# Does functional programming replace GoF design patterns?

Ask Question

Since I started learning F# and OCaml last year, I've read a huge number of articles which insist that design patterns (especially in Java) are workarounds for the missing features in imperative languages. One article I found makes a fairly strong claim:

> 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.**

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.

Additionally, in functional languages which support OOP (such as F# and OCaml), it seems obvious to me that programmers using these languages would use the same design patterns found available to every other OOP language. In fact, right now I use F# and OCaml every day, and there are no striking differences between the patterns I use in these languages vs. the patterns I use when I write in Java.

Is there any truth to the claim that functional programming eliminates the need for OOP design patterns? If so, could you post or link to an example of a typical OOP design pattern and its functional equivalent?

design-patterns oop functional-programming

edited Apr 22 '17 at 0:05
Peter Mortensen
**12.7k** 19 82 110

asked Nov 29 '08 at 20:08
Juliet
**58k** 38 177 218

14 You might look at the article by Steve Yegge (steve-yegge.blogspot.com/2006/03/…) – Ralph Mar 28 '13 at 13:29

17 "the book is language agnostic and the patterns apply to software engineering in general" - it should be noted that the book disagrees with this claim, in the sense that some languages don't need to express certain things like design patterns: "Our patterns assume Smalltalk/C++-level language features, and that choice determines what can and cannot be implemented easily [...] CLOS has multi-methods, for example, which lessen the need for a pattern such as Visitor (Page 331)." (page 4) – Guildenstern Oct 27 '14 at 16:01

5 Also keep in mind that many design patterns aren't even necessary in sufficiently high level imperative languages. – R. Barzell Feb 25 '15 at 20:07

3 @R.Barzell What are those "sufficiently high level imperative languages"? Thanks. – cibercitizen1 Oct 15 '15 at 8:39

5 @cibercitizen1 duck-typed languages with support for higher order functions and anonymous functions. These features supply much of the power that a lot of design patterns were meant to provide. – R. Barzell Oct 15 '15 at 12:50

## 23 Answers

The blog post you quoted overstates its claim a bit. FP doesn't *eliminate* the need for design patterns. The term "design patterns" just isn't widely used to describe the same thing in FP languages. But they exist. Functional languages have plenty of best practice rules of the form "when you encounter problem X, use code that looks like Y", which is basically what a design pattern is.

However, it's correct that most OOP-specific design patterns are pretty much irrelevant in functional languages.

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.

Here is what the Gang of Four has to say about this issue:

> The choice of programming language is important because it influences one's point of view. Our patterns assume Smalltalk/C++-level language features, and that choice determines what can and cannot be implemented easily. If we assumed procedural languages, we might have included design patterns called "Inheritance", "Encapsulation," and "Polymorphism". Similarly, some of our patterns are supported directly by the less common object-oriented languages. CLOS has multi-methods, for example, which lessen the need for a pattern such as Visitor. In fact, there are enough differences

> between Smalltalk and C++ to mean that some patterns can be expressed more easily in one language than the other. (See Iterator for example.)

(The above is a quote from the Introduction to the Design Patterns book, page 4, paragraph 3)

> 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.

But of course there are still design patterns which are *not* solved by FP languages. What is the FP equivalent of a singleton? (Disregarding for a moment that singletons are generally a terrible pattern to use)

And it works both ways too. As I said, FP has its design patterns too, people just don't usually think of them as such.

But you may have run across monads. What are they, if not a design pattern for "dealing with global state"? That's a problem that's so simple in OOP languages that no equivalent design pattern exists there.

We don't need a design pattern for "increment a static variable", or "read from that socket", because it's just what you *do*.

In (pure) functional languages, side effects and mutable state are impossible, unless you work around it with the monad "design pattern", or any of the other methods for allowing the same thing.

> Additionally, in functional languages which support OOP (such as F# and OCaml), it seems obvious to me that programmers using these languages would use the same design patterns found available to every other OOP language. In fact, right now I use F# and OCaml everyday, and there are no striking differences between the patterns I use in these languages vs the patterns I use when I write in Java.

Perhaps because you're still thinking imperatively? A lot of people, after dealing with imperative languages all their lives, have a hard time giving up on that habit when they try a functional language. (I've seen some pretty funny attempts at F#, where literally *every* function was just a string of 'let' statements, basically as if you'd taken a C program, and replaced all semicolons with 'let'. :))

But another possibility might be that you just haven't realized that you're solving problems trivially which would require design patterns in an OOP language.

When you use currying, or pass a function as an argument to another, stop and think about how you'd do that in an OOP language.

> Is there any truth to the claim that functional programming eliminates the need for OOP design patterns?

Yep. :) When you work in a FP language, you no longer need the OOP-specific design patterns. But you still need some general design patterns, like MVC or other non-OOP specific stuff, and you need a couple of new FP-specific "design patterns" instead. All languages have their shortcomings, and design patterns are usually how we work around them.

Anyway, you may find it interesting to try your hand at "cleaner" FP languages, like ML (my personal favorite, at least for learning purposes), or Haskell, where you don't have the OOP crutch to fall back on when you're faced with something new.

As expected, a few people objected to my definition of design patterns as "patching up shortcomings in a language", so here's my justification: As already said, most design patterns are specific to one programming paradigm, or sometimes even one specific language. Often, they solve problems that only *exist* in that paradigm (See monads for FP, or abstract factories for OOP). Why doesn't the abstract factory pattern exist in FP? Because the problem it tries to solve does not exist there. So, if a problem exists in OOP languages, which does not exist in FP languages, then clearly that is a shortcoming of OOP languages. The problem can be solved, but your language does not do so, but requires a bunch of boilerplate code from you to work around it. Ideally, we'd like our programming language to magically make *all* problems go away. Any problem that is still there is in principle a shortcoming of the language. ;)

edited Nov 24 '15 at 6:18
Robert Harvey ♦
**144k** 31 259 395

answered Nov 29 '08 at 23:06
jalf
**201k** 41 286 512

67 Design patterns describe general solutions to basic problems. But that's also what programming languages and

67 Design patterns describe general solutions to basic problems. But that's also what programming languages and platforms do. So you use design patterns when the languages and platforms you are using do not suffice. – yfeldblum Nov 30 '08 at 8:41

123 S.Lott: They describe solutions to problems which exist in a given language, yes. There is no Command design pattern in FP languages, because the problem it tries to solve does not exist. Which means that they solve problems that the language itself can't solve. That is, shortcomings in the language – jalf Nov 30 '08 at 12:42

31 The monad is a mathematical concept, and you're stretching it with your classification. Sure, you can view functions, monoids, monads, matrices or other mathematical concepts as design patterns, but those are more like algorithms and data structures ... fundamental concepts, language independent. – Alexandru Nedelcu Nov 30 '08 at 16:25

33 Sure, monads are a math concept, but they are *also* a pattern. The "FP pattern" of monads is somewhat distinct from the math concept of monads. The former is a pattern used to get around certain "limitations" in pure FP languages. The latter is a universal mathematical concept. – jalf Dec 3 '08 at 18:53

59 Note that monads in Haskell are used for other things than mutable state, for example for exceptions, continuations, list comprehensions, parsing, asynchronous programming and so on. But all these application of monads could probably be called patterns. – JacquesB Jan 2 '09 at 14:33

> Is there any truth to the claim that functional programming eliminates the need for OOP design patterns?

Functional programming is not the same as object-oriented programming. Object-oriented design patterns don't apply to functional programming. Instead, you have functional programming design patterns.

For functional programming, you won't read the OO design pattern books, you'll read other books on FP design patterns.

> language agnostic

Not totally. Only language-agnostic with respect to OO languages. The design patterns don't apply to procedural languages at all. They barely make sense in a relational database design context. They don't apply when designing a spreadsheet.

> a typical OOP design pattern and its functional equivalent?

The above shouldn't exist. That's like asking for a piece of procedural code rewritten as OO code. Ummm... If I translate the original Fortran (or C) into Java, I haven't done anything more than translate it. If I totally rewrite it into an OO paradigm, it will no longer look anything like the original Fortran or C -- it will be unrecognizable.

There's no simple mapping from OO Design to Functional Design. They're very different ways of looking at the problem.

Functional programming (like *all* styles of programming) has design patterns. Relational databases have design patterns, OO has design patterns, procedural programming has design patterns. Everything has design patterns, even the architecture of buildings.

Design patterns -- as a concept -- are a timeless way of building, irrespective of technology or problem domain. However, specific design patterns apply to specific problem domains and technologies.

Everyone who thinks about what they're doing will uncover design patterns.

edited Aug 5 '09 at 21:40
James McMahon
**28.4k** 56 169 253

answered Nov 29 '08 at 20:15
S.Lott
**305k** 64 425 707

That's pretty much what I expected, although I think there is some truth in the claim that functional programming can simplify existing patterns. For example, I find that I no longer need to pass around single-method interfaces when I can pass functions as values instead. – Juliet Nov 29 '08 at 20:35

9 MVC isn't OO design. It's architectural design -- that pattern applies pretty widely. – S.Lott Nov 29 '08 at 21:23

1 @Princess: functional programming isn't necessarily simpler. In your example, yes. For other things, the jury's still out. But you have discarded a Java OO design pattern and adopted a FP design pattern. – S.Lott Nov 29 '08 at 21:26

6 Java 8 will include closures aka anonymous functions aka lambda expressions. This will make the command design pattern obsolete for Java. This is an example of language shortcoming, no? They added a missing feature and now you don't need the design pattern. – Todd Chaffee Sep 29 '12 at 17:19

1 +1 for the closing sentence. Design patterns are meant to simplify programming and make the resulting programs more efficient, at what they are intended to do. – Sorter Jul 15 '13 at 16:55

Brian's comments on the tight linkage between language and pattern is to the point,

The missing part of this discussion is the concept of idiom. Coplien's book, "Advanced C++" was a huge influence here. Long before he discovered Christopher Alexander and the *Column Without a Name* (and you can't talk sensibly about patterns without reading Alexander either), he talked about the importance of mastering idiom in truly learning a language. He used string copy in C as an example, while(*from++ = *to++); You can see this as a bandaid for a missing language feature (or library feature), but what really matters about it is that it's a larger unit of thought, or of expression, than any of its parts.

That is what patterns, and languages, are trying to do, to allow us to express our intentions more succinctly. The richer the units of thought the more complex the thoughts you can express. Having a rich, shared vocabulary at a range of scales - from system architecture down to bit twiddling - allows us to have more intelligent conversations, and thoughts about what we should be doing.

We can also, as individuals, learn. Which is the entire point of the exercise. We each can understand and use things we would never be able to think of ourselves. Languages, frameworks, libraries, patterns, idioms and so on all have their place in sharing the intellectual wealth.

answered Jan 1 '09 at 20:59

grahamsw

7 Thank you! *this* is what patterns are about—"conceptual chunking" to lower cognitive burden. – Randall Schulz Nov 24 '09 at 18:07

And Functional Monads definitely belong in this discussion. – Greg Nov 10 '12 at 3:42

@RandallSchulz: language features (and their idiomatic use, of course) would also fit well into the category of "conceptual chunking to lower cognitive burden." – Roy Tinker Jul 25 '13 at 18:20

---

The GOF book explicitly ties itself to OOP - the title is Design Patterns - Elements of Reusable *Object-Oriented* Software (emphasis mine.)

answered Nov 24 '09 at 15:49

bright
**2,318** 1 17 44

Design Patterns in Dynamic Programming by Peter Norvig has thoughtful coverage of this general theme, though about 'dynamic' languages instead of 'functional' (there's overlap).

answered Nov 29 '08 at 20:23
Darius Bacon
**12.7k** 3 43 51

Here's another link, discussing this topic: http://blog.ezyang.com/2010/05/design-patterns-in-haskel/

In his blog post Edward describes all 23 original GoF patterns in terms of Haskell.

answered Oct 19 '12 at 8:15
folone
**5,651** 11 53 105

3 The article doesn't seem to really show design patterns in Haskell, but show how Haskell addresses those needs without said patterns. – Fresheyeball Sep 23 '14 at 22:23

3 @Fresheyball: Depends upon your definition of patterns. Is mapping a function over a list a variant of the Visitor pattern? I have generally thought the answer was "yes." Patterns are supposed to go beyond a particular syntax. The function being applied could be wrapped as an object or passed as a function pointer, but the concept is the same, to me. Do you disagree? – srm Apr 2 '15 at 21:06

When you try to look at this at the level of "design patterns" (in general) and "FP versus OOP", the answers you'll find will be murky at best.

Go a level deeper on both axes, though, and consider *specific design patterns* and *specific language features* and things become clearer.

So, for example, some specific patterns, like **Visitor**, **Strategy**, **Command**, and **Observer** definitely change or disappear when using a language with **algebraic data types and pattern matching**, **closures**, **first class functions**, etc. Some other patterns from the GoF book still 'stick around', though.

In general, I would say that, over time, specific patterns are being eliminated by new (or just rising-in-popularity) language features. This is the natural course of language design; as languages become more high-level, abstractions that could previously only be called out in a book using examples now become applications of a particular language feature or library.

(Aside: here's a recent blog I wrote, which has other links to more discussion on FP and design patterns.)

answered Nov 29 '08 at 23:15

Brian
**104k** 14 200 277

How can you say the visitor pattern "disappears"? Doesn't it just turn from "make a Visitor interface with a bunch Visit methods" into "use union types and pattern matching"? – Gabe Sep 23 '10 at 19:48

14 Yes, but that changed from a *pattern* which is a design idea that you read about in a book and apply to your code, to "just code". That is, "use union types and pattern matching" is just how you normally code stuff in such a language. (Analogy: if no languages had `for` loops and they all just had `while` loops, then "For" might be an iteration pattern. But when `for` is just a construct supported by the language and how people code normally, then it's not a pattern - you don't need a pattern, it's just code, man.) – Brian Sep 23 '10 at 20:05

4 Put another way, a maybe-not-bad litmus test for "is it a pattern" is: present code written this way to a second-year undergraduate student majoring in CS with one year of experience programming in your language. If you show them the code, and they go "that's a clever design", then it's a pattern. If you show them the code, and they go "well, duh!", then it's not a pattern. (And if you show this "Visitor" to anyone who has done ML/F#/Haskell for a year, they will go "well, duh!") – Brian Sep 23 '10 at 20:10

1 Brian: I think we just have different definitions of a "pattern". I consider any identifiable design abstraction to be a *pattern*, while you consider only non-obvious abstractions to be a *pattern*. Just because C# has `foreach` and Haskell has `mapM` doesn't mean that they don't have the Iterator pattern. I see no problem in saying that the Iterator pattern is implemented as the generic interface `IEnumerable<T>` in C# and the typeclass `Traversable` in Haskell. – Gabe Sep 24 '10 at 16:21

It might be that non-obvious patterns are of use to software engineers but all patterns are of use to language designers. I.e. "If you're creating a new language, make sure to include a clean way to express the iterator pattern." Even the obvious patterns are of interest when we start asking the question, "Is there a better syntax for expressing this idea?" After all, that's what lead someone to create foreach. – srm Apr 2 '15 at 21:09

Norvig's presentation alludes to an analysis they did of all the GoF patterns, and they say that 16 of the 23 patterns had simpler implementations in functional languages, or were simply part of the language. So presumably at least seven of them either were a) equally complicated or b) not present in the language. Unfortunately for us, they are not enumerated!

I think it's clear that most of the "creational" or "structural" patterns in GoF are merely tricks to get the primitive type systems in Java or C++ to do what you want. But the rest are worthy of consideration no matter what language you program in.

One might be Prototype; while it is a fundamental notion of JavaScript, it has to be implemented from scratch in other languages.

One of my favorite patterns is the Null Object pattern: represent the absence of something as an object that does an appropriate kind of nothing. This may be easier to model in a functional language. However, the real achievement is the shift in perspective.

edited Feb 19 '10 at 22:41
Peter Mortensen
**12.7k** 19 82 110

answered Nov 30 '08 at 7:11
Neil Kandalgaonkar

2 What an odd analysis to do since the GoF patterns were specifically designed for class-based OOP languages. Seems a bit like analyzing whether pipe wrenches are good for doing electrical work. – munificent Sep 16 '10 at 22:59

@munificent: Not really. Object-orientation provides polymorphism; functional programming generally provides polymorphism. – Marcin Feb 7 '12 at 14:58

@Marcin an OO programmer means something very different by polymorphism than a functional programmer. – AndrewC Apr 10 '13 at 20:45

@AndrewC I disagree. The OO programmer may think they mean something different, but they don't. – Marcin Apr 28 '13 at 19:20

3 @Marcin In my experience, an OO programmer is typically referring to subtype polymorphism (often just using Object), utilising casts to achieve it, or ad-hoc polymorphism (overloading etc). When a functional programmer says polymorphism they mean parametric polymorphism (i.e. works for *any* type of data - Int, function, list), which is perhaps more like OO's generic programming than it is like anything OO programmers usually call polymorphism. – AndrewC Apr 28 '13 at 20:02

I would say that when you have a language like Lisp with its support for macros, then you can build you own domain-specific abstractions, abstractions which often are much better than the general idiom solutions.

edited Apr 22 '17 at 0:09
Peter Mortensen
**12.7k** 19 82 110

answered Dec 16 '08 at 19:44
Anders Rune Jensen
**2,191** 1 31 49

I'm completely lost. To up something with abstractions... What does that mean? – tuinstoel Jan 1 '09 at 21:50

2 You can build domain specific abstractions (even embedded ones) without macros. Macros just let you pretty them up

by adding custom syntax. – Jon Harrop Apr 14 '11 at 22:07

2 You can think of Lisp as a set of Legos for building programming languages - it's a language but it's also a metalanguage. Which means that for any problem domain, you can custom-design a language that doesn't have any obvious deficiencies. It will require some practice, and Kurt Gödel may disagree, but it's worth spending some time with Lisp to see what it brings to the table (hint, macros). – Greg Nov 10 '12 at 3:39

And even the OO Design Pattern solutions are language specific. Design patterns are solutions to common problems that your programming language doesn't solve for you. In Java, the Singleton pattern solves the one-of-something (simplified) problem. In Scala, you have a top level construct called Object in addition to Class. It's lazily instantiated and there is only one. You don't have to use the Singleton pattern to get a Singleton. It's part of the language.

answered Nov 30 '08 at 18:02

Dan Sickles

Patterns are ways of solving similar problems that get seen again and again, and then get described and documented. So no, FP is not going to replace patterns; however, FP might create new patterns, and make some current "best practices" patterns "obsolete".

answered Aug 26 '13 at 2:10

Edwin Buck
**54k** 4 69 103

3 GoP patterns are ways of solving the problem of the limitations of a particular type of programming language getting in your way. For instance "I want to indirect on classes, and tell them to make objects" -> "You can't, but you can make metaclass-like objects called a Factory". "I want multiple dispatch" -> "You can't, but there is labyrinth you can implement called the Visitor Pattern". Etc. None of the patterns make sense if you're not in a OOP language with specific limitations. – Kaz Dec 9 '13 at 18:02

1 I don't know about "none" of them making sense in other languages, but I'll agree that a lot of them don't make sense in other languages. Adapter and Bridge seem to have more multilingual possibilities, decreasing a bit for visitor, and perhaps a bit less for listener. However, patterns across languages are always going to suffer from "how to do language X's operation in language Y" type shoring up of the language's natural boundaries. A perfect example was the Singleton pattern, which is basically, how do I get C globals in OOP? (which I'll answer, you shouldn't). – Edwin Buck Dec 9 '13 at 19:41

1 I second Kaz: Pattern are not "way of solving similar problems that get seen again and again" but "way of solving similar problems that get seen again and again AND have to be rewritten again and again because the language doesn't allow one to write it only once". In other words, if the language allowed the pattern to be extracted/abstracted in library/class/module etc it stops being a pattern but becomes a library/class/module. In FP, it's much much easier to extract/abstract bit of codes to a function, therefore "pattern" are more easily converted in reusable code making them not a pattern. – mb14 Jun 12 '14 at 18:54

1 Your interpretation is welcome, but the GoF book was clear to define a pattern, and if your read the introductory chapters, it says nothing about languages, or weaknesses of the language. Certainly some languages have areas which will make them leverage some patterns more often, but whether you write it ten times (cut and paste) or implement it once with ten realizations (sub-classing), or have a framework configured to do it ten slightly different ways, is only an implementation detail of the pattern being exposed. – Edwin Buck Jun 12 '14 at 19:05

1 Wandering back into this conversation after years, I think that a lot of people associate Patterns with a specific programming language or a specific programming paradigm. They can be used in such a context, but they existed before programming. "A timeless way of building" discusses patters in building architecture and community planning. This implies that the pattern oriented techniques can be used outside of "limitations of a language" unless you want to call building construction a programming language :) – Edwin Buck Mar 10 '16 at 17:55

As others have said, there are patterns specific of functional programming. I think the issue of getting rid of design patterns is not so much a matter of switching to functional, but a matter of **language features**.

Take a look at how Scala does away with the "singleton pattern": you simply declare an **object** instead of a class. Another feature, pattern matching, helps avoiding the clunkiness of the visitor pattern. See the comparison here: http://andymaleh.blogspot.com/2008/04/scalas-pattern-matching-visitor-pattern.html

And Scala, like F#, is a fusion of OO-functional. I don't know about F# but it probably has this kind of features.

Closures are present in functional language, but need not be restricted to them. They help with the delegator pattern.

One more observation. This piece of code implements a pattern: it's such a classic and it's so elemental that we don't usually think of it as a "pattern", but it sure is:

```
for(int i = 0; i < myList.size(); i++) { doWhatever(myList.get(i)); }
```

Imperative languages like Java and C# have adopted what is essentially a functional construct to deal with this: "foreach".

answered Nov 29 '08 at 21:43

Germán
**3,301** 3 22 33

I would say that Scala includes first-class support for the singleton pattern. The pattern is still there, but the boilerplate code needed to apply the pattern is greatly reduced compared to Java. – JacquesB Jan 2 '09 at 14:23

If opinions were like a*******, well... Look at the rest of the answers. "you simply declare an object instead of a class" is so true, I would explicitly call it an object literal though (ie var singleton = {};). I also like the mention about the foreach pattern. Unfortunately, it looks like most of the people who answered/commented on this question don't understand functional programming and would rather justify the usage of OOP design patterns. +1 for providing concrete examples, I would give more if I could. – Evan Plaice Feb 18 '12 at 15:17

@JacquesB I can't comment on Scala/Haskell but in JavaScript (ie. hybrid functional/imperative) there is absolutely no boilerplate you just adjust the way you declare objects by using combinations of object literal syntax, anonymous functions, passing functions as first class members, and allowing multiple inheritance (eliminating the need for interface contracts). – Evan Plaice Feb 18 '12 at 15:35

The GoF Design Patterns are coding workaround recipes for OO languages that are descendants of Simula 67, like Java and C++.

Most of the "ills" treated by the design patterns are caused by:

- statically typed classes, which specify objects but are not themselves objects;
- restriction to single dispatch (only the leftmost argument is used to select a method, the remaining arguments are considered as static types only: if they have dynamic types, it's up to the method to sort that out with ad-hoc approaches);
- distinction between regular function calls and object-oriented function calls, meaning that object-oriented functions cannot be passed as functional arguments where regular functions are expected and vice versa; and
- distinction between "base types" and "class types".

There isn't a single one of these design patterns that doesn't disappear in the Common Lisp Object System, even though the solution is structured in essentially the same way as in the corresponding design pattern. (Moreover, that object system precedes the GoF book by well over a decade. Common Lisp became an ANSI standard the same year that that book was first published.)

As far as functional programming is concerned, whether or not the patterns apply to it depends on whether the given functional programming language has some kind of object system, and whether it is modeled after the object systems which benefit from the patterns. That type of object-orientation does not mix well with functional programming, because the mutation of state is at the front and centre.

Construction and non-mutating access are compatible with functional programming, and so patterns which have to do with abstracting access or construction could be applicable: patterns like Factory, Facade, Proxy, Decorator, Visitor.

On the other hand, the behavioral patterns like State and Strategy probably do not *directly* apply in functional OOP because mutation of state is at their core. This doesn't mean they don't apply; perhaps they somehow apply in combination with whatever tricks are available for simulating mutable state.

edited Sep 16 '14 at 1:42

answered Dec 9 '13 at 17:45
Kaz
**36.2k** 7 60 95

2 "The GoF Design Patterns are coding workaround recipes" is simply a false statement. – John Peters Mar 10 '16 at 17:31

I'd like to plug a couple of excellent but somewhat dense papers by Jeremy Gibbons: "Design patterns as higher-order datatype-generic programs" and "The essence of the Iterator pattern" (both available here: http://www.comlab.ox.ac.uk/jeremy.gibbons/publications/).

These both describe how idiomatic functional constructs cover the terrain that is covered by specific design patterns in other (object-oriented) settings.

edited Feb 22 '16 at 2:51

answered Sep 17 '10 at 0:30
sclv
**34.4k** 5 76 173

You can't have this discussion without bringing up type systems.

> 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.

That's because these features don't address the same issues that OOP does... they are alternatives to imperative programming. The FP answer to OOP lies in the type systems of ML and Haskell... specifically sum types, abstract data types, ML modules, Haskell typeclasses.

> But of course there are still design patterns which are not solved by FP languages. What is the FP equivalent of a singleton? (Disregarding for a moment that singletons are generally a terrible pattern to use)

The first thing typeclasses do is eliminate the need for singletons.

You could go through the list of 23 and eliminate more, but I don't have time to right now.

answered Nov 30 '08 at 22:20
jganetsk
**127** 1

6 How do typeclasses (the FP equivalent of OOP interfaces) eliminate the need for singletons (the FP equivalent of global state)? – Gabe Sep 23 '10 at 19:51

---

I think only two GoF Design Patterns are designed to introduce the functional programming logic into natural OO language. I think about Strategy and Command. Some of the other GoF design patterns can be modified by functional programming to simplify the design and keep the purpose.

edited Oct 2 '12 at 10:30
jonsca
**8,315** 11 46 55

answered Oct 1 '12 at 10:51
CyaNnOrangeHead
**41** 1

4 Thing is, many patterns' main point is to harness polymorphism to do things that decent support for FP concepts could allow automatically. (Most incarnations i've seen of Builder, for example, are just half-assed currying.) Once you can easily treat functions as values, patterns often simplify to the point of triviality. They become "pass a callback" or "have a dictionary of callbacks" -- and different builder classes, for example, can all but disappear. IMO a pattern stops being a pattern once it's trivial enough to be just *how things work*, rather than something you need to implement. – cHao Jun 15 '13 at 17:41

---

OOP and the GoF patterns deal with states. OOP models reality to keep the code base as near as possible

OOP and the GoF patterns deal with states. OOP models reality to keep the code base as near as possible to the given requirements of reality. GoF design patterns are patterns that were identified to solve atomic real world problems. They handle the problem of state in a semantic way.

As in real functional programming no state exists, it does not make sense to apply the GoF patterns. There are not functional design patterns in the same way there are GoF design patterns. Every functional design pattern is artifical in contrast to reality as functions are constructs of math and not reality.

Functions lack the concept of time as they are always return the same value whatever the current time is unless time is part of the function parameters what makes it really hard to prrocess "future requests". Hybrid languages mix those concepts make the languages not real functional programming languages.

Functional languages are rising only because of one thing: the current natural restrictions of physics. Todays processors are limited in their speed of processing instructions due to physical laws. You see a stagnation in clock frequency but an expansion in processing cores. Thats why parallelism of instructions becomes more and more important to increase speed of modern applications. As functional programming by definition has no state and therefore has no side effects it is safe to process functions safely in parallel.

GoF patterns are not obsolete. They are at least necessary to model the real world requirements. But if you use a functional programming language you have to transform them into their hybrid equivalents. Finally you have no chance to make only functional programs if you use persistence. For the hybrid elements of your program there remains the necessity to use GoF patterns. Any other element that is purely functional there is no necessity to use GoF patterns because there is no state.

Because the GoF pattern are not necessary for real functional programming that doesn't mean that the SOLID principles should not be applied. The SOLID principles are beyond any language paradigm.

answered Jun 28 '16 at 22:11
oopexpert
**610** 7 10

2 FP can have state -- just no global, shared, or mutable state. – vt5491 Jul 8 '17 at 20:33

Essentially, **yes**!

- When a pattern circumvents the missing features (high order functions, stream handling...) that ultimalty facilitate composition.
- The need to re-write patterns' implementation again and again can itself be seen as a language smell.

Besides, this page (AreDesignPatternsMissingLanguageFeatures) provides a "pattern/feature" translation table and some nice discussions, if you are willing to dig.

edited Apr 22 '17 at 0:10
Peter Mortensen
**12.7k** 19 82 110

answered Oct 19 '12 at 19:44
YvesgereY
**1,473** 10 13

---

Functional programming does not replace design patterns. Design patterns can not be replaced.

Patterns simply exist; they emerged over time. The GoF book formalized some of them. If new patterns are coming to light as developers use functional programming languages that is exciting stuff, and perhaps there will be books written about them as well.

answered Nov 30 '08 at 22:00
ktingle
**345** 2 8

1 Design patterns can not be replaced? That is a bit closed minded I think. We can probably all agree that design patterns are meant to solve programming problems, and I at least would like to hope that some day we can solve those problems without design patterns. – Metropolis May 21 '10 at 20:50

3 Any *particular* pattern might be replaceable, but the concept of *patterns* can not. Remember that the term "pattern" arose in field of *architecture*. – Frank Shearar Jun 3 '11 at 11:05

1 Patterns are not meant to solve programming problems. Patterns are ways we program. The documentation of patterns are meant to help solve programming problems. – Torbjørn Jan 5 '12 at 13:02

3 @Torbjørn: Patterns are ways we program *when the language gets in the way*. They exist because of some mismatch between the program's desired behavior and the language's built-in abilities, where the requirements and the abilities don't map well or map ambiguously. If it weren't for that, there'd be no pattern; you'd have one implementation that's just *how things are done*, and other implementations would effectively be not worth considering. – cHao Jun 15 '13 at 20:05

1 Except that patterns truly exist only to facilitate communication. There is no other purpose. And in all of the design meetings I've attended over the years, a discussion of the *algorithm* is what was important, not the pattern. The pattern rarely explained what was truly going on in any meaningful sense. Does it explain precisely the O(n) vs O(n Log(n)) impacts? No. Does it explain how easily it'll fit with the existing architecture? No. Full scale algorithm discussions do. I'm not arguing that patterns should be retired per se, but if they were, almost nothing would suffer as a result. – user4229245 May 18 '15 at 13:57

In the new 2013 book named *"Functional Programming Patterns- in Scala and Clojure"* the author Michael.B. Linn does a decent job comparing and providing replacements in many cases for the GoF patterns and also discusses the newer functional patterns like 'tail recursion', 'memoization', 'lazy sequence', etc.

This book is available on Amazon. I found it very informative and encouraging when coming from an OO background of a couple of decades.

edited Apr 22 '17 at 0:15
Peter Mortensen
**12.7k** 19 82 110

answered May 20 '14 at 12:04
manish
**31** 2

I think that each paradigm serves a different purpose and as such cannot be compared in this way.

I have not heard that the GoF design patterns are applicable to every language. I have heard that they are applicable to all **OOP languages**. If you use Functional programming then the domain of problems that you solve is different from OO languages.

I wouldn't use functional language to write a user interface but one of the OO languages like C# or Java would make this job easier. If I were writing a functional language then I wouldn't consider using OO Design Patterns.

answered Nov 29 '08 at 20:18
Vincent Ramdhanie
**87.2k** 20 120 168

OOP and FP have different goals, OOP aims to encapsulate the complexities/moving parts of software components and FP aims to minimize the complexity and dependencies of software components however these 2 paradigms are not necessarily 100% contradicting and could be applied together to get the benefit from both worlds. Even with a language that does not natively support functional programming like C#, you could write functional code if you understand the FP principles, likewise you could apply OOP principles using F# if you understand OOP principles, patterns and best practices. You would make the right choice based on the situation and problem that you try to solve, regardless of the programming language you use.

answered Dec 13 '16 at 10:57

Dogu Arslan
**1,450** 9 21

As the accepted answer said, OOP and FP all have their specific patterns.

However, there are some patterns which are so common that all programming platforms I can think of should have. Here is an (incomplete) list:

- Adapter. I can hardly think of a useful programming platform which is so comprehensive (and self-fulfilled) that it does not need to talk to the world. If it is going to do so, an adapter is definitely needed.
- Façade. Any programming platforms that can handle big source code should be able to modularise. If you were to create a module for other parts of the program, you will want to hide the "dirty" parts of the code and give it a nice interface.
- Interpreter. In general, any program is just doing two things: parse input and print output. Mouse inputs need to be parsed, and window widgets need to be printed out. Therefore, having an embedded interpreter gives the program additional power to customise things.

Also, I noticed in a typical FP language, Haskell, there is something similar to GoF patterns, but with different names. In my opinion this suggest they were there because there are some common problems to solve in both FP and OOP languages.

- Monad transformer and decorator. The former used to add additional ability into an existing monad, the latter add additional ability to an existing object.

edited Apr 22 '17 at 0:23
Peter Mortensen
**12.7k** 19 82 110

answered Jun 12 '14 at 12:28
Earth Engine
**4,766** 3 29 57

**protected** by Nasreddine May 18 '16 at 10:21

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 reputation on this site (the association bonus does not count).

Would you like to answer one of these unanswered questions instead?