Composable Decorators vs. Imperative Utility Methods

The decorator pattern is my favorite among all other patterns I'm aware of. It is a very simple and yet very powerful mechanism to make your code highly cohesive and loosely coupled. However, I believe decorators are not used often enough.…

(yegor256.com)

Sihle • a month ago

Yegor, do you think Java have got it wrong again with the Stream API, where methods can be chained like the String object utility methods? With the Stream API you could have stream.map().filter().map.... an so on. Is this against the spirit of composable software?

Yegor Bugayenko author ➚ Sihle • a month ago

Yes, I think it's a wrong design. Instead of this:

```
4 == Arrays.asList("jeff", "walter")
  .stream()
  .filter(name -> name.startsWith("j"))
  .map(name -> name.length())
  .sum();
```

Would be better to do this:

```
4 == new Sum(
  new MapTo(
    new Filter(
      Arrays.asList("jeff", "walter"),
      (String name) -> name.startsWith("j")
    ),
    (String name) -> name.length()
  )
);
```

1 · Reply · Share ›

David Raab ➦ Yegor Bugayenko · a month ago

Seriously? Your version improves nothing and introduces a lot more problems. Your version has the problem of the "Pyramid of Doom" (http://survivejs.com/common_pr...). And the logic is even reversed. You need to read it from the inside to the outside. And the whole API comes by the way from functional languages. Sure it is not OO, and i think once again that is the only reason why you dislike it, and you even would use a bad OO version with Pyramid of Doom and reversed logic.

I mean your version is so hard to read. it gets even hard to read where the line "(string name) -> name.length()" belongs to. And every time you want to add something to it you have to indent your code more and more, and things that belong together gets even more splittet over different lines. While the functional Stream version is easy to read. Has no indention problems, and you can read it step by step from top to down. It is even hard in your version to see on which data the whole thing starts.

A Functional version in F# would look like

```
let sum =
    ["jeff"; "walter"]
    |> List.filter (fun name -> name.StartsWith("j"))
    |> List.map (fun name -> name.Length)
    |> List.sum
```

just as an example to see where the Stream API has its origin. And thank goodness, it is absolutely not an OO API.

· Reply · Share ›

Suseika · 2 months ago

There is one thing about decorators I can't seem to understand. We decorate objects in order to create behavior that can be derived from the decorated object. But do we really need that behavior to adhere to the interface of the decorated object?

For example, in Takes framework there is a class RqMethod that decorates an HTTP request by providing method `String method()` that tells the HTTP method used. But why

do we need RqMethod to be an instance of Request? Is it just so that the implementation of RqMethod has access to the behavior of the decorated object as if it is its own behavior? I can't figure out why a user of RqMethod would need any of RqMethod's behavior other than the `String method()` method.

What is the reason for extending RqWrap? I think RqMethod should be implemented like this so it has just the interesting behavior:

```
public final class RqMethod /* extends nothing */ {

private final Request req;

public RqMethod(final Request req) {
this.req = req;
}

public String method() throws IOException {
final String line = req.head().iterator().next();
final String[] parts = line.split(" ", 2);
return parts[0].toUpperCase(Locale.ENGLISH);
}

}
```

1 ▲ | ▼ • Reply • Share ›

Yegor Bugayenko author ➚ Suseika • a month ago

Good question. If RqMethod would not implement Request interface, we won't be able to pass it where a Request is expected. We will have to pass them together, an instance of Request and an instance of RqMethod. It is more convenient to "inherit" the interface.

1 ▲ | ▼ • Reply • Share ›

Suseika ➚ Yegor Bugayenko • a month ago

But why pass them together? If a client of a Request needs an RqMethod, the client can instantiate an RqMethod himself — the client already has the necessary request. Passing RqMethod to a Client1 basically says that Client1 needs to know the method of a Request — but that's implementation details! All client needs in a plain Request. Client2 whose client is Client1 shouldn't concern himself with the implementation of Client1, should he?

▲ | ▼ • Reply • Share ›

Yegor Bugayenko author ➚ Suseika • a month ago

Not really. If we instantiate RqMethod every time we need it, we won't be able to control performance. We want an object to live for as long as possible, to give him an ability to preserve its local data and re-use them.

▲ | ▼ • Reply • Share ›

Marcos Douglas Santos ↗ Yegor Bugayenko • a month ago

We talk about this "problem" and you said that would refactor this. "Convenient" is not a good argument if you are building something pure object-oriented, don't you think? Maybe you have a problem in your design.

• Reply • Share ›

Yegor Bugayenko author ↗ Marcos Douglas Santos • a month ago

We're already refactoring our code, to make it more compliant with "all public methods are coming from interfaces" principle, thanks to you for noticing this problem earlier. But this is not relevant to the question asked above.

1 • Reply • Share ›

David Raab • 2 months ago

Actually when the article first came out i wanted to write something about it, but i knew it would be longer. Now i found some time. At first i want to say that i don't mean anything aggressively or try to insult you. But often that happens when some people have completely other opinions. Actually what i see in a lot of your articles is the absolutely OO-fanaticism. Well, i don't meant it as an insult, but you often have the believe that everything that is not OO is automatically bad. And that is not the case. Directly from your beginning for example you say that Utility methods are procedural and thus bad.

But that is also not true. At first. Also Utility methods can be completely OO. You can even program OO in a language like C that doesn't support classes at all. OO is an idea not some semantics and keywords in a language. I just can say that you should look more into a functional language or just looking into how you can program OO in C, because it will really widen your view of what is OO and what is not. Especially the last part is important, if OO is not some semantic and you can program OO in C, does that mean you also can program "not OO" in an OO language? Absolutely. And what you have shown so far is exactly that.

Well at first, let me say that i don't say that decorative Patterns are bad or something like that. But how you used it here doesn't make anything more OO. Lets for example view your first two decorators. It is named "TextInFile" and "PrintableText" as classes. You later use it like that

```
final Text text = new PrintableText(
    new TextInFile(new File("/tmp/a.txt"))
);
String content = text.read();
```

Now lets assume you would have written two static methods instead, it would look exactly the same (Sorry, im not a Java guy, so my fault if this is not completely valid Java code, but i guess you can see here what is important)

```
final String content = PrintableText(
    TextInFile(new File("/tmp/a.txt"))
);
```

You even write a senctence like

> As you can see, the PrintableText doesn't read the text from the file. It doesn't really care where the text is coming from. It delegates text reading to the encapsulated instance of Text. How this encapsulated object will deal with the text and where it will get it doesn't concern PrintableText.

Actually the exact same is true, for the implementation of the static methods. Does the static method "PrintableText" care where the Text is coming from? No. Does it read the Text from a file? No. It also delegates the text reading to the encapsulated method of TextInFile. And how this method deal with the Text and where it will get it also doesn't concern the PrintableText static method. This is just one example. In fact absolutely everything that you describe in your article is also true for static methods.

You also can compose absolutely any static method. In fact composing it is even easier. Composing it are the fundamental of functional programing. You always can compose any function if the output of one function is the input of another.

If you have a function that takes an A and returns an A you can compose any function with the same signature in any order. You highlight your composability for example with this code.

```
final Text text = new AllCapsText(
  new TrimmedText(
    new PrintableText(
      new TextInFile(new File("/tmp/a.txt"))
    )
  )
);
String content = text.read();
```

But the whole stuff also works with static methods

```
final String content = AllCapsText(
  TrimmedText(
    PrintableText(
      TextInFile(new File("/tmp/a.txt"))
    )
  )
);
```

Your code is absolutely not more or less composable than static methods. But i said before that composability can even be easier. Why? Because your composability is really hidden and splittet in parts. When i see a function i just need to knew the inpout type and the output type and knew if it can compose. If i have a function signatures like

```
String TrimmedText(String source)
String PrintableText(String source)
String AllCapsText(String source)
```

I easily can see that i can compose those methods. I can compose those functions in any order because all of them takes a String and returns a String. But can i also compose "TextInFile" in there in any order? Can i for example do

```
TextInFile(TrimmedText("foobar"));
```

Absolutely not, because the TextInFile needs a "File" object as an input. But can you compose those classes together? Also "TextInFile" implements the "Text" interface, but can you do that with your code? Also absolutely not. Because your "TextInFile" class also needs an "File" object. So even that your "TextInFile" class implements the "Text" interface like "PrintableText" do it, you still can't use the "PrintableText" object as an input for TextInFile. Even if both implements the same interface. You can compose them, but only in a specific order.

The important stuff for composing is always the input and output. In your Decorator example the input of TextInFile is always the type of the constructor while the output is the return type of the "read" method. Only if those matches you can compose it in any order. If it doesn't match you have to fullfill the exact order. Sure that is also true for a static method. But in a static method those information are not split into different locations. You can see them directly.

```
String TextInFile(File source)
String TrimmedText(String source)
String PrintableText(String source)
String AllCapsText(String source)
```

In code like this i can see it directly that i cannot use the output of TrimmedText, for the input of TextInFile. Exactly like your code can't do it. But the difference is it is also never suggested that it is able to do it. In your version all classes implements the Text interfaces and you still have the exact same restrictions. Nothing in your version makes it "better composable" or "more composable" as static methods. You also don't win any more abstraction or your code is more declarative. A function named "TrimmedText" is exactly the same amount declarative as a class named "TrimmedText".

And your code is also not more OO. Your classes takes an input and gives an output. The only thing you did is re-implementing the idea of a simple static method in an OO-mess. Putting it in a class, with a constructor and an interface and a method absolutely doesn't make it more OO, your code is exactly the same procedural or not OO as a bunch of static methods. It seems to me that you look to much at semantics. Just using classes, methods, constuctors or interfaces absolutely makes no code more OO. I absolutely can say that you should look into how you can program OO in a language like C. Because if you understand it in a language like this you will also see that your example is absolutely not more OO.

And even if something would be OO, or some other code is procedural. It doesn't make it

And even if something would be OO, or some other code is procedural. It doesn't make it automatically better. OO is just a paradigm like procedural, functional, AOP, event based, asynchronous, reactive, ... and so on. OO is not the holy grail. Not being OO is not bad, it is just not OO. For some problems an OO solution can be better, for others a functional or procedural solution can be better. And this is a really important point. Because if you lose those black/white mentality you really have to think about arguments.

You for example have to explain why you think that

```
final Type content = new X(new Y(new Z("Foo")))
```

is better than

```
final Type content = X(Y(Z("Foo")))
```

just saying that one solutions is OO (what it is not), and the other is not, is absolutely not an argument. It is a lame excuse to not really think about the problem at all.

1 ▲ | ▼ • Reply • Share ›

Esteban • 3 months ago

Hi Yegor. Great article. I'd like to ask what's the difference between a helper utility class with lots of static methods and the facade design pattern.

▲ | ▼ • Reply • Share ›

Yegor Bugayenko author ➦ Esteban • 3 months ago

Well, they are different, but they have a lot in common. Facade, if used wisely, may look OK in terms of OOP. It is rarely being used that way, unfortunately.

▲ | ▼ • Reply • Share ›

coder.slynk • 3 months ago

I've figured it out. Your problem. You seem to think of yourself as some sort of master of OOP, yet you forget about one of its most important concepts... encapsulation. In all the articles I've read from you, you seem to completely toss out encapsulation in favor of decoration.

What happens when you want to ensure the same logic is applied to multiple objects across various screens of interaction from the user?

Lets say you have 3 screens and on each screen you want all names be proper cased, with last name truncated to a single letter with a dot on the end, and have the whole thing ellipsized if it's greater than 13 characters. These names are supplied to you from a third party, lets say facebook, so you can't have the server send you the names already formatted. You receive these names and must pass them as strings into a text view to be displayed. The text view's method setText() cannot me extended as the text view is a final class. So, you, as I understand this, would do the following:

Text formatted = new EllipsizedText(new TruncatedText(new ProperCaseText(name)));
view.setText(formatted.read());

You would then copy paste your solution. As the project moves on, names are inevitably

placed in more locations. You're now up to 10 screens, but that's fine because trusty ctrl+c and ctrl+v are still working. Now your designer wants to try out a new style, all lower case names. You now start sifting through your code trying to find every occurrence of a name so that you can change the casing. Did you find all of them? Did you properly change the logic in every occurrence? Is there an easier way to ensure that the logic is consistent and easy to alter in the future?

That's where encapsulation would have saved you. Any of the following would have been a preferable solution:

Text formatted = new NameText(name);
view.setText(formatted.read());

or, and I know you hate this one:

view.setText(StringFormatter.formatName(name));

Either way you now have a single function to alter in the future if your designer changes his mind. You *know* that all names in your application are consistently formatted, so long as everyone uses your single universal function / class. You don't have to worry about whether someone did the ellipses before the trim, or forgot the trim all together.

1 ▲ | ▼ • Reply • Share ›

Yegor Bugayenko author ➚ coder.slynk • 3 months ago

I didn't get why copy-and-paste was chosen as a solution? Everything was good, until that turning point. If you "copy and paste", there are no more design principles could be applicable. Let's try again, but without copy and paste :)

▲ | ▼ • Reply • Share ›

coder.slynk ➚ Yegor Bugayenko • 3 months ago

How do you avoid copy paste without encapsulation?

▲ | ▼ • Reply • Share ›

Yegor Bugayenko author ➚ coder.slynk • 3 months ago

I don't really see any connection between these two things. Can you show the code? The code above is good, I would do it even better:

```
Text formatted = new EllipsizedText(new TruncatedText(new ProperC
view.setText(formatted);
```

▲ | ▼ • Reply • Share ›

coder.slynk ➚ Yegor Bugayenko • 3 months ago

Right, and you would do that in 10+ locations (essentially copy pasting, even if you re-type the solution.) If you then had to change the format of all names in your application, you would have to change it in 10+ locations. If you encapsulated the decoration into a utility class that you hate so much, you would have a single place

that needs to be edited that would affect all names in the app.

With *your* solution, you'd have to hunt down and change multiple instances of the same code and hope that your edits were all correct. Your code is harder to maintain, and less descriptive. Your example doesn't make it clear that the reason you are applying 3 transforms to the string is because it's a name and that's how names are *supposed* to be formatted. For all an outside observer knows, that's just how you felt like formatting it in this one instance.

StringFormatter.formatName(name);

While this is a utility class, you can tell from the name that it's a class used to encapsulate string formatting for the application. You know that what ever formatName() does, it is used to properly format names. You don't need to know what format name does. It could use decoration in the background, it could use a for loop and manually edit every character individually. It doesn't matter. All you should care about is that your new name field is properly formatted. You shouldn't have to remember the exact logic for formatting a name every time you need to do it.

1 ▲ | ▼ • Reply • Share ›

Yegor Bugayenko author ➦ coder.slynk • 3 months ago

Aha, I understood what you talking about. The solution here is that you should introduce a new class, which will represent a properly formatted name for your application. Something like this:

```
public class NiceText implements Text {
  private final Text origin;
  NiceText(Text txt) {
    this.origin = new EllipsizedText(new TruncatedText(new Proper
  }
  String read() {
    return this.origin.read();
  }
}
```

See the idea?

▲ | ▼ • Reply • Share ›

coder.slynk ➦ Yegor Bugayenko • 3 months ago

Then you would need a separate class definition for every type of formatting, each with a unique and meaningful name. In the end, the only difference between encapsulating the logic in a utility function vs a decorator class is that the decorator class performs more poorly. It adds additional object allocation and in some languages would expand the vtable. A decorator class constructor is *still* a

function, whether you want to view it as one or not. A constructor is simply a function that returns a new object allocation. A utility function can also return an object. Your rules are arbitrary and hurt the performance of your applications.

coder.slynk ➚ coder.slynk • 3 months ago

And a small follow up. I'm sure you're thinking object allocation and extra class definitions... who cares these days? We, as of a month ago, ran into Android's function definition limit. More info on that:

https://www.facebook.com/notes...

Also, Android specifically contradicts everything you recommend. It tells you to use static methods and avoid object allocations. Your views are unrealistic and impractical to real world programming:

https://developer.android.com/...

Yegor Bugayenko author ➚ coder.slynk • 3 months ago

Check this article, it explains my view on this "practical vs perfect" problem: http://www.yegor256.com/2014/1...

coder.slynk ➚ Yegor Bugayenko • 3 months ago

"That's why, while developing a new software product, those who pay for it will care mostly about its maintainability. Project sponsors will understand that the best solution they can get for their money is the one that is the most readable, maintainable, and automated."

But your solution *doubles* at the very least all function definitions in the app. Your solution adds a read method *and* a constructor; where as a utility class would only add a single function for every format. You also add overhead to the entire application. In a couple instances, that may not mean much. But when applied as a coding standard, as you seem to hold your views, that means that if you were on my team and we hit the function limitations on Android, we'd be screwed. Your inefficiency would be tied into the entire application. We were able to eliminate a few external libraries to fix our issue; that wouldn't be possible with you. We'd have to refactor the entire application.

You talk about the differences between a hacker and a designer. I would never consider someone to be a designer who does not architect their solution for the platform at hand. Your views, while maybe a good starting place for general good practices, are impractical and detrimental to a lot of platforms, Android included.

You seem the think there's some masterful catch all solution that can be applied to any project regardless of environment, usage, scalability, etc. A designer is adaptable and understands what corners can be cut while keeping integrity in their application. A designer understands the limitation of the tech stack they are working with and ensures the solutions they invent work within their eco system.

1 ^ | v • Reply • Share ›

Yegor Bugayenko author ➚ coder.slynk • 3 months ago

How about placing all our code into one place? We can have just one big "Application.java" file with a single 50000-lines "main()" method inside. It will work much faster than all that utility classes. We will eliminate all constructors, all method calls. It will be a super effective and fast software product. Android team will be proud of us. What do you think?

2 ^ | v • Reply • Share ›

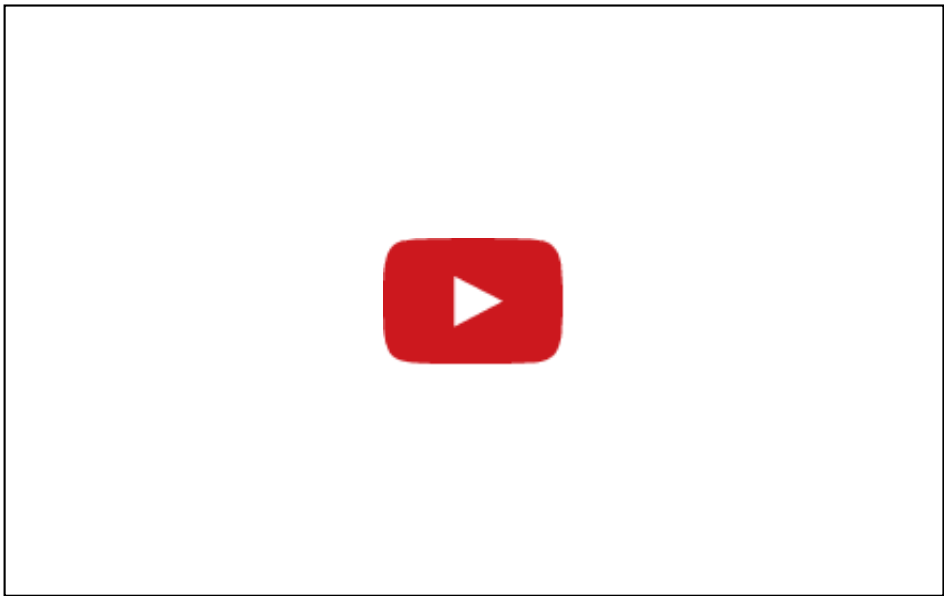
Oliver Doepner ➚ Yegor Bugayenko • a month ago

Yegor: It does not reflect positively on you, when you try to ridicule a valid point (inefficiency due to exceeding object allocation and GC, and the code overhead of "class with constructor and method instead of just a method") by drawing up an extreme that nobody proposed (single class for the whole application). This is called "setting up a strawman".

^ | v • Reply • Share ›

coder.slynk ➚ Yegor Bugayenko • 3 months ago

By the way:



^ | v • Reply • Share ›

coder.slynk ➚ Yegor Bugayenko • 3 months ago

Just as tossing all your code in one place would be bad, so is

splitting your 50000 lines of code into 50000 classes. Programming requires balance. You are literally recommending that people create an entire class to encapsulate a one line function. If you don't see that as a problem, then I'm not sure how to convince you.

• Reply • Share ›

Michael Hay • 3 months ago

final String[] parts = new String.Split(
new String.UpperCased(
new String.Trimmed("hello, world!")
)
);

Problems with this idea:
- String.Split cannot subclass String[], so it could never work.
- you have to create a new object on every call, which is less efficient than the current methods (String.trim and toUppercase both currently `return this` when possible)
- you're exposing the concrete type of all these decorators, and String.UpperCased and String.Trimmed presumably subclass String, making String non-final.

- it's massively verbose

Doing this just so your code is "more object-oriented" seems like classic cargo-culting.

1 • Reply • Share ›

John Fielder • 3 months ago

Damn, now that I've properly learned the use of the Decorator pattern, I can't use any web framework simply because every single one of them forces me to break lots and lots of OOP principles and leads to buggy, ugly and inflexible code. I started looking at your Takes framework, however I don't really like the fact that all the class names are abbreviated. With the advent of IDEs, I think class names should be full words, especially when they wouldn't even be that long. I'm looking to maybe create a fork of the Takes framework, I just love how short and concise everything looks, however I can't figure out how you would unit test an entire application (by which I mean mocking different parts of it without creating an entire application context). Also, is there any particular reason why there will be no support for WebSockets?

1 • Reply • Share ›

Yegor Bugayenko author ➦ John Fielder • 3 months ago

I totally understand your frustration with existing frameworks. They indeed are far from being object-oriented. That's why I decided to create Takes. Thanks for trying it out! If you have suggestions about it or corrections, don't hesitate to submit tickets into its Github issue tracker: https://github.com/yegor256/ta... Regarding the class names, I abbreviated them on purpose, to hide less important information and emphasize what's important. For example, TsFork means "Takes that fork". Fork is important while "takes" is just a prefix. You will get used to it :)

You can see how an entire application can be test in Rultor source code. Rultor is

You can see how an entire application can be test in Rultor source code. Rultor is the first real project that is using Takes and it's fully tested. See its test classes here: https://github.com/yegor256/ru...

• Reply • Share ›

John Fielder ➚ Yegor Bugayenko • 3 months ago

Thanks a lot for the help! I really hope your framework will be used by the community and gets features added to it, I'd love to see it some day become as popular as the majority of the current hacky frameworks. BTW, do you have any books to recommend?

• Reply • Share ›

Yegor Bugayenko author ➚ John Fielder • 3 months ago

I usually recommend Object Thinking by David West. I think it's the best book about object-oriented programming and the entire object paradigm. If you're serious about OOP, you must read it.

1 • Reply • Share ›

Marcos Douglas Santos ➚ Yegor Bugayenko • 3 months ago

+1

Object thinking is the best book about OOP.

• Reply • Share ›

Andrzej Ludwikowski • 3 months ago

I'm a big fan of Decorators, but you chose really bad example, because decorated version of "txt.trim().toUpperCase().split" is not readable and very inconvenient to use.

• Reply • Share ›

bryo • 3 months ago

Is it possible to create this pattern decorator with a fluent API ?
It could be used like that in the exemple :

```
final Text text =
new TextInFile(new File("/tmp/a.txt"))
.allCapsText()
.trimmedText()
.printableText();
String content = text.read();
```

• Reply • Share ›

Alexey Dmitriev ➚ bryo • 3 months ago

You might want to take a look at RxJava, and more specifically, to its compose() function which can be feeded with Transformer's subclasses. More details in the blog post here: http://blog.danlew.net/2015/03...

With the help of it I switched from using Decorators to fluent interface. For example:

return new TopTracksCommand(...)

```
.observe()
.compose(new CachedTrackTransformer(locallyCachedTracks))
.compose(new ProgressiveTracklistTransformer(...))
.compose(new PriorityScheduler(...));
```

• Reply • Share ›

Oleg Majewski ➚ bryo • 3 months ago

technically you can, but this would violate the single responsibility principle how would your TextInFile class looks like? ;-)

• Reply • Share ›

Oleg Majewski • 3 months ago

good article, not far away and you have clojure code :)

• Reply • Share ›

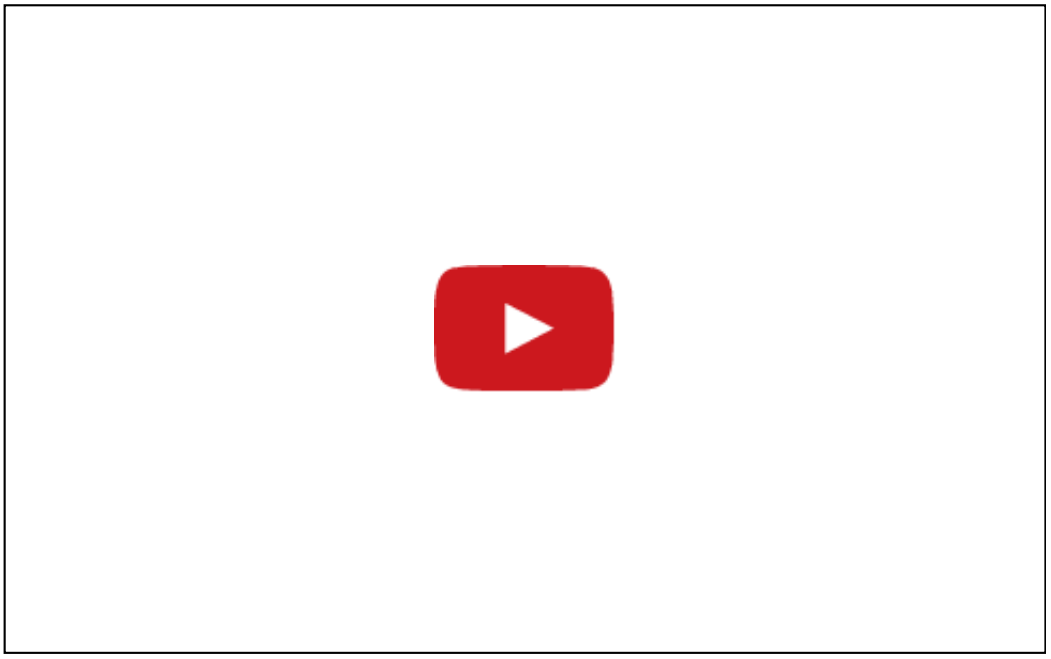
Yegor Bugayenko author ➚ Oleg Majewski • 3 months ago

Indeed, this approach is very close to functional programming, because it is also declarative. I don't like Clojure for its crypted and difficult to digest syntax, but I like functional programming a lot. Not just functional, but declarative. Logical programming is also a great paradigm. So, no surprise the code above reminds Clojure to you :)

• Reply • Share ›

Oleg Majewski ➚ Yegor Bugayenko • 3 months ago

yepp clojure is very hard to read if you see it first time everything is strange, check out this:



• Reply • Share ›

Marcos Douglas Santos • 3 months ago

This article came in good time.

I would like help for this problem bellow, using immutable objects and decorator pattern.

Imagine this scenario:
- there is an Agenda;
- Agenda has Days;
- each Day has Hours;
- you have Hour too.

So, every things works and I have tests for all classes and behaviors (that I do not need expose here for simplifying).
I have three simple objets (Agenda, Day and Hour) and two lists (Days and Hours).

Days could be add more day inside yourself and Hours the same with hour.
But now I want to persist these objects (Agenda->Day->Hour). I would like to use the Decorator pattern so (pseudo-code):
class SimpleDay implements Day;
class SaveDay implements Day;
day = new SaveDay(new Day(d));

and so on.

Using a DB, if I called agenda.days.add(5) the code will persist:
- 5 days
- for each day I will have hours (9am to 6pm)

Questions:
1- Using immutable objects, ie, each object is initialized on constructor, how can I decorate days list (inside agenda), hours list (inside a day) to persist these objects?
2- Considering you said the most GoF's patterns are anti-patterns, how the program will know the types of objects that will use when need to instantiate (SimpleDay, SaveDay)?

Yegor Bugayenko author ↗ Marcos Douglas Santos • 3 months ago

Too many questions here, I'm not sure I can answer them. Can you try to simplify it?

Marcos Douglas Santos ↗ Yegor Bugayenko • 3 months ago

Yes, I'll try.

When an Agenda is created, a Days list is created and belongs to Agenda. Each Day of this list creates other list, ie, Hours list that have many Hour objects.

The question is:
Using no DI, none of GoF patterns, no heritage, considering all objects as immutable and everything is created on constructors of these objects, how can I "intercept" the creations of these lists (Days and Hours) to decorate them to use "save classes" -- they work with DB and SQL -- considering these lists are created inside of Agenda (Days list) and Day (Hours list)?

In others words, if you didn't understand yet, is:
How can I decorate objects that have others objects, returning them by methods?
Eg:
// get the first hour of the first day
h = agenda.days().get(0).hours().get(0);

So how can I decorate days and hours lists, instances of SimpleDays and SimpleHours respectively, -- that only have the business rules -- to use SaveDays and SaveHours, to use database and SQL?

Is that clear now? :)

• Reply • Share ›

Yegor Bugayenko author ➚ Marcos Douglas Santos • 3 months ago

You have to decorate the Agenda. You should create SqlAgenda and it will create SqlDays and they will create SqlHours. Take a look at these classes, for example: https://github.com/jcabi/jcabi...

1 • Reply • Share ›

Marcos Douglas Santos ➚ Yegor Bugayenko • 3 months ago

Thanks for the examples.

Well, I saw you recreate, decorate and return new objects every time the methods are called.
In your examples, I didn't see methods that return new lists. But using your approach if I have day.get(0).hours() and I need to decorate hours list, everytime the program will need this list it will be recreated? Ie, needs a loop in all itens, recreate all hours objects and return a new list? Am I correct?

At the first time I wrote my code using the same way you did but I'm afraid about performance. So, after, I've changed that and I added new attributes for the decorators, to keep the new lists (decorated lists). But this is not immutable... then I'm thinking how can I "intercept" the construtors -- to use another classes -- or if is better to use something like Abstract Factory pattern. But I don't like this

More from yegor256.com