be to create a base class for both rectangles and circles as well as any other shapes that Aldford can think of which defines an abstract method for calculating it's area.

```
public abstract class Shape
{
public abstract double Area();
}

Inheriting from Shape the Rectangle and Circle classes now looks like
this:

public class Rectangle : Shape
{
public double Width { get; set; }
public double Height { get; set; }
public override double Area()
{
return Width*Height;
}
}

public class Circle : Shape
{
public double Radius { get; set; }
public override double Area()
{
return Radius*Radius*Math.PI;
}
}
```

This conflates several issues that should be separate:

1.) establishing a data-type hierarchy

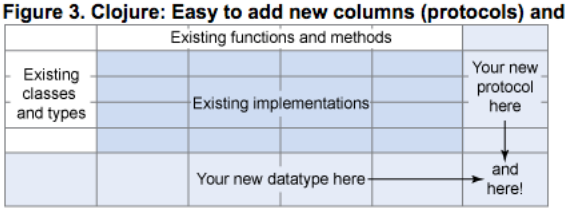
2.) enforcing a contract

3.) mutating state

Again, mixing together the definition of data-types, and their hierarchy, with the state and the behavior leads to bloated code.

Even if the above 3 problems could be solved (No True OOP Programmer allows bloated code) the kind of rigidity suggested by this principle will tend to limit code re-use. There needs to be easy ways to add new behavior to old interfaces, and to add new interfaces to old behavior. Toward that end, **Clojure offers protocols that allow extending old types to new behavior**:

Figure 3. Clojure: Easy to add new columns (protocols) and rows (datatypes)



Here is an example of adding new methods to old objects:

```
; From Sean Devlin's talk on protocols at Clojure Conj

(defprotocol Dateable
(to-ms [t]))

(extend java.lang.Number
Dateable
{:to-ms identity})

(extend java.util.Date
Dateable
{:to-ms #(.getTime %)})

(extend java.util.Calendar
Dateable
{:to-ms #(to-ms (.getTime %))})
```

This is Clojure's answer to the expression problem. This is a simple and low-ceremony way of extending existing interfaces and promoting code re-use.

Over time, programmers working with OOP languages build up certain instincts, and certain kinds of flexibility go against those instincts. For instance, in his talk with Yukihiro Matsumoto (the inventor of Ruby), **Bill Venners says it is "a bit scary" to think about changing interfaces at runtime**:

> Bill Venners: In Ruby, I can add methods and variables to objects at runtime. I have certainly added methods and variables to classes, which become objects at runtime, in other languages. But in Java, for example, once a class is loaded or an object is instantiated, its interface stays the same. Allowing the interface to change at runtime seems a bit scary to me. I'm curious how I would use that feature in Ruby. What's the benefit of being able to add methods at runtime?

> Yukihiro Matsumoto: One example is a proxy. Instead of designing individual proxy classes for each particular class, in Ruby you can create an all purpose proxy that can wrap any object. The proxy can probe the object inside of it and just morph into the proxy for that object. The proxy can add methods to itself so it has the same interface as the wrapped object, and each of those methods can delegate to the corresponding method in the wrapped object. So an all-purpose proxy, which can be used to wrap any object, is an example of how a library class can adapt to the environment.

Interface segregation principle

This is how Wikipedia defines the principle:

> The interface-segregation principle (ISP) states that no client should be forced to depend on methods it does not use. ISP splits interfaces which are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them. Such shrunken interfaces are also called role interfaces. ISP is intended to keep a system decoupled and thus easier to refactor, change, and redeploy. Within object-oriented design, interfaces provide layers of abstraction that facilitate conceptual explanation of the code and create a barrier preventing dependencies. According to many software experts who have signed the Manifesto for Software Craftsmanship, writing well-crafted and self-explanatory software is almost as important as writing working software. Using interfaces to further describe the intent of the software is often a good idea. A system may become so coupled at multiple levels that it is no longer possible to make a change in one place without necessitating many additional changes. Using an interface or an abstract class can prevent this side effect.

I think we can all agree that contracts both enforce strictness and also communicate intent, both of which are important. If we substitute “contracts” for “interfaces” we can agree that “Using interfaces to further describe the intent of the software is often a good idea.” But we also know what a pain it is to have to deal with a huge number of tiny classes, and we know that its better to have fewer, more general interfaces, so ISP sounds terrible, because it “splits interfaces which are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them.”

ISP is a crutch that props up OOP, and OOP needs propping up because it is a disaster. “ISP is intended to keep a system decoupled and thus easier to refactor, change, and redeploy” but the larger question is “Does OOP make it easy to refactor, change, and redeploy?” If OOP undermines these things, then ISP is no more than medicine for a patient who is very sick.

We can all agree that “writing… self-explanatory software is almost as important as writing working software”. But how do we create self-explanatory software? The crucial thing is to reveal the underlying model of the data, that is, the data-types and their hierarchy. **Fredrick Brooks once said**:

> Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won’t usually need your flowcharts; they’ll be obvious.

Eric Raymond updated this to:

> Show me your code and conceal your data structures, and I shall continue to be mystified. Show me your data structures, and I won’t usually need your code; it’ll be obvious.

But how can I find your data structures? **Joe Armstrong (inventor of Erlang) explains how OOP undermines this crucial bit of clarity**:

> In an Object Oriented Programming Language (OOPL) data type definitions belong to objects. So I can’t find all the data type definition in one place. In Erlang or C I can define all my data types in a single include file or data dictionary. In an OOPL I can’t — the data type definitions are spread out all over the place.
>
> Let me give an example of this. Suppose I want to define a ubiquitous data structure. A ubiquitous data type is a data type that occurs “all over the place” in a system.
>
> As lisp programmers have know for a long time it is better to have a smallish number of ubiquitous data types and a large number of small functions that work on them, than to have a large number of data types and a small number of functions that work on them.
>
> A ubiquitous data structure is something like a linked list, or an array or a hash table or a more advanced object like a time or date or filename.
>
> In an OOPL I have to choose some base object in which I will define the ubiquitous data structure, all other objects that want to use this data structure must inherit this object. Suppose now I want to create some “time” object, where does this belong and in which object…

Here is a simpler definition of Interface segregation principle: “many client-specific interfaces are better than one general-purpose interface.” As discussed above, this is an awful idea, but perhaps it is more relevant to point out that even among OOP proponents, **a large group has been moving toward simplified interfaces, as Bruce Eckel first noted in 2005**:

> Recently, two people that I know have gotten into an indirect tiff about what “simplicity” means. Martin Fowler first made the assertion that the Ruby library was simple and elegant.
>
> …Martin’s argument is that Java’s List interface requires you to say aList.get(aList.size -1) to get the last element, and this seemed silly to him. Which it is, if you have unified all sequence containers (that is, list containers) into a single type, as Ruby and Python do. Java, however, follows the C++ STL approach of providing different types based on the efficiency of various operations. The Java libraries do not unify to a single list type because of efficiency issues, so you have to decide if you are going to be fetching the last element from a list a lot, and if you are you use a LinkedList, which does have a getLast() method

Efficiency issues are the only issues which justify a confusing plethora of interfaces, but these issues are never the ones that OOP proponents use to defend OOP. After all, if your principal concern is efficiency issues, perhaps you need to give up on the abstractions of OOP and drop down to a lower level language?

Dependency inversion principle

I already touched on this above when I quoted Chad Myers, and I remarked at the astonishing complexity introduced to deal with a problem that only exists because of OOP: how to instantiate or compose objects.

I am exaggerating a little bit. The truth is that any system that uses plugins (**as defined by Patterns of Enterprise Application Architecture** will need a decoupled way to polymorphically figure out which concrete implementation of the plugin should be used. However, non-OOP languages do not wrestle with the question “How do I instantiate the objects that my current object depends upon?” Outside of OOP, we do not need to worry about instantiation, and so configuration tends to be simpler.

I first learned about the Dependency Inversion Principle, and Dependency Injection, in 2004 when I read **Martin Fowler’s essay on the subject**. I am simply going to pull some quotes from his essay. As you read this, please ask yourself “What are we gaining in exchange for all this work?” I believe the answer is “nothing”: we are struggling with a problem that only exists if we use OOP:

> In the Java community there’s been a rush of lightweight containers that help to assemble components from different projects into a cohesive application. Underlying these containers is a common pattern to how they perform the wiring, a concept they refer under the very generic name of “Inversion of Control”…
>
> One of the entertaining things about the enterprise Java world is the huge amount of activity in building alternatives to the mainstream J2EE technologies, much of it happening in open source. A lot of this is a reaction to the heavyweight complexity in the mainstream J2EE world, but much of it is also exploring alternatives and coming up with creative ideas. A common issue to deal with is how to wire together different elements: how do you fit together this web controller architecture with that database interface backing when they were built by different teams with little knowledge of each other.
>
> There are three main styles of dependency injection. The names I’m using for them are Constructor Injection, Setter Injection, and Interface Injection.
>
> [launches into example of listing movies]
>
> …PicoContainer uses a constructor to decide how to inject a finder implementation into the lister class. For this to work, the movie lister class needs to declare a constructor that includes everything it needs injected.
>
> class MovieLister…
> public MovieLister(MovieFinder finder) {
> this.finder = finder;
> }
>
> The finder itself will also be managed by the pico container, and as such will have the filename of the text file injected into it by the container.
>
> class ColonMovieFinder…
> public ColonMovieFinder(String filename) {
> this.filename = filename;
> }
>
> The pico container then needs to be told which implementation class to associate with each interface, and which string to inject into the finder.
>
> private MutablePicoContainer configureContainer() {
> MutablePicoContainer pico = new DefaultPicoContainer();
> Parameter[] finderParams = {new ConstantParameter(“movies1.txt”)};
> pico.registerComponentImplementation(MovieFinder.class,
> ColonMovieFinder.class, finderParams);
> pico.registerComponentImplementation(MovieLister.class);
> return pico;
> }

> This configuration code is typically set up in a different class. For our example, each friend who uses my lister might write the appropriate configuration code in some setup class of their own. Of course it's common to hold this kind of configuration information in separate config files. You can write a class to read a config file and set up the container appropriately. Although PicoContainer doesn't contain this functionality itself, there is a closely related project called NanoContainer that provides the appropriate wrappers to allow you to have XML configuration files. Such a nano container will parse the XML and then configure an underlying pico container. The philosophy of the project is to separate the config file format from the underlying mechanism.

I remember reading this in 2004, when I was still insecure about my opinions as a computer programmer, and I remember feeling stupid and wondering why we needed to do so much work. Nowadays I would say with confidence: there is no need to do so much work.

Interestingly, towards the end, Martin Fowler argues that the advantage of injection is "simple conventions":

> The advantage of injection is primarily that it requires very simple conventions – at least for the constructor and setter injections. You don't have to do anything odd in your component and it's fairly straightforward for an injector to get everything configured.
>
> Interface injection is more invasive since you have to write a lot of interfaces to get things all sorted out. For a small set of interfaces required by the container, such as in Avalon's approach, this isn't too bad. But it's a lot of work for assembling components and dependencies, which is why the current crop of lightweight containers go with setter and constructor injection.
>
> The choice between setter and constructor injection is interesting as it mirrors a more general issue with object-oriented programming – should you fill fields in a constructor or with setters.
>
> My long running default with objects is as much as possible, to create valid objects at construction time. This advice goes back to Kent Beck's Smalltalk Best Practice Patterns: Constructor Method and Constructor Parameter Method. Constructors with parameters give you a clear statement of what it means to create a valid object in an obvious place. If there's more than one way to do it, create multiple constructors that show the different combinations.
>
> Another advantage with constructor initialization is that it allows you to clearly hide any fields that are immutable by simply not providing a setter. I think this is important – if something shouldn't change then the lack of a setter communicates this very well. If you use setters for initialization, then this can become a pain. (Indeed in these situations I prefer to avoid the usual setting convention, I'd prefer a method like initFoo, to stress that it's something you should only do at birth.)
>
> But with any situation there are exceptions. If you have a lot of constructor parameters things can look messy, particularly in languages without keyword parameters. It's true that a long constructor is often a sign of an over-busy object that should be split, but there are cases when that's what you need.

Stop for a moment and think about how many brilliant minds have wasted countless hours debating "a more general issue with object-oriented programming – should you fill fields in a constructor or with setters." If you have ever read about Leonardo DaVinci then you've probably thought about what a tragedy it was that such a great visionary lived at a time when the technology was lacking to allow him to move forward with his ideas, and likewise, it is a tragedy that so many hyper-intelligent people had the peak of their careers overlap with the time when OOP ideology was at its peak.

There are situations where software configuration is especially difficult, for instance, with hardware, if you are creating software for a stereo system, or a robot, and you want the software to be generic enough that it will work for different kinds of hardware of that category. But at least then, the difficulty of configuration is arising directly from the problem that you are trying to solve, whereas OOP inflicts this problem on us gratuitously.

To the extent that we are wrestling with issues instantiation, we are dealing with a problem that is unique to OOP — and this is a good argument against OOP. To the extent that we are talking about issues of dependancies, the problem is real, though for some categories of software, such as software that is served from a server (in a client-server architecture, for instance, a website) configuration can nowadays be automated with something like Chef or Ansible. You might argue "That only moves the complexity somewhere else, that does not solve the complexity" but that takes us to a subject that, though important, has nothing to do with OOP.

Static type checking

Software development is a rapidly evolving field and our ideas of "best practice" change with each decade. We have learned bitter lessons from the failures of past theories. Intelligent criticism of past mistakes should always be welcome. And yet, I would note, if you search on the Web for criticism of existing software development practices, a surprising fact is how much of that criticism regards itself as criticism of OOP, even when it has nothing to do with OOP. This is the mental hegemony of OOP — to criticize software development must mean criticising objects. **Consider**

the slideshow essay "Objects have failed", written by an author who thinks they are attacking OOP, though none of their criticisms are specific to OOP:

> L. Peter Deutsch: Seven Fallacies of Distributed Computing
>
> • The network is reliable
>
> • Latency is zero
>
> • Bandwidth is infinite
>
> • The network is secure
>
> • Topology doesn't change
>
> • There is one administrator
>
> • Transport cost is zero

This is a good list of mistakes to avoid, but it has nothing to do with OOP.

The rest of the essay is an attack on static data-type enforcement, though it seems to be under the misguided belief that it is attacking OOP:

> The thrust of much of the last 30 years of computer science has been to try to make it so that failures can't happen — by using more and more static notions, by making programming languages more strict, and by heavyweight methodologies.
>
> Although one view of objects is that they are autonomous entities with identity and even a sense of self-preservation, in fact objects have been taken over by static thinkers who have imposed types and static notions making software brittle in their attempts to make it robust.

Let us remember that there are many OOP languages that are dynamic: Ruby, Javascript, Groovy, etc. This essay is attacking a very particular type of OOP:

> "The strong typing of object-oriented languages encourages narrowly defined packages that are hard to reuse. Each package requires objects of a specific type; if two packages are to work together, conversion code must be written to translate between the types required by the packages." [John K. Ousterhout]

I am personally open-minded in the debate between dynamic versus strict typing. When I confront a problem that I have never confronted before, then I prefer to start dynamic and eventually become strict. For this reason, I've become a fan of optional type systems that I can layer onto my code as I come to understand a particular problem domain. As I wrote in "**How ignorant am I, and how do I formally specify that in my code?**":

> The most obvious argument against static typing is that it claims a level of confidence that it can not possibly have. Developers using static typing specify all types at the beginning of their project, before they have learned anything about the problem they need to solve. The advocates of static typing will make the counter-argument that it is easy to change types as one learns more about the problem. But I would assert that it is misleading to pretend that you understand the system at the beginning. I want to be honest about how ignorant I am when I begin to work on a problem that I have never encountered before.

The above essay "Objects have failed" offers this quote from William Cook:

> The key question is whether typing has reduced the expressiveness of languages. I guess that my experience says "yes." This is because type systems are not generally able to describe the sophisticated abstractions that we want to build. Although these abstractions may be sophisticated, that does not mean they are impossible to understand — they are often quite intuitive. But by their very nature, type systems are required to be able to efficiently and automatically prove assertions about programs, and this will tend to impose limitations on what kinds of things a type system can do. We also do not want our types to be larger than our programs, so this imposes limits on their complexity. One thing that is not recognized enough is that types are partial specifications, and it might be nice to allow different levels of detail in your types, without having to resort to "object" (the type with no detail).
>
> …Static typing is tied to the notion of designing up front. It is also connected with fear. I have to say that I really like types, in general. But I don't like them when the prevent me from expressing myself. Or, as is more likely but much more subtle, when they cause a language designer to distort a language to suit the needs of the type system.
> — William Cook

Mark Taver (inventor of the Qi/Shen language) recently wrote:

> (quote from someone named Racketnoob) "I have the strange feeling that types hamper a programmer's creativity."

> The above sentence is a compact summary of the reluctance that programmers often feel in migrating to statically typed languages — that they are losing something, a degree of freedom that the writer identifies as hampering creativity.
>
> Is this true? I will argue, to a degree — yes. A type checker for a functional language is in essence, an inference engine; that is to say, it is the machine embodiment of some formal system of proof. What we know, and have known since Godel's incompleteness proof, is that the human ability to recognize truth transcends our ability to capture it formally. In computing terms our ability to recognize something as correct predates and can transcend our attempt to formalise the logic of our program. Type checkers are not smarter than human programmers, they are simply faster and more reliable, and our willingness to be subjugated to them arises from a motivation to ensure our programs work.
>
> That said, not all type checkers are equal. The more rudimentary and limited our formal system, the more we may have to compromise on our natural coding impulses. A powerful type system and inference engine can mitigate the constraints placed on what Racketnoob terms our creativity. At the same time a sophisticated system makes more demands of the programmer in terms of understanding. The invitation of adding types was thus taken up by myself, and the journey to making this program type secure in Shen emphasizes the conclusion in this paragraph.

I'll offer an example from Haskell, which is a beautiful language with very powerful abilities for defining the contracts on any function:

```
multThree :: (Num a) => a -> a -> a -> a
multThree x y z = x * y * z
```

The first line is the contract, the second line is the function.

> Take a look at this offensively simple function:
>
> ```
> multThree :: (Num a) => a -> a -> a -> a
> multThree x y z = x * y * z
> ```
>
> What really happens when we do multThree 3 5 9 or ((multThree 3) 5) 9? First, 3 is applied to multThree, because they're separated by a space. That creates a function that takes one parameter and returns a function. So then 5 is applied to that, which creates a function that will take a parameter and multiply it by 15. 9 is applied to that function and the result is 135 or something. Remember that this function's type could also be written as
>
> multThree :: (Num a) => a -> (a -> (a -> a)).
>
> The thing before the -> is the parameter that a function takes and the thing after it is what it returns. So our function takes an a and returns a function of type (Num a) => a -> (a -> a). Similarly, this function takes an a and returns a function of type (Num a) => a -> a. And this function, finally, just takes an a and returns an a.

If you want really strong typing, you should use non-OOP languages.

What do critics of OOP say?

Above, I've talked about 12 items that proponents of OOP seem to regard as unique strengths of OOP. I believe I have shown 2 things:

1.) Those items which are potentially good (data hiding, contract enforcement, polymorphism) are not unique to OOP and, in fact, stronger versions of these things are available in non-OOP languages.

2.) Those items that are unique to OOP (dependency injection, constructor values) are awful and exist only because OOP is awful.

However, critics of OOP would go beyond these 12 items, and list a variety of flaws which OOP proponents seem unaware of, but which loom large in the imaginations of those of us who behold OOP in horror. There are issues of concurrency, which I'll talk about further below. For now I want to focus on notation.

I'll admit, there is nothing natural about computer programming languages, and we all grow used to what we work with, and no one doubts that the **Sapir-Whorf hypothesis** applies to programming languages. If you spend 10 years working with Java then Java seems natural to you, and if you spend 10 years working with Lisp, then Lisp seems natural to you. All the same, to the limited extent that it is possible to be objective when comparing languages, we can look at how close the code matches the task we are trying to achieve. In his presentation "**Simple made easy**" Rich Hickey talks about "incidental complexity". How much do you have to engage in stupid tricks and worthless ceremony to get a job done?

Let's look at 2 examples of FizzBuzz. **Here is an example in Java**:

```
public class FizzBuzz {
public static void main(String[] args) {
for(int i = 1; i <= 100; i++) {
if (((i % 5) == 0) && ((i % 7) == 0))
System.out.print("fizzbuzz");
```

```
else if ((i % 5) == 0) System.out.print("fizz");
else if ((i % 7) == 0) System.out.print("buzz");
else System.out.print(i);
System.out.print(" ");
}
System.out.println();
}
}
```

And here is an example in Clojure:

```
(doseq [n (range 1 101)]
(println
(match [(mod n 3) (mod n 5)]
[0 0] “FizzBuzz”
[0 _] “Fizz”
[_ 0] “Buzz”
:else n)))
```

Which code is more direct to its task? (I worry that I will be misunderstood: sometimes these examples are used to talk about mutable versus immutable state. I am not raising that issue here. I ask you to focus merely on notation: which notation allows the clearest expression of the FizzBuzz problem?)

Let’s look at another example. **Here we ask if a string is upper-case in Scala**:

```
public boolean hasUpperCase(String word) {
if (null != word)
return any(charactersOf(word), new Predicate() {
public boolean apply(Character c) {
return isUpperCase(c);
}
})
else
return false;
}
```

And here is the same in Clojure:

```
(defn has-uppercase? [string]
(some #(Character/isUpperCase %) string))
```

Which code is more direct in its task?

The OOP version also introduces an anti-pattern that Miško Hevery (and other engineers at Google) have criticized in their sarcastic post “How to Write Untestable Code”:

> Make Your Own Dependencies – Instantiate objects using new in the middle of methods, don’t pass the object in. This is evil because whenever you new up an object inside a block of code and then use it, anyone that wants to test that code is also forced to use that concrete object you new’ed up. They can’t “inject” a dummy, fake, or other mock in to simplify the behavior or make assertions about what you do to it.

As mentioned before the issue of dependency comes up in OOP in a particuarlly awful form: the concern about how to instantiate objects. Someone is bound to argue “The Clojure version has a dependency on the Character class” but the point is that you don’t have to instantiate the Character class.

Why does anyone use OOP with PHP?

This is a bit of an aside, but I think it is revealing of how much the tech industry has fallen under the thrall of OOP. There was no reason to add OOP features to PHP, and yet OOP features were added to PHP: this teaches us a lot about the dominance of OOP ideology.

The problems with OOP were not well understood during the 1980s and 1990s. At that time, OOP was the “believable promise” upon which the industry was willing to gamble its future. Hopes were high that OOP was the silver bullet that would finally make software development painless. PHP did not get real OOP until 2004, at which point the problems with OOP were better known. PHP had become popular before it had OOP, and it seems to me that it could have remained a good language, for its niche, if it never had any OOP features. I am often told “PHP needed OOP to gain acceptance in the Enterprise” which should make all of us ask 2 questions:

1.) why does the Enterprise favor OOP?

2.) did PHP need to gain acceptance in the Enterprise?

PHP has one great strength: it is easy to learn, and so there is an abundance of PHP programmers, and so they are fairly cheap to hire. This is an important circumstance which some corporations find very attractive. Please note, I am not being ironic here, I really do think that some companies are wise to follow a “low skill, low cost” approach to software development. Not every project needs genius, some projects just need something simple, which can be built by someone who is a novice.

The problems with PHP, as a language, are well known. You can read the much discussed essay “**PHP: A Fractal Of Bad Design**” to get a sense of some of the weaknesses of PHP:

> PHP is not merely awkward to use, or ill-suited for what I want, or suboptimal, or against my religion. I can tell you all manner of good things about languages I avoid, and all manner of bad things about languages I enjoy. Go on, ask! It makes for interesting conversation.
>
> PHP is the lone exception. Virtually every feature in PHP is broken somehow. The language, the framework, the ecosystem, are all just bad. And I can't even point out any single damning thing, because the damage is so systemic. Every time I try to compile a list of PHP gripes, I get stuck in this depth-first search discovering more and more appalling trivia. (Hence, fractal.)
>
> PHP is an embarrassment, a blight upon my craft. It's so broken, but so lauded by every empowered amateur who's yet to learn anything else, as to be maddening. It has paltry few redeeming qualities and I would prefer to forget it exists at all.

Over the last 15 years, my own feelings for PHP have traveled 270 degrees: from love to hate to acceptance. Back in 2001 I built a CMS with PHP, and for several years I ran a business that was based around that CMS. Later I became an expert with the Symfony framework, a heavyweight OOP framework, and Symfony made me hate PHP. You can read my essay "**Why PHP is obsolete**" to get a feel for how anti-PHP I became. But now I've come round to the view that the worst thing about PHP is its OOP features, and if you avoid them you end up with a script language that has some utility for writing blog software or cron jobs.

What is PHP good for? The 2 most popular projects written in PHP are WordPress and Drupal. Both of these projects avoid OOP (or rather, Drupal avoided OOP until this year). That says a great deal about the strengths and weaknesses of PHP. There are no "killer apps" written in OOP PHP. It's a language for simple functions doing simple things.

When I worked at WineSpectator.com and Timeout.com both companies were using the Symfony framework. I ask this in all seriousness: why would any corporation first decide to use PHP, and then decide to use a complex framework like Symfony? The main advantage of PHP is that it is easy to learn, and so there is an abundance of PHP programmers — this is what makes it attractive to corporations. But if corporations have decided to follow a "lost skill, low cost" model of software development, how does it make any sense to then use a framework that has deep object hierarchies, " **Design Patterns**", a Dependency Injection container, etc? Does it make sense to hire novice programmers and then ask them to learn a framework that is almost as complex as Struts and Spring and Rails? Or, if you are not hiring novice programmers, then why are you using PHP?

More so, PHP is slow, but using OOP with PHP is terribly slow. If you look at the TechEmpower comparison of frameworks, **raw PHP is in the middle, whereas PHP OOP frameworks, such as Symfony, are at the bottom, and Cake failed to complete the test**. They speculate that some PHP functions, such as raw calls to the MySql functions, have been highly optimized, and therefore have decent speed. But the PHP OOP frameworks are a performance disaster.

Still, I no longer hate PHP. If I need to write a simple cron script, I can use Ruby or PHP, but creating a Ruby Gem has a few extra steps ("bundle gem", "rake install", etc) so PHP has slightly less ceremony for writing a simple cron script. **As David Heinemeier Hansson (inventor of Ruby On Rails) says**:

> [PHP] has really received an unfair reputation. For the small things I've been using it for lately, it's absolutely perfect.
>
> I love the fact that it's all just self-contained. That the language includes so many helpful functions in the box. And that it managed to get distributed with just about every instance of Apache out there.
>
> For the small chores, being quick and effective matters far more than long-term maintenance concerns. Or how pretty the code is. PHP scales down like no other package for the web and it deserves more credit for tackling that scope.

It would be interesting to know if he used the OOP features of PHP. I am assuming that he did not.

Concurrency Oriented Programming

For the current generations of working computer programmers, OOP programming has attained what I think can reasonably be described as "mental hegemony". By that I mean that the ideas of OOP are so pervasive that many programmers are unable to think about the problems of computation outside of the OOP perspective. Issues of static data-type checking, issues of monolithic versus decomposed services, issues of concurrency, issues of algorithms, all are now typically discussed under the lens of OOP. For an example, consider the book "Java Concurrency In Practice" (JCIP). I am a huge fan of this (copyright 2006) book, written by Brian Goetz, etc, **and the book is often recommended to people who want to learn about concurrent programming (see the StackOverflow question "Recommend a good resource for approaches to concurrent programming?")**. In the year 2014, it is difficult to imagine a book about concurrency, written for the pure-logic or functional styles of programming, becoming so widely recommended. Obviously you can do concurrent programming using a non-OOP language, but for now the best books on the subject will tend to be written from an OOP perspective. **If you look at the book list put together by Rich Hickey (inventor of Clojure), JCIP is the only book that focuses exclusively on concurrency**.

I've read JCIP twice, and I would estimate that about 50% of it is specific to OOP. All of the code examples are in Java. And yet, the other 50% of the book has nothing to do with OOP, it is simply good advice for building concurrent applications. Consider this bit from page 221, and as you read this, try to spot all the places where a subject is touched upon that is not strictly related to OOP:

> While the goal may be to improve performance overall, using multiple threads always introduces some performance costs compared to the single-threaded approach. These include the overhead associated with coordinating between threads (locking, signaling, and memory synchronization), increased context switching, thread creation and teardown, and scheduling overhead. When threading is employed effectively, these costs are more than made up for by greater throughput, responsiveness, or capacity. On the other hand, a poorly designed concurrent application can perform even worse than a comparable sequential one.
>
> (A colleague provided this amusing anecdote: he had been involved in the testing of an expensive and complex application that managed its work via a tunable thread pool. After the system was complete, testing showed that the optimal number of threads was… 1. This should have been obvious from the outset; the target system was a single-CPU system and the application was almost entirely CPU bound.)
>
> In using concurrency to achieve better performance, we are trying to do two things: utilize the processing resources we have more effectively, and enable our program to exploit additional processing resources if they become available. From a performance monitoring perspective, this means we are looking to keep the CPUs as busy as possible. (Of course, this doesn't mean burning cycles with useless computation, we want to keep the CPUs busy with useful work.) If the program is compute-bound, then we may be able to increase capacity by adding more processors; if it can't keep the processors we have busy, adding more won't help. Threading offers a means to keep the CPU(s) "hotter" by decomposing the application so there is always more work to be done by an available processor.
>
> …The familiar three-tier application model — in which presentation, business logic, and persistence are separated and may be handled by different systems — illustrates how improvements in scalability often come at the expense of performance. A monolithic application where presentation, business logic, and persistence are intertwined would almost certainly provide better performance for the first unit of work than would a well-factored multi-tier implementation distributed over multiple systems. How could it not? The monolithic application would not have the network latency inherent in handing off tasks between tiers, nor would it have to pay the costs inherent in separating a computational process into distinct abstracted layers (such as queuing overhead, coordination overhead, and data copying).
>
> However, when the monolithic system reaches its processing capacity, we could have a serious problem: it may be prohibitively difficult to significantly increase capacity. So we often accept the performance costs of longer service time or greater computing resources used per unit of work so that our application can scale to handle greater load by adding resources.

Here we have 5 paragraphs full of insights about computing, none of which has anything to do with OOP. But, for now, a successful book about concurrency is inevitably written from the OOP perspective. That is the mental hegemony of OOP.

What are the alternatives to OOP? **Wikipedia lists more than 50 programming paradigms** though some of them overlap and are redundant. I will focus on just one alternative to OOP, a kind of programming that Joe Armstrong (inventor of Erlang) calls "Concurrency Oriented Programming".

Generally speaking, when I see the word "thesis" I think "boring", so I had low expectations when I opened up **Joe Armstrong's thesis about the creation of Erlang**. I only meant to skim it, but his work is surprisingly readable and I was drawn along till I had read the whole thing. I strongly recommend it.

We all know that **Moore's Law has come to an end**, and this increases the importance of concurrent programming, as computers now speed up through the use of multiple CPUs, rather than faster CPUs.

Joe Armstrong lists the requirements for complex telephony equipment, which sounds like the kind of system you could build with OOP, if only it weren't for the reliability that is mentioned at the end:

> Quality requirements — switching systems should run with an acceptable level of service even in the presence of errors. Telephone exchanges are expected to be extremely reliable. Typically having less than two hours of down-time in 40 years.

Let's repeat that last part:

> Typically having less than two hours of down-time in 40 years.

Can this be done with OOP? Of course not! Using OOP, you will face two hours of down-time every month, or maybe every week, rather than every 40 years.

Erlang is a work of genius and I feel some frustration that it does not get more attention. For this reason, I will quote at length some of the ideas that lead to **Concurrency Oriented Programming**

> Concurrency Oriented Programming also provides the two major advantages commonly associated with object-oriented programming. These are polymorphism and the use of defined protocols having the same message passing interface between instances of different process types.
>
> When we partition a problem into a number of concurrent processes we can arrange that all the processes respond to the same messages (ie they are polymorphic,) and that they all follow the same message passing interface…
>
> When we perform a simple action, like driving a car along a freeway, we are aware of the fact that there may be several hundreds of cars within our immediate environment, yet we are able to perform the complex task of driving a car, and avoiding all these potential hazards without even thinking about it.
>
> In the real world sequential activities are a rarity. As we walk down the street we would be very surprised to find only one thing happening, we expect to encounter many simultaneous events.
>
> If we did not have the ability to analyze and predict the outcome of many simultaneous events we would live in great danger, and tasks like driving a car would be impossible. The fact that we can do things which require processing massive amounts of parallel information suggests that we are equipped with perceptual mechanisms which allow us to intuitively understand concurrency without consciously thinking about it.
>
> When it comes to computer programming things suddenly become inverted. Programming a sequential chain of activities is viewed as the norm , and in some sense is thought of as being easy, whereas programming collections of concurrent activities is avoided as much as possible, and is generally perceived as being difficult.
>
> I believe that this is due to the poor support which is provided for concurrency in virtually all conventional programming languages. The vast majority of programming languages are essentially sequential; any concurrency in the language is provided by the underlying operating system, and not by the programming language.
>
> We often want to write programs that model the world or interact with the world. Writing such a program in a COPL is easy. Firstly, we perform an analysis which is a three-step process:
>
> 1. We identify all the truly concurrent activities in our real world activity.
>
> 2. We identify all message channels between the concurrent activities.
>
> 3. We write down all the messages which can flow on the different message channels
>
> Now we write the program. The structure of the program should exactly follow the structure of the problem. Each real world concurrent activity should be mapped onto exactly one concurrent process in our programming language. If there is a 1:1 mapping of the problem onto the program we say that the program is isomorphic to the problem. It is extremely important that the mapping is exactly 1:1. The reason for this is that it minimizes the conceptual gap between the problem and the solution. If this mapping is not 1:1 the program will quickly degenerate, and become difficult to understand. This degeneration is often observed when non-CO languages are used to solve concurrent problems. Often the only way to get the program to work is to force several independent activities to be controlled by the same language thread or process. This leads to a inevitable loss of clarity, and makes the programs subject to complex and irreproducible interference errors.
>
> Isolation has several consequences:
>
> 1. Processes have “share nothing” semantics. This is obvious since they are imagined to run on physically separated machines.
>
> 2. Message passing is the only way to pass data between processes. Again since nothing is shared this is the only means possible to exchange data.
>
> 3. Isolation implies that message passing is asynchronous. If process communication is synchronous then a software error in the receiver of a message could indefinitely block the sender of the message destroying the property of isolation.
>
> 4. Since nothing is shared, everything necessary to perform a distributed computation must be copied. Since nothing is shared, and the only way to communicate between processes is by message passing, then we will never know if our messages arrive (remember we said that message passing is inherently unreliable.) The only way to

know if a message has been correctly sent is to send a confirmation message back

Programming a system of processes subject to the above rules may appear at first sight to be difficult — after all most concurrency extensions to sequential programming languages provide facilities for almost exactly the opposite, providing things like locks, and semaphores, and provision for shared data, and reliable message passing. Fortunately, the opposite turns out to be true — programming such a system turns out to be surprisingly easy, and the programs you write can be made scalable, and fault-tolerant, with very little effort.

About Erlang's robustness in the face of failure:

The inability to isolate software components from each other is the main reason why many popular programming languages cannot be used for making robust system software.

It is essential for security to be able to isolate mistrusting programs from one another, and to protect the host platform from such programs. Isolation is difficult in object-oriented systems because objects can easily become aliased.
—Bryce

Bryce goes on to say that object aliasing is difficult if not impossible to detect in practice, and recommends the use of protection domains (akin to OS processes) to solve this problem

In a paper on Java Czajkowski, and Daynes, from Sun Microsystems, write:

The only safe way to execute multiple applications, written in the Java programming language, on the same computer is to use a separate JVM for each of them, and to execute each JVM in a separate OS process. This introduces various inefficiencies in resource utilization, which downgrades performance, scalability, and application startup time. The benefits the language can offer are thus reduced mainly to portability and improved programmer productivity. Granted these are important, but the full potential of language-provided safety is not realized. Instead there exists a curious distinction between "language safety," and "real safety".

… tasks cannot directly share objects, and that the only way for tasks to communicate is to use standard, copying communication mechanisms, …

These conclusions are not new. Very similar conclusions were arrived at some two decades earlier by Jim Gray who described the architecture of the Tandem Computer in his highly readable paper Why do computers stop and what can be done about it. He says:

As with hardware, the key to software fault-tolerance is to hierarchically decompose large systems into modules, each module being a unit of service and a unit of failure. A failure of a module does not propagate beyond the module.
…
The process achieves fault containment by sharing no state with other processes; its only contact with other processes is via messages carried by a kernel message system.

Languages which support this style of programming (parallel processes, no shared data, pure message passing) are what Andrews and Schneider refer to as a "Message oriented languages." The language with the delightful name PLITS (Programming language in the sky) is probably the first example of such a programming language:

The fundamental design decision in the implementation of RIG (RIG was a small system written in PLITS) was to allow a strict message discipline with no shared data structures. All communication between user and server messages is through messages which are routed by the Aleph kernel. This message discipline has proved to be very flexible and reliable.

…Bearing in mind these arguments, and our original requirements I advocate a system with the following properties:

1. Processes are the units of error encapsulation — errors occurring in a process will not affect other processes in the system. We call this property strong isolation.

2. Processes do what they are supposed to do or fail as soon as possible.

3. Failure, and the reason for failure, can be detected by remote processes.

4. Processes share no state, but communicate by message passing.

This quote from Gray's paper may seem irrelevant to a conversation about OOP, but the conclusion I draw from it is that OOP has nothing to do with building robust, fault-tolerant systems (the features needed to build robust, fault-tolerant systems can be implemented better with non-OOP languages).

> There is considerable controversy about how to modularize software. Starting with Burroughs' Espol and continuing through languages like Mesa and Ada, compiler writers have assumed perfect hardware and contended that they can provide good isolation through static compile-time type checking. In contrast, operating systems designers have advocated run-time checking combined with the process as the unit of protection and failure.
>
> Although compiler checking and exception handling provided by programming languages are real assets, history seems to have favored the run-time checks plus the process approach to fault-containment. It has the virtue of simplicity—if a process or its processor misbehaves, stop it. The process provides a clean unit of modularity, service, fault containment and failure.
>
> Fault containment through fail-fast software modules: The process achieves fault containment by sharing no state with other processes; its only contact with other processes is via messages carried by a kernel message system.

Why is OOP popular?

Given all the problems with OOP, we can reasonably ask why it became popular. **I believe Edsger W. Dijkstra gets at the heart of the matter when he writes**:

> Industry suffers from the managerial dogma that for the sake of stability and continuity, the company should be independent of the competence of individual employees. Hence industry rejects any methodological proposal that can be viewed as making intellectual demands on its work force. Since in the US the influence of industry is more pervasive than elsewhere, the above dogma hurts American computing science most. The moral of this sad part of the story is that as long as computing science is not allowed to save the computer industry, we had better see to it that the computer industry does not kill computing science.

Joe Armstrong offered this theory about why OOP became popular:

> Reason 1 — It was thought to be easy to learn.
>
> Reason 2 — It was thought to make code reuse easier.
>
> Reason 3 — It was hyped.
>
> Reason 4 — It created a new software industry.
>
> I see no evidence of 1 and 2. Reasons 3 and 4 seem to be the driving force behind the technology. If a language technology 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.
>
> This is is the real driving force behind OOPs.

Above, I have, with some humor, suggested that proponents of OOP tend to indulge the No True Scotsman fallacy when arguing for OOP. On a more serious note, some proponents of OOP might try to defend themselves by suggesting that we are facing issues of commensurability: proponents of the pure-logic paradigm, or the functional paradigm, or the OOP paradigm, talk past each other because we can not understand the axioms on which each other's arguments rest. **As Wikipedia defines commensurability**:

> Commensurability is a concept, in philosophy of science, whereby scientific theories are commensurable if scientists can discuss them in terms permitting direct comparison of theories to determine which theory is truer. On the other hand, theories are incommensurable if they are embedded in starkly contrasting conceptual frameworks whose languages lack sufficiently overlapping meanings to permit scientists to directly compare the theories or to cite empirical evidence favoring one theory over the other. Discussed by Ludwik Fleck in the 1930s, and popularized by Thomas Kuhn in the 1960s, the problem of incommensurability results in scientists talking past each other, as it were, while comparison of theories is muddled by confusions about terms' contexts and consequences.

I am willing to believe that this issue explains some of the disconnect between some of the more thoughtful proponents of the different styles. And yet, there has to be something more going on, since the empirical evidence is so overwhelmingly against OOP. Proponents of OOP are arguing against reality itself, and they continue to do so, year after year, with an inflexibility that must have either non-rational or cynically financial sources. I've come to believe that OOP is now a "Zombie Idea". **I am borrowing the phrase from the economist Paul Krugman**:

> Zombie ideas — a phrase I originally saw in the context of myths about Canadian health care — are policy ideas that keep being killed by evidence, but nonetheless shamble relentlessly forward, essentially because they suit a political agenda.

And there is an explicitly political idea that drove OOP to its peak in the 1990s: the idea of outsourcing. The idea of outsourcing software development rested on some assumptions about how software development should work, in particular the idea of the “genius” architect, backed by an army of morons who act as secretaries, taking dictation. OOP was the software equivalent of a trend that became common in manufacturing during the 1980s: design should stay in the USA while actual production should be sent to a 3rd World country. Working with UML diagrams, writing code could be reduced to mere grunt work, whereas the design of software could be handled by visionaries, possessed with epic imaginations, who could specify an OO hierarchy which could then be sent to India for a vast team to actually type out. And the teams in India (or Vietnam, or Romania, etc) were never trusted, they were assumed to be idiots, and so, for a moment, there was a strong market demand for a language that treated programmers like idiots, and so the stage was set for the emergence of Java. **Facundoolano sums up the distrst that Java has for programmers**:

> Quoting Eckel: Java treats programmers like they are stupid, Python doesn’t.
>
> Java design and libraries consistently make a huge effort making it difficult for a programmer to do bad things. If a feature was a potential means for a dumb programmer to make bad code, they would take away the feature altogether. And what’s worse, at times they even used it in the language implementation but prohibited the programmer to do so. Paul Graham remarkably pointed this out as a potential flaw of Java before actually trying the language:
>
> Like the creators of sitcoms or junk food or package tours, Java’s designers were consciously designing a product for people not as smart as them.
>
> But this approach is an illusion; no matter how hard you try, you can’t always keep bad programmers from writing bad programs. Java is a lousy replacement to the learning of algorithms and good programming practices.
>
> The implications of this way of thinking are not limited to language features. The design decisions of the underlying language have a great effect on the programmer’s idiosyncrasy. It’s common to see Java designs and patterns that make an unnecessary effort in prohibiting potential bad uses of the resulting code, again not trusting the programmer’s judgment, wasting time and putting together very rigid structures.
>
> The bottom line is that by making it hard for stupid programmers to do bad stuff, Java really gets in the way of smart programmers trying to make good programs.

The hype in favor of OOP built up during the 1980s and early 90s, but I would say the zenith of the idea was from 1995 to 2005. After 2005, very slowly, a reaction against OOP developed. This reaction initially showed 3 tendencies:

1.) a movement away from static typing and toward dynamic typing

2.) a movement away from compiled languages and toward script languages

3.) a movement away from heavy-weight frameworks and toward lighter frameworks

And then eventually a 4th tendency appeared:

4.) a movement away from OOP and toward the “functional” paradigm

Back in December of 2005, Bruce Eckel managed to catch the exact moment of change, in his essay “**The departure of the hyper-enthusiasts**“:

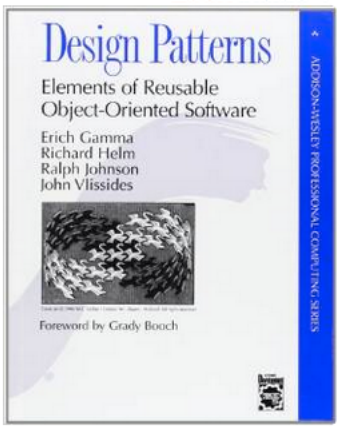
> The Java hyper-enthusiasts have left the building, leaving a significant contingent of Java programmers behind, blinking in the bright lights without the constant drumbeat of boosterism.
>
> But the majority of programmers, who have been relatively quiet all this time, always knew that Java is a combination of strengths and weaknesses. These folks are not left with any feelings of surprise, but instead they welcome the silence, because it’s easier to think and work.
>
> Where did the hyper-enthusiasts go? To Ruby, apparently. This is chronicled in Bruce Tate’s book “Beyond Java,” which should probably be titled “Why Ruby is Better than Java.” …In many places he plays fast and loose, and almost at the end of the book he declares that he doesn’t have time to learn these other languages in any depth — although he has no trouble condemning the same languages in his rush to Ruby. Such a statement should be in the first paragraph of the book: “I’ve decided that I love Ruby, so I will condemn other languages without fully understanding them”.

Please note the bitterness of Eckel’s last remark: this bitterness has been the hallmark of each of the debates that the tech industry has had over the merits of OOP. Someone outside the industry might wonder why emotions run so high, regarding a technical subject, but those of us in the industry know the answer: learning a language, and its eco-system, costs us 1 or 2 years, it is therefore a large

investment, and we all tend to react with the emotions of people who are facing the devaluing of one of our most important assets.

The great saddness of OOP is the waste of brilliant minds

I feel very sad when I consider how many brilliant minds have wasted countless hours trying to find solutions to problems that only exist because of the use of OOP. I think about this every time I read a debate about Design Patterns. Written in 1994, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, the book **Design Patterns: Elements of Reusable Object-Oriented Software** established Design Patterns as one of the main ways that people talk about OOP.



Jeff Atwood described at least 2 problems with the concept of Design Patterns:

> 1. Design patterns are a form of complexity. As with all complexity, I'd rather see developers focus on simpler solutions before going straight to a complex recipe of design patterns.
>
> 2. If you find yourself frequently writing a bunch of boilerplate design pattern code to deal with a "recurring design problem", that's not good engineering– it's a sign that your language is fundamentally broken.

There were many people who were ready to jump to the defense of Design Patterns:

> Steve Rowe agrees that the patterns should be used as examples of good design and principles to apply and not as a reference book but he says that Jeff is off the mark because he attacks the concept instead of the way where the blame is on the people who apply them wrongly. He concludes that patterns should be treated as examples for good design not as dogma:
>
> *Design patterns are very useful when we study how they work so we can create similar patterns. They are bad when we try to copy them directly. If one reads the Gang of Four, he will realize that the authors often give several examples of each pattern and they're all slightly different. One might also notice that there is a lot of talk about the OO concepts that lead to the patterns.*

But the point is, the need for Design Patterns suggests a deficiency of abstraction in the language itself:

> Aristotle Pagaltzis left a comment on Cedric's blog and rationalized Mark's critisism Dominus says that design patterns are a sign of a deficiency of a language for the purpose that the design pattern addresses. In other words, the Visitor pattern used in Java points to the fact that Java is deficient in terms of list processing: the `map` and `filter` constructs need to be emulated with lengthy OO incantations.
>
> He is _not_ saying that the use of design patterns _as such_ is bad. He is saying they are a sign of a deficiency.

If the language allows sufficiently powerful forms of abstraction, then there is no need for Design Patterns. And OOP languages fail to offer the needed levels of abstraction, which leads to an excessive thirst for more abstraction. When a boat sinks, and people are lost at sea in a raft, the lack of water often drives people to drink the salt water of the sea, which, rather than helping, greatly worsens their thirst — and OOP programmers are exactly like this, desperate for abstraction, and using OOP to try to acheive abstraction, and yet OOP only makes the situation worse, leading to excess. **Mike Gerwitz is charitable enough to suggest that OOP is not inherently terrible, only its excesses are**:

> The problem is that, with the excitement and misunderstandings that surround "good" object-oriented design, designers are eager to over-abstract their implementations (I have been guilty of the same thing). Object oriented programming is often taught to novice CS students (often with the reign of Java in schools)—teaching practices that can be good principles when properly applied and in moderation—which I have also seen contribute to such madness.

> Abstractions are highly important, but only when necessary and when they lead to more concise representations of the problem than would otherwise occur (note that some problems are inherently complicated and, as such, a concise representation may not seem concise). I'm a strong advocate of DSLs when abstractions begin to get in the way and increase the verbosity of the code (languages with strong macro systems like lisp help eliminate the need for DSLs written from scratch) —design patterns exist because of deficiencies in the language: They are "patterns" of code commonly used to achieve a certain effect.

The point is, OOP induces those excesses, so suggesting that OOP is good and only the excesses of OOP are bad is a bit like saying that crystal meth is good and only the excesses of crystal meth are bad.

Another area where I often see brilliant minds wasting countless hours is in discussions about how to avoid having utility code. **Again, in a sarcastic post, Miško Hevery (explaining how to write untestable code) sums up the official OOP ideology about utility code**:

> Utils, Utils, Utils! – Code smell? No way – code perfume! Litter about as many util and helper classes as you wish. These folks are helpful, and when you stick them off somewhere, someone else can use them too. That's code reuse, and good for everyone, right? Be forewarned, the OO-police will say that functionality belongs in some object, as that object's responsibility. Forget it, you're way to pragmatic to break things down like they want. You've got a product to ship after all!

Conversations about utility code spring up everywhere that OOP languages are used. For instance, in the world of Ruby On Rails, there is an ongoing debate about what to do with fat models. **The Code Climate blog has this to say about utility code**:

> As you add more intrinsic complexity (read: features!) to your application, the goal is to spread it across a coordinated set of small, encapsulated objects (and, at a higher level, modules) just as you might spread cake batter across the bottom of a pan. Fat models are like the big clumps you get when you first pour the batter in. Refactor to break them down and spread out the logic evenly. Repeat this process and you'll end up with a set of simple objects with well defined interfaces working together in a veritable symphony.
>
> You may be thinking:
>
> "But Rails makes it really hard to do OOP right!"
>
> I used to believe that too. But after some exploration and practice, I realized that Rails (the framework) doesn't impede OOP all that much. It's the Rails "conventions" that don't scale, or, more specifically, the lack of conventions for managing complexity beyond what the Active Record pattern can elegantly handle. Fortunately, we can apply OO-based principles and best practices where Rails is lacking.
>
> Don't Extract Mixins from Fat Models
>
> Let's get this out of the way. I discourage pulling sets of methods out of a large ActiveRecord class into "concerns", or modules that are then mixed in to only one model. I once heard someone say:
>
> "Any application with an app/concerns directory is concerning."
>
> And I agree. Prefer composition to inheritance. Using mixins like this is akin to "cleaning" a messy room by dumping the clutter into six separate junk drawers and slamming them shut. Sure, it looks cleaner at the surface, but the junk drawers actually make it harder to identify and implement the decompositions and extractions necessary to clarify the domain model.

In this case "concerns" is synonymous with utility code. The article then makes 7 suggestions:

1. Extract Value Objects

2. Extract Service Objects

3. Extract Form Objects

4. Extract Query Objects

5. Introduce View Objects

6. Extract Policy Objects

7. Extract Decorators

I have 2 problems with this list of 7 items:

1.) all 7 ideas are really just utility code in disguise

2.) we have to waste time having this conversation because of OOP

There is nothing wrong with utility code. In some sense, all code is utility code. We do want to organize our code in reasonable ways. We should probably group related functions together. When writing Clojure code, I tend to put all my functions for creating HTML in one namespace, and all my functions for querying the database in another namespace. But I don't have to worry organizational issues nearly as much as I would have to in an OOP language, at least partly because I don't have to mix state, data-types hierachies, and behavior together. In OOP, the organization of the class hierarchy can have dire consequences. Wondering if a particular bit of state belongs in Class A or Class B is a serious issue. In functional languages, this issue carries less weight: **people still discuss how code should be organized**, but there is no sense that utility code is bad, and less philosophical worries about how impure one's architecture is.

Slava Akhmechet went so far as to suggest that design patterns are useless in functional languages:

> Most people I've met have read the Design Patterns book by the Gang of Four. Any self respecting programmer will tell you that the book is language agnostic and the patterns apply to software engineering in general, regardless of which language you use. This is a noble claim. Unfortunately it is far removed from the truth.
>
> Functional languages are extremely expressive. In a functional language one does not need design patterns because the language is likely so high level, you end up programming in concepts that eliminate design patterns all together.

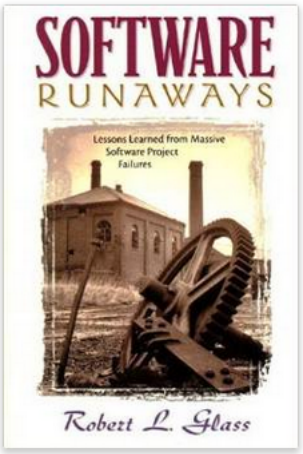
"jalf" responded on StackOverflow:

> I don't think it should be particularly controversial to say that design patterns in general only exist to patch up shortcomings in the language. And if another language can solve the same problem trivially, that other language won't have need of a design pattern for it. Users of that language may not even be aware that the problem exists, because, well, it's not a problem in that language.
>
> *The main features of functional programming include functions as first-class values, currying, immutable values, etc. It doesn't seem obvious to me that OO design patterns are approximating any of those features.*
>
> What is the command pattern, if not an approximation of first-class functions? :) In a FP language, you'd simply pass a function as the argument to another function. In an OOP language, you have to wrap up the function in a class, which you can instantiate and then pass that object to the other function. The effect is the same, but in OOP it's called a design pattern, and it takes a whole lot more code. And what is the abstract factory pattern, if not currying? Pass parameters to a function a bit at a time, to configure what kind of value it spits out when you finally call it.
>
> So yes, several GoF design patterns are rendered redundant in FP languages, because more powerful and easier to use alternatives exist.

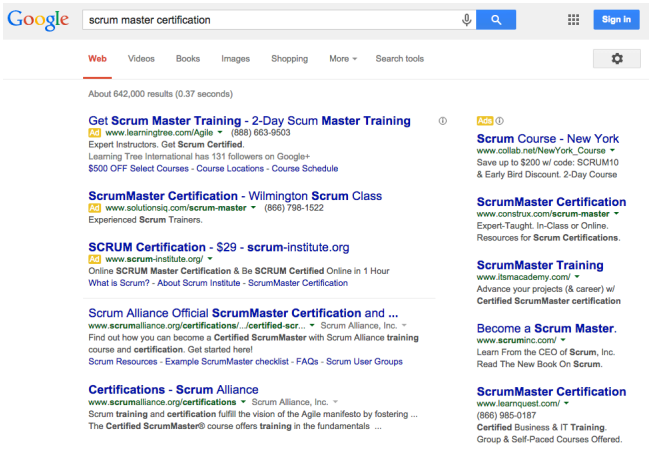
As software developers, we will forever be discussing certain issues that are fundamental to the craft of software. But we should try, as much as possible, to avoid conversations that waste our precious hours on problems that only exist because of the weaknesses of the languages that we are using. The correct answer to conversations about the Factory Pattern or the Command Pattern or the Singleton Pattern is to find a better programming language.

OOP is expensive

Estimating the duration and costs of software projects is difficult. Robert L. Glass has written several books on why this is so. Check out his book "**Software Runaways: Monumental Software Disasters**". Because software disasters are so expensive, businesses keep looking for some silver bullet that will bring predictability and reliability to software projects. OOP was promoted as the silver bullet, but it never quite worked out.



As the years went by, and OOP failed to deliver on its promise, a vast secondary industry of advisors grew up around it. You see, OOP could be a silver bullet, if only you did it right, and these people, with particular credentials, just happen to know what you needed to do to make OOP the silver bullet that it really wanted to be. Search Google for "scrum master certified" and you get a sense of what a huge industry the "We can save OOP" industry has become:



Of course, methodologies such as Scrum and Agile and Lean are not specific to OOP, but, realistically, the use of these methodologies and the use of OOP has so far had near perfect overlap.

To be clear, I think the right methods can offer real benefits, and I think training can help spread "best practice". And if ever you find one of those rare project managers who knows how to unleash the full potential of your team, **grapple them unto thy soul with hoops of steel**. True leadership is rare, and almost priceless. But many of the popular methodologies arose specifically because of the pain businesses have felt regarding software development, and because of the failure of OOP to deliver on its promises.

Thankfully, there has been a backlash against some of the bureaucracy that has grown up around these supposedly lightweight methodologies. **Dave Thomas wrote**:

> Thirteen years ago, I was among seventeen middle-aged white guys who gathered at Snowbird, Utah. We were there because we shared common beliefs about developing software, and we wondered if there was a way to describe what we believed.
>
> It took less than a day to come up with a short list of values. We published those values, along with a list of practices, as the Manifesto for Agile Software Development:
>
> Individuals and Interactions over Processes and Tools
>
> Working Software over Comprehensive Documentation
>
> Customer Collaboration over Contract Negotiation, and
>
> Responding to Change over Following a Plan
>
> …The word "agile" has been subverted to the point where it is effectively meaningless, and what passes for an agile community seems to be largely an arena for consultants and vendors to hawk services and products.
>
> …Once the Manifesto became popular, the word agile became a magnet for anyone with points to espouse, hours to bill, or products to sell. It became a marketing term, coopted to improve sales in the same way that words such as eco and natural are. A word that is abused in this way becomes useless — it stops having meaning as it transitions into a brand.
>
> This hurts everyone, but I'm particularly sensitive to the damage it does to developers. It isn't easy writing code, and developers naturally are on the lookout for things that will help them deliver value more effectively. I still firmly believe that sticking to the values and practices of the manifesto will help them in this endeavor.

Richard Bishop writes what he calls the "angry developer version" of the same idea:

> I've been in a three day long sprint planning meeting for a six week "sprint." I've seen user story cards with nothing but "Make dashboard better" written on them by Certified Scrum Masters. It's pathetic. Even more pathetic is that I accepted their job offer, but I didn't know any better. They told me they were Agile, after all.

> …Every time I talk with a so-called Agile company about how they work I get a laundry list of SaaS Web applications. Trello, Basecamp, JIRA, Pivotal — none of these tools existed when the Agile Manifesto was carved into Kent Beck's chest while he was being forcefully held to an Altair 8080 by the other fifteen Agile Manifesto founders (it's still unclear which founder did the carving).
>
> These tools aren't the solution. Agile doesn't need a solution, it is the solution, or rather a guiding light to the solution for your particular team. These tools are masquerading around pretending to be the solution to people. People that haven't learned what Agile is all about. These people just hit the Wikipedia page for Agile after seeing it mentioned in a job post because their degree in Communications couldn't get them any other gig.

Don't get me wrong, I'm not suggesting that OOP is responsible for companies with hours to bill and products to sell. There were over-priced consultants before OOP, and there will be over-priced consultants after OOP. But such consultants, to the extent that they act as parasites, extracting money without adding much value, are facillitated by the whatever weaknesses exist in the technologies we use. Their ideal customer is the one who is in pain. I've heard it said "death is what you wish for when you are being tortured" and likewise, a silver bullet is what you wish for you when your expensive new technology stack is failing. We owe it to ourselves, as developers, to seek technologies that make our jobs as easy as possible. Programming is hard enough, there is no need to weigh ourselves down with the extra burden of unneeded ceremony.

What does it mean to say that a language is low ceremony? That is a bit of a subjective call, and even for a particular person, one's opinion will change over time. I recall when I first discovered Ruby On Rails, back in 2005. It had tools that would auto-generate my model classes for me, which I thought was very cool. More so, the model classes were simple. If I had database tables for "users", "products", "sales" and "purchases", the tools would probably generate 4 files for me, that would look like this:

```
class User
end

class Product
end

class Sale
end

class Purchase
end
```

At the time, I thought, this is fantastic! I didn't have to write getters and setters, that was implied! My limited experience with Java had left me with a bad taste in my mouth — Java was verbose! I had to write every function to get and set a variable. How tedious! (Even PHP, a script language, forced me to write getters and setters, just like Java.) Ruby saved us from all that.
But the present constantly changes the meaning of the past. The years go by and we see things differently. I spent 2 years away from Ruby, working with Clojure, and then I came back to Ruby and I was astonished to realize that it now seemed high-ceremony to me. Look at those 4 empty classes! That's 4 files, and 8 lines of code, that are mere boilerplate — they don't help me with my task, I just have to write it so that when method_missing is called it will have some context to work with. What seemed low-ceremony to me, in 2005, strikes me as high-ceremony in 2014.

Have you ever seen a dog chase its tail? No matter how fast the dog goes, it can not catch up with its tail, because the tail always recedes at the same speed that the dog moves. OOP programmers are just like that, trying to get the amount of abstraction needed to catch the enlightenment promised by OOP (or, to put that less charitably, trying to escape the pain of OOP). But they can never get there, since the language forces them to declare data-types, behavior and state all in one spot. The problem is inherent to the language. The only real solution is to use a non-OOP language. But OOP programmers instead try to solve the problem inside the language. They keep piling layer on top of layer of abstraction, hoping that they will eventually have enough abstraction that they can get free of the pain they are in. **Regarding this quest for abstraction, Joel Spolsky has written a spoof, about building a spice rack, which we all find funny exactly because it comes so close to the truth**:

> "A hammer?" he asks. "Nobody really buys hammers anymore. They're kind of old fashioned."
>
> Surprised at this development, I ask him why.
>
> "Well, the problem with hammers is that there are so many different kinds. Sledge hammers, claw hammers, ball-peen hammers. What if you bought one kind of hammer and then realized that you needed a different kind of hammer later? You'd have to buy a separate hammer for your next task. As it turns out, most people really want a single hammer that can handle all of the different kinds of hammering tasks you might encounter in your life."
>
> "Hmmmmmm. Well, I suppose that sounds all right. Can you show me where to find a Universal Hammer."
>
> "No, we don't sell those anymore. They're pretty obsolete."

“Really? I thought you just said that the Universal Hammer was the wave of the future.”

“As it turns out, if you make only one kind of hammer, capable of performing all the same tasks as all those different kinds of hammers, then it isn’t very good at any of them. Driving a nail with a sledgehammer isn’t very effective. And, if you want to kill your ex-girlfriend, there’s really no substitute for a ball-peen hammer.”

“That’s true. So, if nobody buys Universal Hammers anymore, and if you’re no longer selling all those old-fashioned kinds of hammers, what kinds of hammers do you sell?”

“Actually, we don’t sell hammers at all.”

“So…”

“According to our research, what people really needed wasn’t a Universal Hammer after all. It’s always better to have the right kind of hammer for the job. So, we started selling hammer factories, capable of producing whatever kind of hammers you might be interested in using. All you need to do is staff the hammer factory with workers, activate the machinery, buy the raw materials, pay the utility bills, and PRESTO…you’ll have *exactly* the kind of hammer you need in no time flat.”

“But I don’t really want to buy a hammer factory…”

“That’s good. Because we don’t sell them anymore.”

“But I thought you just said…”

“We discovered that most people don’t actually need an entire hammer factory. Some people, for example, will never need a ball-peen hammer. (Maybe they’ve never had ex-girlfriends. Or maybe they killed them with icepicks instead.) So there’s no point in someone buying a hammer factory that can produce every kind of hammer under the sun.”

“Yeah, that makes a lot of sense.”

“So, instead, we started selling schematic diagrams for hammer factories, enabling our clients to build their own hammer factories, custom engineered to manufacture only the kinds of hammers that they would actually need.”

“Let me guess. You don’t sell those anymore.”

“Nope. Sure don’t. As it turns out, people don’t want to build an entire factory just to manufacture a couple of hammers. Leave the factory-building up to the factory-building experts, that’s what I always say!!”

“And I would agree with you there.”

“Yup. So we stopped selling those schematics and started selling hammer-factory-building factories. Each hammer factory factory is built for you by the top experts in the hammer factory factory business, so you don’t need to worry about all the details that go into building a factory. Yet you still get all the benefits of having your own customized hammer factory, churning out your own customized hammers, according to your own specific hammer designs.”

“Well, that doesn’t really…”

“I know what you’re going to say!! …and we don’t sell those anymore either. For some reason, not many people were buying the hammer factory factories, so we came up with a new solution to address the problem.”

“Uh huh.”

“When we stepped back and looked at the global tool infrastructure, we determined that people were frustrated with having to manage and operate a hammer factory factory, as well as the hammer factory that it produced. That kind of overhead can get pretty cumbersome when you deal with the likely scenario of also operating a tape measure factory factory, a saw factory factory, and a level factory factory, not to mention a lumber manufacturing conglomerate holding company. When we really looked at the situation, we determined that that’s just too complex for someone who really just wants to build a spice rack.”

“Yeah, no kidding.”

“So this week, we’re introducing a general-purpose tool-building factory factory factory, so that all of your different tool factory factories can be produced by a single, unified factory. The factory factory factory will produce only the tool factory factories that you actually need, and each of those factory factories will produce a single factory based on your custom tool specifications. The final set of tools that emerge from this

> process will be the ideal tools for your particular project. You'll have *exactly* the hammer you need, and exactly the right tape measure for your task, all at the press of a button (though you may also have to deploy a few *configuration files* to make it all work according to your expectations)."
>
> "So you don't have any hammers? None at all?"

All of this complexity is expensive: the more bloated the code becomes, the more work is needed to maintain it. Functional languages such as Haskell, Erlang and Clojure offer powerful approaches to the problems that software developers have always faced. All of the so-called strengths of OOP can be found in these languages. If you are a fan of strict data-typing, then use Haskell. If you are a fan of the Actor model, use Erlang. If you'd like to work in a flexible, dynamic language where Immutable is the default, use Clojure. Or, on a completely different level, if you need to write a simple 20 line cron script, use PHP, without any OOP features. But don't ever use OOP. OOP does not have any unique strengths. It brings a great weight of complexity for no benefit.

OOP was the great "believable promise" that inspired much of the tech industry for much of the last 30 years. For awhile, people hoped it was the silver bullet that could solve the problems of software development. We now know that it is an experiment that failed. It is time to move on. It is time that we, as a community, admit that this idea has failed us, and we must give up on it.

OOP is an expensive disaster which must end.

Source