Community

yegor256.com

65 Comments • Created 3 months ago

Don't Create Objects That End With -ER

Manager. Controller. Helper. Handler. Writer. Reader. Converter. Validator. Router. Dispatcher. Observer. Listener. Sorter. Encoder. Decoder. This is the class names hall of shame. Have you seen them in your code? In open source librarie…

(yegor256.com)

65 Comments

Recommend 5 Share Sort by Newest

Join the discussion…

David Johnston • 2 months ago

How about "-or"?

ApplePossessor ap = new ApplePossessor(apples);
Apple apple = ap.biggestApple(AppleComparisonMetric.WEIGHT);

The goal being to create a black-box that I can give a bunch of apples to and then ask for them back when I need them.

Then, having done that, it is not a stretch to add a public static method whereby I simply hand over the apples while simultaneously making the request. So now I'm back to a utility class that can be instantiated if I wish to have it keep the apples long-term or can use statically if the apples are no longer important beyond having retrieved the one I want.

Generalists are people too and they can do an acceptable job with considerably less overhead than hiring a bunch of specialists.

People, of which programmers are arguably members, are familiar with dumb data objects and intelligent people objects. Decomposing and then putting up a friendly facade is considerably more supportable than creating unnatural intelligence in the data. As was mentioned elsewhere - the needed functionality for this particular scenario would likely be best accomplished by incorporating pure side-effect free functions within whatever method the top-level application called that necessitated the retrieval of the largest apple in the first place.

• Reply • Share ›

Albert Mendonca • 2 months ago

I think the meta point made here (with a tad bit of exaggeration) is to transition from procedural way of thinking to thinking in terms of *real* objects. Domain Driven Design is a great example of this and i fully support this notion.

1 • Reply • Share ›

Literally__No__One • 3 months ago

Joel Spolsky you aren't. I'm flabbergasted with the amount of dogmatism over unimportant trivialities shown here, with any good points being made in the posts ending up lost in said dogmatism. Appalling.

• Reply • Share ›

amor enew • 3 months ago

so for example : BookHelper class will be BookTable or BookDatabase

1 • Reply • Share ›

coder.slynk • 3 months ago

You would hate Golang, with your arbitrarily self imposed rules:

https://golang.org/doc/effecti...
https://golang.org/doc/effecti...

Also, logically a class name like "Sorted" does not declare whether it sorts or not. In fact, for all I know, the constructor could throw an exception if the passed list is not already sorted.

• Reply • Share ›

rtoal → coder.slynk • 2 months ago

True, but Golang is not even _trying_ to be OO in the classical sense of term (Kay's "little computers" or Bugayenko's "living organisms"). Golang just lets you use -er-named interfaces on dumb structs. If you want to program with dumb structs, use Go and use it well. If you want to program in a more pure OO style, there are good reasons to avoid -ER.

1 • Reply • Share ›

coder.slynk → rtoal • 2 months ago

If you're choosing what language to use for your project based on its relative "OOness" you're doing it wrong. Golang simply outperforms most server side languages, it scales better, and is centered around a thriving open source community. To toss that aside because you are obsessed with OOP source community. To toss that aside because you are obsessed with OOP makes you a bad software developer (the generalized "you", not you specifically.) Go's structs are smarter than C structs, they just are not as powerful as a class but then again, I rarely need polymorphism in a server side project anyway so who cares.

• Reply • Share ›

rtoal coder.slynk • 2 months ago

I agree with you and like and use Go. Just saying that if you want to be in the OO world, and you want to be all-OO, then fine, follow Yegor's advice and keep a clean, all OO architecture. If you want some of the best what modern languages have to offer, "Code in Go, Code in Go, Code in Go" (sung to the tune of that Frozen song). I also like Rust and Julia, and these languages also are not pure-OO. Yes, obsessing over a paradigm is detrimental, and to say OO trumps Go-style, Rust-style, or Julia-style architectures would be wrong. But a consistent style is a virtue. And right Implementation inheritance and all that fancy polymorphism isn't the biggest deal with OO, though Java and C++ noobs seem to think it is. Alan Kay has always said it was about messaging, not the C++ stuff we all ended up with. (EDIT: I now see that your reply to me was likely based on my use of the word "dumb" to qualify structs! I meant dumb in the sense of dumb objects being data structures without embedded behavior as opposed to Go being dumb!)

1 • Reply • Share ›

coder.slynk rtoal • 2 months ago

I agree, a consistent style is a virtue, but only on a project by project basis. The author of this article, however, has stated in his other articles that OOP is *the* only correct way to program period and he refuses to use any other style regardless of what the project demands. That's obviously his choice, but you would be missing the intention of this article if you took it at face value; he writes these articles not as "OOP" rules but as rules that every project should follow, independent of paradigms. I mean he never once said that you should avoid -ER classes in *OOP*; he specifically states:

"They all end in "-er". And what's wrong with that? They are not classes, and the objects they instantiate are not objects. Instead, they are collections of procedures pretending to be classes."

But objects exist outside of OOP, which I'm not sure the author is even aware of or not. My point, other than to poke fun at the author, was to show that -ER classes are expected and correct in other languages/paradigms/styles. Making blanket statements like this Author makes in his articles is just pointless without focus. He basically just says "go read these other things if you don't believe my proposed thesis, I'm just going to show you what to do instead." His only explanation is that -ER classes are all manager classes and only explanation is that -ER classes are all manager classes and manager classes are bad, but only 1 class name he mentioned at the beginning of the article is actually a manager class (maybe 2 with the Observer.)

The article would have actual value if it discussed exactly why -ER classes inherently go against OO principles. Instead, it just yet again comes of as an opinionated mess without any research behind it.

2 • Reply • Share ›

Yegor Bugayenko author coder.slynk • 3 months ago

Yeah, Golang is a shame...

• Reply • Share ›

Sihle • 3 months ago

...that reminds me: I need to eat my green apple I brought for today. Thanks for great article by the way...

• Reply • Share ›

Riccardo Cardin • 3 months ago

Ok, I've understood your point of view. But I think we have to make some considerations about objects. As far as I learnt until now as a developer, I think we could divide objects in two sets:

1) Objects which model reality and domains and the operations defined on them;
2) Objects which cooperate with the previuos ones to build a software architecture through which the user's needs are satisfied.

The objects belonging the first set might follow the "not -ER" rule that you're explaining in the article.
Objects like Controllers, Services, Decorators, Factories belong to the second set. They have nothing to deal with the "real world", but they help the first ones to interact eachother. Then, I think that for this second set your "not -ER" rule is too strict and cannot hold.

1 • Reply • Share ›

Yegor Bugayenko author Riccardo Cardin • 3 months ago

There should be no difference between "foreign" and "domestic" objects. Each object has its own scope of visibility and inside this scope all other objects are foreign to it. Whether these objects are connected to reality, as you call it, doesn't matter. We actually don't need to know that. All we know about an object is the behavior it exposes to us via its methods.

1 • Reply • Share ›

Riccardo Cardin Yegor Bugayenko • 3 months ago

Ok, I agree with you when you say that both kinds of object have its own scope of visibility. This is one of the principles of OOP. But, what I am trying to say is that "domestic" objects should not care about persisting themself or that they are exposed as JSON objects through the Internet. These functionalities should be implemented by "foreign" objects, respecting OOP principles of information hiding, polimorphism, and so on. Then, imho, it is correct that a "persistence manager" is called using a -ER postfix.

Also the Decorators, which you love so much, fall into "foreign" objects. They help other kind of object to extend their features in a dynamic and composable way.

3 • Reply • Share ›

David Raab • 3 months ago

> List<apple> sorted = new Sorter().sort(apples);
> Apple biggest = sorted.get(0);

The code above is not only imperative. It is not even object-oriented. In fact it is just prodecural or functional. Call a function and get a result. The method sort could also live in a Utility class (in C#)

> var sorted = Sorter.sort(apples)

Your improvement

> List<apple> sorted = new Sorted(apples);
> Apple biggest = sorted.get(0);

is in fact more OO. But it is still not declarative! Declarative is the definition of "WHAT TO DO" not "HOW TO DO IT". Sorting and selecting the first one is still imperative because you describe an algorithm "first sort, then select last" to describe how to get the biggest apple. But a declarative version would just be a "give me the biggest apple".

In that sense the second version is more OO, but still not declarative. Versions how a declarative implementation could look like

# A static Helper Utility
var biggest = Apple.Biggest(apples)

# A specific Apple Collection with a "biggest"
var biggest = apples.BiggestApple()

# A more generic version for a "BiggestApple" method
var biggest = apples.Max()

1 • Reply • Share ›

Yegor Bugayenko author → David Raab • 3 months ago

You're absolutely right. Using my example, I would do it like this:

```
Apple biggest = new Biggest(new Sorted(apples));
```

• Reply • Share ›

Marcos Douglas Santos → Yegor Bugayenko • 3 months ago

So, we will make a class for each method? :)

4 • Reply • Share ›

Kanstantsin Kamkou ➦ Marcos Douglas Santos • 3 months ago

Each interface I assume

• Reply • Share ›

Marcos Douglas Santos ➦ Kanstantsin Kamkou • 3 months ago

Considering the examples, I see a class for each method.

• Reply • Share ›

Afroradiohead ➦ Marcos Douglas Santos • 3 months ago

I like the idea of class for each method.

A lot of people have been preaching the single responsibility principle, yet have a hard time confining to it. A class that literally does 1 thing and only 1 thing would be a class with only a constructor/desctructor methods as public, every other method is private or doesn't exist, and properties are readonly public.

Anyway the above is a tangent. Either way I do think we need to take a step back and question OO's rules so we can better design our software... and I'm just replying to you because you noticed something i noticed heh

• Reply • Share ›

Cliff ➦ Afroradiohead • 2 months ago

That sounds distinctly like Functional Decomposition, which is an anti-pattern in OOP.

• Reply • Share ›

Afroradiohead ➦ Cliff • 2 months ago

I read the article on http://c2.com/cgi/wiki?Functio... which I think you pointed to but isn't showing in Disqus. But it looks similar, though it's definitely different.

I'm talking about :
-Constructor is the only method of that object. (I'm using languages that doesn't have polymorphism so in those high level languages I imagine it might be okay to have multiple constructors).
-Once the object is built, nothing can change it's properties (except in dynamic languages like javascript, the object can listen to events and change itself)
-The object's properties are readonly to the outside (No setters allowed. For that matter, no getters either. It should not be the object's responsibility to cater to everything wanting data from it.)

With that said, yeah Functional Decomposition looks like it can get wild for sure.

• Reply • Share ›

David Raab ↪ Afroradiohead • 3 months ago

If you follow your advice then you also can stop using OO and just use functional programming. Instead of a class for every method you just create a function for, well, every function. Every function is automatically "single responsibility".

1 ^ | ⌄ • Reply • Share ›

Marcos Douglas Santos ↪ Afroradiohead • 3 months ago

In principle it seems a good idea, but in a more complex system there would be a huge explosion of classes. Imagine you thinking of names for a million classes.

1 ^ | ⌄ • Reply • Share ›

Afroradiohead ↪ Marcos Douglas Santos • 3 months ago

Millions of classes would definitely be crazy! But here's the thing, I've implemented this in a few systems and the alternative seemingly felt like the system was on the road to being complexed. This principle made it clear what the object "is". Unfortunately, because I'll probably never create a complex system, I won't know if it makes more/less classes :(

On that note, in yegor's example, I would do the following:

BiggestSortedAppleList biggestSortedAppleList = new BiggestSortedAppleList(appleList);

^ | ⌄ • Reply • Share ›

David Raab ↪ Afroradiohead • 3 months ago

Not that i would ever create a class for that, but if you do then the following makes more sense.

> BiggestApple biggestApple = new BiggestApple(appleList);

Because "sorting" is an implementation detail. You also can get the biggest Apple without sorting, and if it gets sorted or not is not important.

^ | ⌄ • Reply • Share ›

Afroradiohead ↪ David Raab • 3 months ago

A list of apples can totally be sorted without a class, and truth be told that's might be a better way to go? (Because I haven't coded it yet I can't tell)

As for the class implementation of biggestApple = BiggestApple(appleList), when i look at biggestApple I see 1 apple. The world List is a suffix that lets anyone read and understand "hey it's a list of apples". As for the word Sorted, it may not be perfect, but it was meant to indicate that something was done to this AppleList.

Ideally, at least when I design, I like to make my classes named in a way where I never have to look inside of that file to know what it does. BiggestApple or BiggestAppleList to me would mean Biggest Apple Object or Biggest Apple Object List. I do like the idea of Biggest Apple Sorted Object List... or Sorted By Biggest Apple Object List (SortedByBiggestAppleList)... but again, unfortunately we haven't created a rule everyone can agree with when designing class :((yet!)

Also... explaining in your code whether it was sorted or not is very important. I'm not a fan of comments and a straight up explaining it in a class name is as clear as I could get.

• Reply • Share ›

David Raab Yegor Bugayenko • 3 months ago

Yes, that would be much more declarative. In a completely OO language that would probably the best way. But i wouldn't do the whole thing OO at all. One reason because i feel much more as a functional guy. Another thing is because i feel OO is more bloated than it should be and to unflexible. For example creating just two classes just for sorting and selecting the biggest doesn't feel easy or intuitive. And for the unflexibility. How exactly is the "biggest apple" measured? Is the biggest apple the one with the maximum high? The one with the greatest volume? The apple that has the maximum weight?

You can build your Apples with a predefined equality that chooses one of those. But how can you than select other Apples based on another measurement? How can i say "Give the talest apple", "Give the apple that weight the most", "Give the apple with the greatest volume", ... and so on.

OO in general doesn't help so much, or i feel that OO code is too much complicated for even easy things like that. In a functional language you just have a function, provide a list, and provide a function that chooses the comparision. For example in F# you could end up with

```
let maxHeightApple = List.maxBy (fun apple -> apple.height) apples
let maxVolumeApple = List.maxBy (fun apple -> apple.volume) apples
let maxWeightApple = List.maxBy (fun apple -> apple.weight) apples
```

Or if you want a extra function for it

```
let heighestApple = List.maxBy (fun apple -> apple.height)
let biggestApple = heighestApple apples
```

This is a lot more easier than OO in general. I don't exactly knew how to start a generic Biggest class that have the same flexibility without ending in hundred lines of code, interface definition or other stuff. So i think your example is probably from what was given the best OO way. But i wouldn't solve the whole thing at all with OO-code at all. Even in C# i would end up with LINQ Code like

var biggestApple = apples.Aggregate((maxApple, currentApple) => currentApple.height > maxApple.height ? currentApple : maxApple);

that is more what a "List.maxBy" would do. And i still think this is still declarative. I would probably never end up writing classes for it, because i don't really see any value in creating two seperated classes for it.

• Reply • Share ›

Yegor Bugayenko author ➦ David Raab • 3 months ago

The problem with this "apple.weight" approach is that this ".weight" is a getter, which turns an "apple" object into a data bag and the entire object paradigm gets ruined. That's why an integration of a functional approach with an object-oriented one, I believe, is not feasible.

• Reply • Share ›

David Raab ➦ Yegor Bugayenko • 3 months ago

Yes, at some point it would just be a data bag and the OO paradigm would be ruined. Well in F# that would be a Record that is just a data bag so that paradigm is just fine in F#. But at the other point if you really want the flexibility to sort Apples based on a selected value, how would you do that without "ruining" OO?

1 • Reply • Share ›

Yegor Bugayenko author ➦ David Raab • 3 months ago

By letting each apple to implement "Comparable<Apple, Apple>" interface. The apple should know how to compare itself to another apple. I'm not sure this answers the question though...

• Reply • Share ›

David Raab ➦ Yegor Bugayenko • 3 months ago

Actually not really. The question was how you could choose the Comparison by yourself. If you implement the interface then you give an apple a predefined comparison, but what is when you want another comparison? In real life i could easily sort apples by size, volume, color or different apple cultivars, the price in a store or by name or a lot of other categories. If you have all data and write an "Apple" web-store you probably also want that your users can do something like that. I also can choose on Amazon how my articles should be ordered, and i also don't have to use the default comparison that the developer thought that he thinks was usefull.

And what i also learned from functional language is that it makes sense that a lot of things can't be compared. In F# you can add Flags like "NoComparison" or "NoEquality" to something, and if you still try to compare something it will just fail. In an OO language you often expect that everything can be compared, but that doesn't really make sense. If you take 10 apples and ask someone on the street to sense. If you take 10 apples and ask someone on the street to please sort it, people will probably ask: "Sort by what?"

Something like an apple don't really have a "default" comparison or a default order. And you also don't really would ask an apple itself to please compare itself to another apple, because an apple can't answer you. And actually a lot of things in real life work in that way. You can't even provide a default sort order of books. Should you sort by name? by the publishing year? the ISBN number? If you go in a bookstore you will never find such an order. Usually books are first sorted by the category then by name. And also the current most popular books are seen first. So you usually have a "Programing" Section and another section for cooking. In an IT bookstore the section are even more detailed and all the Java, C# or C++ books have its own section, and so on.

You can sort movie titles by name, but often you find a table with an order of how much they earned in the cinema. And i think this is something important. Usually a lot of things don't have a natural sorting order. Practical you could say that everything without a number in it don't have it. It also don't make sense that OO programing tells you that every class must have a comparison or an equality.

All of that doesn't mean that you can't have an comparison. You absolutely can have it, but the important thing is to make it clear that things like apple should never have a "default" sorting order. If i see code like

> let sortedApples = sort appleList

or

> let greater = apple1 > apple2

i can't say what the result should be, because what is the natural sorting order of an apple? An apple has none of this. In an OO language you now have to probably start looking at the code how comparison is implemented. It would make more sense if everybody is forced to specify a sorting order explicitly. It also makes the code easier to understand and more declarative.

> let sortedApples = sortByName appleList
> let greaterApple = compareByName apple1 apple2

• Reply • Share ›

coder.slynk ➦ David Raab • 3 months ago

Which is why traditional OOP favors composition. The apple shouldn't know anything about how to sort itself, the person sorting should. A simple factory called AppleSorter could work, but the author hates both utility classes and classes ending with er. Traditional OOP would say to use a derivative of the Factory pattern:

```
class AppleFilter{
    private AppleFilter(){}

    public static final Apple getLargestApple(final List<apple> ap
        // Instantiates, or reuses a comparator that knows how to
        // returns the largest apple
    }


    public static final Apple getSmallestGreenApple(final List<app
        // Instantiates, or reuses a comparator that knows how to
        // returns the smallest green apple
    }
}
```

You now can have what ever subclasses that you want of Apple (Green Apples, plastic apples, apple laptops, etc) and the Filter class is the only one that knows how to filter a list of apples. The apple shouldn't know how to sort itself.

If you want to bring in composition, you would need a custom implementation of List that accepts a Comparable<apple,apple> that would then determine when it should sort.

```
final SortedApples applesBySize = new SortedApples(unsorted, sizeC
final SortedApples applesByColor = new SortedApples(unsorted, colo

applesBySize.get(0);
applesByColor.get(applesByColor.size()-1);
// etc
```

Your comparator could be literally anything and would allow you to sort the same list in different ways. But then you can not consolidate the sorting logic app wide. You would have to store your comparators in global scope or hope that every place an apple is being sorted that the programmer is using the same logic inside their Comparable implementations.

As always, decide what you need most and use it. Don't try to follow these "one size fits all" rules.

• Reply • Share ›

Yegor Bugayenko author David Raab • 3 months ago

You're making a valid point and I don't have an answer. I'm still thinking about it. I will probably write something about it in the future.

For now, it's an open problem for me too.

1 ^ | ˅ • Reply • Share ›

Kanstantsin Kamkou • 3 months ago

Kinda hard to link the example of "get the biggest one without sorting" with the code-piece you put. In case of Sorted you have to sort the Array some day. The .get(0) works with sorted list in both cases. Otherwise .get(25) became nightmare :)

^ | ˅ • Reply • Share ›

Marcos Douglas Santos ➦ Kanstantsin Kamkou • 3 months ago

I think the right choice would be

```
Apple biggest = sorted.max();
```

^ | ˅ • Reply • Share ›

Matthew Phillips ➦ Kanstantsin Kamkou • 3 months ago

my guess: its up to the sorted object. one possible implementation get(0) doesn't bother with sorting, get(max) [ie. get smallest] is the same, however any other get (e.g. get(5)) probably will benefit from sorting then getting. but its up to the sorted object to make this decision not the caller.

1 ^ | ˅ • Reply • Share ›

Kanstantsin Kamkou ➦ Matthew Phillips • 3 months ago

I think that the example is just not complicated enough. Lets expand it a little bit by time-sorting (you have to iterate over objects by default). How you see it? The first I have in my mind is:

```
List<apple> sorted = new Sorted(apples, new SortLogicForApples());
sorted.first();
sorted.at(23);
sorted.last();
```

p.s.: *.get(1) sounds more like get only one object :)

^ | ˅ • Reply • Share ›

Tomáš Beneda ➦ Kanstantsin Kamkou • 3 months ago

How the Sorted class should look like? Should it implement the same interface as given argument (List?) or extend some class (is sorted list an ArrayList?) or provide accessor method (getList()). Sorter is plain and simple - stateless object with just one method.

^ | ˅ • Reply • Share ›

Marcos Douglas Santos ➦ Kanstantsin Kamkou • 3 months ago

Don't do that

```
List<apple> sorted = new Sorted(apples, new SortLogicForApples()
```

If the logic is in another object, what Sorted do?

The second part is OK for me.

• Reply • Share ›

Kanstantsin Kamkou Marcos Douglas Santos • 3 months ago

As a proxy which is holding the behavior methods for a sort logic ("at", "last", "first", etc.). These are just my thoughts.

1 • Reply • Share ›

Andre Bogus • 3 months ago

A good point. If you have a bag of Utility methods, at least name them "BagOfXyUtilities". :-)

That said, *interfaces* can very well be named whatev-er (or something-or). Listener, Logger, Visitor, Iterator, you name it.

• Reply • Share ›

Will Sherwood • 3 months ago

you make a great point, however you don't practice what you preach. Looking through your work on JCABI, more than 10 classes out of that project end in -er. MethodCacher, MethodInterrupter, MethodLogger, MethodValidator, Parallelizer, QuietExceptionsLogger, Repeater, AsyncReturnTypeProcessor, etc. Maybe some refactoring can happen in this project. Also, why not rename @Cacheable to @Cached and @Timeable to @Timed? It seems that the Cacheable interface has a strict contract that caches the return value. The word 'cacheable' does not capture this statement: the suffix -able to me at least seems flimsy, like it can cache, but it doesn't have to. @Cached would then mean that this result WILL be cached and there is no doubt about that.

2 • Reply • Share ›

Yegor Bugayenko author Will Sherwood • 3 months ago

You're right, some of the classes in my own projects require refactoring. The more you develop, the more you learn :)

2 • Reply • Share ›

Marcos Douglas Santos Yegor Bugayenko • 3 months ago

Hmmm... :-\

• Reply • Share ›

Oleg Majewski Marcos Douglas Santos • 3 months ago

no surprise, with that blogs Yegor gets a lot of feedback for himself :) you never stop to learn ;)

1 • Reply • Share ›

Roman1337 • 3 months ago

StringBuilder, Scanner, BufferedReader, ...Why would I do it differently from Oracle?

• Reply • Share ›

Christian Hujer Roman1337 • 3 months ago

Because it's time for revolution. It's time for the verbs to demand rights of their own and start rebellion in the kingdom of nouns.

1 ^ | ˅ • Reply • Share ›

Marcos Douglas Santos ↪ Christian Hujer • 3 months ago

:)

^ | ˅ • Reply • Share ›

Tomáš Beneda ↪ Roman1337 • 3 months ago

Well BufferedReader should be just Buffered wrapper (ouch, another -er!). But what about StringBuilder? IntermediateString? Does the name really matters?

^ | ˅ • Reply • Share ›

Oleg Majewski ↪ Tomáš Beneda • 3 months ago

BuildableString

What is a BufferedReader, just looking at the name no idea what it does, it reads what?

After looking at java doc: Reads text from a character-input stream

aha, ok, then what about BufferedReadableText ?

>Does the name really matters?

YES, it does!

^ | ˅ • Reply • Share ›

Tomáš Beneda ↪ Oleg Majewski • 3 months ago

Thanks, that makes sence.

BufferedReader has always some Reader as constructor argument - so for me the Buffered is good (and short) enough.

^ | ˅ • Reply • Share ›

Andre Bogus Tomáš Beneda • 3 months ago

That won't work, because BufferedReader has to be a Reader. And there is also BufferedWriter, BufferedInputStream and BufferedOutputStream. It's the classical decorator pattern that Yegor loves so much (and which ends in -or, btw.)

Now with Java 1.7 you could `import static java.io.Reader.buffered` and make the class an inner class of Reader, but that's bad for discoverability.

Same argument goes for the Builder pattern — notice how it ends in -er?

• Reply • Share ›

Oleg Majewski Tomáš Beneda • 3 months ago

and what is reader? it reads what? ;-)

1 • Reply • Share ›

Oleg Majewski Roman1337 • 3 months ago

why not?

1 • Reply • Share ›

Christian Hujer • 3 months ago

It's so true, and let me add a few things. When you have a Sorter, you have a List which is sorted after you used the Sorter, but as soon as you modify the list, it's probably no longer sorted. You're in a trap of temporal coupling. I like this example very much because it demonstrates how OOP (not in the sense of polymorphism for dependency inversion, but in the sense of self-thinking objects) is incomplete without proper use of FP. Same goes for Reader / Writer. Their problems are the obvious temporal coupling between open() and close(). Looking at Listener / Observer, these are the kingdom of noun's poor attempt to mock function pointers. But they're all about operation, not state. They're procedures (functions, methods, operations, call it what you like), but not classes. P.S.: And -er or -or probably doesn't make a lot of difference, we should look out for those -or as well. Maybe not all of them, but many of them.

3 • Reply • Share ›

Denis Christian Hujer • 2 months ago

so Sorted should be immutable object then. it should keep its sorted state.
but what if you change the object inside the Sorted? then it should either rebuild itself or when you call getBiggestApple() it should do a sort again

• Reply • Share ›

Christian Hujer Denis • 2 months ago

As a collection, you probably don't want it to be immutable. But you probably want the fact that it's in sorted state to be immutable. Plus, when it's sorted, you probably are not interested in random access, like accessing the nth element, because due to sorting you wouldn't know what the nth element would be. So you would want a `Set`, not a `List`, and you would want a `SortedSet`, which keeps its elements in sorted order no matter how many you add and remove. There are enough data structures out there that can retain their sorted state whenever you change the contents. But that all again comes down to a certain degree of immutability. The fact that an apple is no longer the biggest should be possible because other apples got removed and added, but an apple itself shouldn't change its size. Disclaimer: that's not applicable in all situations.

• Reply • Share ›

Marcos Douglas Santos Christian Hujer • 3 months ago

Perfect!
Now let's wait **@Yegor Bugayenko**'s comments. :)

1 • Reply • Share ›

尤川豪 • 3 months ago

Nice point. Very inspiring. Thanks!

• Reply • Share ›

Paddy3118 • 3 months ago

Validator? It ends in "or"?

What about escalator, resistor and governor?

More from yegor256.com

yegor256.com recently published

How to Implement an Iterating Adapter

13 Comments Recommend

yegor256.com recently published

How Cookie-Based Authentication Works in the Takes Framework

11 Comments Recommend

yegor256.com recently published

Constructors Must Be Code-Free

68 Comments Recommend