Are side effects a good thing? [closed]

Asked 15 years, 8 months ago Modified 4 years, 4 months ago

Viewed 11k times

32

Closed. This question is opinion-based. It is not currently accepting answers.

Want to improve this question? Update the question so it can be answered with facts and citations by editing this post.

Closed 4 years ago.

Improve this question

I feel the term rather pejorative. Hence, I am flabbergasted by the two sentences in Wikipedia:

> Imperative programming is known for employing side effects to make programs function. Functional programming in turn is known for its minimization of side effects. [1]

Since I am somewhat Math-biased, the latter sounds excellent. What are the arguments for side-effects? Do

they mean the loss of control or the acceptance of uncertainty? Are they a good thing?

functional-programming procedural-programming side-effects

Share

Improve this question

Follow

edited Apr 18, 2009 at 21:28

asked Apr 18, 2009 at 17:49

Léo Léopold Hertz 준영

140k ● 187 ● 459 ● 713

13 Answers

Sorted by: Highest score (default)

58

Every so often I see a question on SO which forces me to spend half an hour editing a really bad Wikipedia article. The article is now only moderately bad. In the part that bears on your question, I wrote as follows:

> In computer science, a function or expression is said to have a **side effect** if, in addition to producing a value, it also modifies some state or has an *observable* interaction with calling functions or the outside world. For example, a function might modify a global or a static variable, modify one of its arguments, raise an

> exception, write data to a display or file, read data, call other side-effecting functions, or launch missiles. In the presence of side effects, a program's behavior depends on past history; that is, the order of evaluation matters. Because understanding an effectful program requires thinking about all possible histories, side effects often make a program harder to understand.
>
> Side effects are essential to enable a program to interact with the outside world (people, filesystems, other computers on networks). But the degree to which side effects are used depends on the programming paradigm. Imperative programming is known for uncontrolled, promiscuous use of side effects. In functional programming, side effects are rarely used. Functional languages such as Standard ML and Scheme do not restrict side effects, but it is customary for programmers to avoid them. The functional language Haskell restricts side effects with a static type system; only a function that produces a result of IO type can have side effects.

answered Apr 20, 2009 at 3:03

Norman Ramsey

202k ● 62 ● 371 ● 541

30

Side effects are a necessary evil, and one should seek to minimize/localize them.

Other comments on the thread say effect-free programming is sometimes not as **intuitive**, but I think that what people consider "intuitive" is largely a result of their prior experience, and most people's experience has a heavy imperative bias. Mainstream tools are becoming more and more functional each day, because people are discovering that **effect-free programming leads to fewer bugs** (though admittedly sometimes a new/different class of bugs) due to less possibility of separate components interacting via effects.

Almost no one has mentioned performance, and **effect-free programming usually has worse performance** than effectful, since computers are von-Neumann machines that are designed to work well with effects (rather than being designed to work well with lambdas). Now that we're in the midst of the multi-core revolution, this may change the game as people discover they need to take advantage of cores to gain perf, and whereas parallelization sometimes takes a rocket-scientist to get right effect-fully, it can be easy to get right when you're effect-free.

Share Improve this answer

Follow

answered Apr 18, 2009 at 20:30

Brian

119k ● 17 ● 243 ● 304

4 Effect-free programming can be as efficient as effectful programming (see mlton.org or caml.inria.fr), but it's true that compiler writers must work harder to make it so.
– Norman Ramsey Apr 20, 2009 at 2:24

18

In von-Neumann machines, side effects are things that make the machine work. Essentially, no matter how you write your program, it'll need to do side-effects to work (at a low level view).

Programming without side effects means abstracting side effects away so that you could think about the problem in general -without worrying about the current state of the machine- and reduce dependencies across different modules of a program (be it procedures, classes or whatever else). By doing so, you'll make your program more reusable (as modules do not depend on a particular state to work).

So yes, side-effect free programs are a good thing but side-effects are just inevitable at some level (so they cannot be considered as "bad").

Share Improve this answer Follow

edited Apr 18, 2009 at 18:27

answered Apr 18, 2009 at 17:57

Mehrdad Afshari
422k ● 92 ● 859 ● 794

> Side-effect free programs are not a good thing. All side-effect free programs can be optimized away entirely.
> – David Conrad Dec 17, 2012 at 17:05

10

Pro:

- In the end side effects are what you want to accomplish.
- Side effects are natural for code that interacts with outside world.
- They make many algorithms simple.
- To avoid using side effects, you need to implement loops by recursion, thus your language implementation needs tail call optimization.

Con:

- Pure code is easy to parallelize.
- Side effects can make code complicated.
- Pure code is easier to prove correct.

For example Haskell, at first it seems very elegant, but then you need to start playing with outside world and it's not so much fun anymore. (Haskell moves state as a function parameter and hides it into things called Monads, which enable you to write in imperative look-a-like style.)

Share Improve this answer edited Apr 19, 2009 at 8:57

Follow

answered Apr 18, 2009 at 17:57

John Smith

4,512 4 35 34

2 Side effects make algorithms simple because they are very powerful, and powerful tools should be used sparingly. – Anton Tykhyy Apr 18, 2009 at 21:32

1 For loops, the compiler only needs to optimize tail recursions, not tail calls in general, and this is just too easy. – Ingo Apr 20, 2009 at 8:32

2 Less vs. more modular is more clear and specific than "complicated", I think. – Kzqai Feb 14, 2010 at 18:30

8

Side effects are just like any other weapon. They are unquestionably useful, and potentially very dangerous when improperly handled.

Like weapons, you have side effects of all different kinds of all different degrees of lethality.

In C++, side effects are totally unrestricted, thanks to pointers. If a variable is declared as "private", you can still access or change it using pointer tricks. You can even alter variables which aren't in scope, such as the parameters and locals of the calling function. With a little help from the OS (mmap), you can even modify your program's machine code at runtime! When you write in a language like C++, you are elevated to the rank of Bit God, master of all memory in your process. All

optimizations the compiler makes to your code are made with the assumption you don't abuse your powers.

In Java, your abilities are more restricted. All variables in scope are at your control, including variables shared by different threads, but you must always adhere to the type system. Still, thanks to a subset of the OS being at your disposal and the existence of static fields, your code may have non-local effects. If a separate thread somehow closes System.out, it will seem like magic. And it *will* be magic: side effectful magic.

Haskell (despite the propaganda about being pure) has the IO monad, which requires you register all your side effects with the type system. Wrapping your code in the IO monad is like the 3 day waiting period for handguns: you can still blow your own foot off, but not until you OK it with the government. There's also unsafePerformIO and its ilk, which are Haskell IO's black market, giving you side effects with "no questions asked".

Miranda, the predecessor to Haskell, is a pure functional language created before monads became popular. Miranda (as far as I've learned... if I'm wrong, substitute Lambda Calculus) has no IO primitives at all. The only IO done is compiling the program (the input) and running the program and printing the result (the output). Here, you have full purity. The order of execution is completely irrelevant. All "effects" are local to the functions which declare them, meaning never can two disjoint parts of code effect each other. It's a utopia (for mathematicians).

Or equivalently a distpia. It's boring. Nothing ever happens. You can't write a server for it. You can't write an OS in it. You can't write SNAKE or Tetris in it. Everyone just kind of sits around looking mathematical.

Share Improve this answer

Follow

answered Apr 26, 2010 at 14:57

Tac-Tics

1,925 ● 13 ● 19

> Sorry about ressurecting a answer from 10 years ago, but don't you mean a dystopia? – ARI FISHER Nov 27, 2020 at 17:39

7

Without side-effects, you simply can't do certain things. One example is I/O, since making a message appear on the screen is, by definition, a side-effect. This is why it's a goal of functional programming to minimize side-effects, rather than eliminate them entirely.

Setting that aside, there are often instances where minimizing side-effects conflicts with other goals, like speed or memory efficiency. Other times, there's already a conceptual model of your problem that lines up well with the idea of mutating state, and fighting against that existing model can be wasted energy and effort.

Share Improve this answer

Follow

answered Apr 18, 2009 at 17:59

Pillsy

9,891 ● 1 ● 45 ● 72

4

It is true, as some people here mention, that without side effects one cannot make a useful application. But from that it does not follow that using side effects in an uncontrolled way is a good thing.

Consider the following analogy: a processor with an instruction set that had no branch instructions would be ansolutely worthless. However, it does not follow that programmers must use **goto**s all the time. On the contrary, it turned out that structured programming and later OOP languages like Java could do without even having a goto statement, and nobody missed it.

(To be sure, there is still goto in Java - it's now called **break**, **continue** and **throw**.)

Share Improve this answer

Follow

answered Apr 20, 2009 at 8:46

Ingo

36.3k ● 6 ● 55 ● 102

if, else, while, and for are just as much of gotos are break, continue, and throw are. – hasen Apr 20, 2009 at 9:03

3

Side effects are essential for a significant part of most applications. Pure functions have a lot of advantages. They are easier to think about because you don't have to worry about pre and post-conditions. Since they don't change state, they are easier to parallelize, which will become very important as the processor-count goes up.

Side effects are inevitable. And they should be used whenever they are a better choice than a more complicated but pure solution. The same goes for pure functions. Sometimes a problem is better approached with a functional solution.

It's all good =) You should use different paradigms according to the problem you're solving.

Share Improve this answer

Follow

answered Apr 18, 2009 at 18:42

bigmonachus

993 ● 1 ● 8 ● 15

3

Without side effects, you can't perform I/O operations; so you can't make a useful application.

Share Improve this answer

Follow

answered Apr 18, 2009 at 20:36

hasen

166k ● 66 ● 196 ● 233

2

That quote really made me chuckle. That said, I find the minimization of side effects to really translate to code that is much easier to reason about and maintain. However, I don't have the luxury of exploring functional programming quite as much as I would like.

The way I look at it when working in object-oriented and procedural languages that revolve around side effects is

to *contain* and *isolate* side effects.

As a basic example, a video game has a necessary side effect of rendering graphics to a screen. However, there are two different kinds of design paths here with respect to side effects.

One seeks to minimize and loosen coupling by making the renderer very abstract and basically told what to render. The other parts of the system then tell the renderer what to draw and that could be a batch of primitives like triangles and points with projection and modelview matrices or maybe something higher-level like abstract models and cameras and lights and particles. Either way, such a design revolves around many things causing external side effects, since potentially many parts of the codebase will be pushing changes to the renderer (no matter how abstract or indirect, the net effect is still a whole bunch of things in such a system triggering external rendering side effects).

The other way is to *contain/isolate* those side effects. Instead of the renderer being told what to render, it instead becomes coupled to the game world (though this could just be some basic abstractions and maybe access to a scene graph). Now it accesses the scene on its own (read-only access) and looks through the scene and figures out what to render using more of a pull-style design. That leads to more coupling from renderer to game world but it also means the side effects related to

screen output are now completely contained inside the renderer.

This latter design *contains* or *isolates* side effects, and I find that type of design much easier to maintain and keep correct. It still causes side effects but all the side effects related to outputting graphics to a screen are now entirely contained in the renderer. If there's an issue there, you know the bug is going to be in the renderer code and not the result of something external misusing it and telling it the wrong things to do.

Because of this, when it comes to coupling, I have always found it more desirable to maximize efferent (outgoing) couplings in things that cause external side side effects and minimize afferent (incoming) couplings. This applies regardless of abstractions. In the context of side effects, a dependency to `IRenderer` is still a dependency to a concrete `Renderer` as far as communication goes with respect to what side effects are going to happen. The abstraction makes no difference as far as what side effects are going to occur.

The renderer should depend on the rest of the world so that it can completely isolate those side effects to the screen; the rest of the world shouldn't depend on the renderer. Same kind of analogy for a file saver. The file saver shouldn't be told what to save by the outside world. It should look at the world around it and figure out what to save on its own. Such would be the design path that seeks to isolate and contain side effects; it tends to be

more pull-based than push-based. The result tends to introduce a bit more coupling (though it could be loose) if you graph out the dependencies since the saver might need to be coupled with things it's not even interested in saving, or the renderer might need read-only access to things it's not even interested in rendering to discover the things it is interested in renderering.

However, the end result is that the dependencies flow *away* from side effects instead of *towards* side effects. When we have a system with many dependencies flowing towards pushing external side effects, I have always found those the hardest to reason about since so many parts of the system could potentially be changing external states to the point where it's not just hard to figure out what's going to happen but also *when* and *where*. So the most straightforward way to correct/prevent that problem is to seek to make dependencies flow *away* from side effects, not towards them.

Anyway, I have found favoring these types of designs the practical way to help avoid bugs and also help detect and isolate them when they exist to make them easier to reproduce and correct.

Another useful strategy I find is to make side effects more homogeneous for any given loop/phase of the system. For example, instead of doing a loop which removes associated data from something, delinks it, and then removes it, I have found it much easier if you do three homogeneous loops in such cases. The first

homogeneous loop can remove associated data. The second homogeneous loop can delink the node. The third homogeneous loop can remove it from the rest of the system. That's on a lower-level note related more to implementation than design, but I have often found the result easier to reason about, maintain, and also even optimize (easier to parallelize, e.g., and with improved locality of reference) -- you take those non-homogeneous loops triggering multiple different types of side effects and break them down into multiple homogeneous loops, each triggering just one uniform kind of side effect.

edited Jan 3, 2018 at 8:57

answered Jan 3, 2018 at 8:13

user4842163

Since your program has to have side effects to have any output or interesting effect (apart from heating your CPU), the question is rather where these side effects should be triggered in your program. They become only harmful if they are hidden in methods you don't expect them.

As a rule of thumb: Separate pure methods and methods with side effects. A method that prints something to the console should do only that and not compute some interesting value that you might want to use somewhere else.

Share Improve this answer

Follow

answered Jun 17, 2015 at 10:05

lex82

11.3k ● 2 ● 46 ● 71

0

Well, it's a lot easier and more intuitive to program with side effects, for one thing. Functional programming is difficult for a lot of people to wrap their head around -- find someone who's taught/TAed a class in Ocaml and you'll probably get all kinds of stories about people's abject failure to comprehend it. And what good is having beautifully designed, wonderfully side effect free functional code if nobody can actually follow it? Makes hiring people to get your software done rather difficult.

That's one side of the argument, at least. There's any number of reasons lots of people are going to have to learn all about functional style, side-effect-less code. Multithreading comes to mind.

Share Improve this answer

Follow

answered Apr 18, 2009 at 17:54

Promit

3,507 ● 1 ● 21 ● 30

2 Actually, while it is more difficult to find programmers in a functional language, many people argue that finding **great** programmers is actually easier. – Magnus Kronqvist Jun 20, 2011 at 16:39

I think the balanced answer is that one should look for opportunities to minimize or avoid side effects or to think

0

of where they are and look for opportunities to move them somewhere else to make code easier to understand.

Here I give two versions of some code. When I started to look for opportunities to change code that looks like the first version into code that looks like the second version, my life got better:

It might have an object with attributes `first, second, third` and methods `first_method, second_method, third_method` like

```
def first_method(self):
    # calculate and set self.first

def second_method(self):
    # use self.first to calculate and set self.second

def third_method(self):
    # use self.first and self.second to calculate self
```

and then a higher level method:

```
def method(self):
    self.first_method()
    self.second_method()
    self.third_method()
```

Now imagine the names aren't prefixed by "first", "second", "third" but describe what the methods do, and that the program is more complex. This is a problem I routinely face when exploring code that abuses side effects. I constantly have to look at the implementation of

the functions to understand what the effect of calling them is and how they work together.

Now without going nuts with the "side effects are evil" stuff, we can still benefit from a desire to avoid side effects:

```
def first_func():
    # calculate the first thing and return it

def second_func(first_thing):
    # use first_thing to calculate second_thing and re

def third_func(first_thing, second_thing):
    # use first_thing and second_thing to calculate th
```

and that higher level method could look like

```
def method(self):
    self.first_thing = first_func()
    self.second_thing = second_func(self.first_thing)
    self.third_thing = third_func(self.first_thing, se
```

We still have side effects, `method` sets attributes of the object, but when we go to try to understand how the functions that make it up work together, it's crystal clear that you have to call them in this order and it is also crystal clear what each of them need to do their work.

Also, with the first version, when looking at the implementation of `method`, who knows what other attributes of the object get changed by each function. When looking at the second version, it's plain for all to see that calling `method` changes only three attributes

without having to look at the implementation of the functions.

This example might seem simplistic because I crafted it out of thin air for this explanation. I tried my best to convey some real problems that I have when trying to understand code that was written without a certain slight disdain for side effects. Real world code that operates like the first version is harder to understand than real world code that operates like the second version.

My take on side effects, as others have said, is that it is worth while to ask yourself whether the side effects can be moved or avoided and that doing so often results in code that is easier to understand.

edited Aug 12, 2020 at 19:29

answered Jul 9, 2020 at 1:16

Philippe Carphin
148 ● 2 ● 7