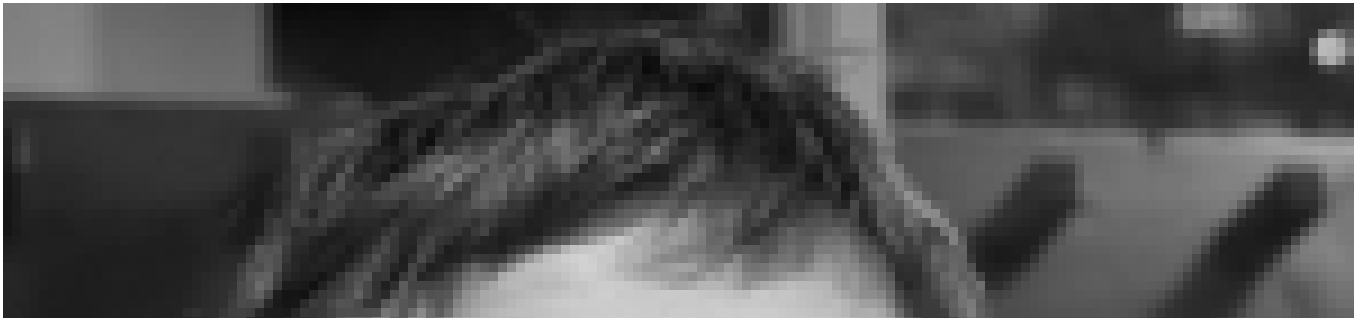




Community

 yegor256.com

55 Comments · Created 8 months ago



DI Containers are Code Polluters

While dependency injection (aka, "DI") is a natural technique of composing objects in OOP (known long before the term was introduced by Martin Fowler), Spring IoC, Google Guice, Java EE6 CDI, Dagger and other DI frameworks turn it...

[\(yegor256.com\)](#)

55 Comments

 Recommend Share

Sort by Newest ▾



Join the discussion...

**andyg8180** · 6 days ago

What if person A wanted to use Tweets() but then person B wanted to use a new class called SuperTweets(). How would you handle that in the code above?

 |  · Reply · Share ›**Yegor Bugayenko** author → andyg8180 · 3 days ago

Then you create another instance of agent, with SuperTweets() inside.

 |  · Reply · Share ›**Timo Reitz** · 2 months ago

Although I agree with most of your article, I disagree with your conclusion. You ask "How does that help us?", I want to try to answer that.

Let's assume you use an DI container, which allows you only to set parameters, a combination of a name and a primitive value, and to set services by assigning a tuple (classname, array of arguments) to a name. Of course it also allows you to retrieve a

(classname, array of arguments) to a name. Of course it also allows you to retrieve a service. Now instead of using a bunch of new operators, you define the relationships in the container, retrieve the top-level instance from it and runs it.

What are the consequences?

You lose compile-time type checking, but if there are end-to-end tests in place, this should not be a problem.

You have to name every parameter and service. Although this means more typing, it also helps in clarification. Look at that number 49092213 in your example - what does it do? With the container in place, you would have to give it a name.

It also helps in describing the application's structure in a declarative way, especially if using XML, YAML, JSON (or whatever is fancy at the moment) instead of Java code.

I also think your example is misleading. While there are certainly applications with a strict hierarchical structure, the general case is a directed acyclic graph, which cannot be mapped that easily to a bunch of nested new expressions.

^ | v · Reply · Share ›



Yegor Bugayenko author → Timo Reitz · a month ago

Yes, you're right, a directed acyclic graph (aka spaghetti code) is what we get when we use DI containers. A properly designed application should have a strict hierarchical structure of objects, encapsulating each other.

^ | v · Reply · Share ›



hdn · 3 months ago

I agree. DI containers are unnecessary. But your code could be a lot more readable if you employ the Builder pattern. We have recently replaced all of our Spring setup with Builders. We still have all of loose coupling benefits with the container, but the code is now much easier to manage, test and debug.

^ | v · Reply · Share ›



John Fielder · 3 months ago

I completely agree with what you've said here, except with the final code example. To me it is pretty unreadable because of Java syntax (maybe in a different language it should be much more readable, or maybe with more spaces and comments in between the construction lines). I've always thought DI containers were used to describe how an object should be constructed: for example, design an application like you have here, and then just describe the creation of objects in an XML file. That way, you have (at least to me) easier to read configuration, avoidance of anti-patterns, and a centralized configuration file that could easily be updated. What are your thoughts on this?

^ | v · Reply · Share ›

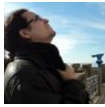


Yegor Bugayenko author → John Fielder · 3 months ago

For me this code looks very clean and very easy to understand, mostly because I like composable decorators (see this post: <http://www.yegor256.com/2015/0...> and use them very often. And XML configurations looks very counter-intuitive to me and I can't read them, including Spring XML configs. So, I believe it's a matter of habit... or a mindset. Try to use composable decorators more and you'll see how easily-

readable this code will become :)

1 ^ | v · Reply · Share ›



Guido Contreras Woda · 3 months ago

Hi Yegor! Thanks for sharing this.

I have a few (namely 2) problems with your suggested way of designing / constructing applications, and I'd love to have some feedback from you on them:

1. This huge [Agents.java](#) file has too many reasons to change. If any object in the system changes its dependencies, this file needs to reflect that. And, being the main point of entry, this makes every single change to any object a very risky one, as the "most important" file on the system will also change as well.

2. When I see this [Agents.java](#) file, I don't care much about "beauty" or "readability". My first thought was "the programmer that maintains this file has to have a very deep knowledge of the whole project." This is why I feel it's very complicated in terms of maintainability. I understand (and agree with) your preference on small projects, but this puts a very high barrier on new devs, no matter how small the project is.

But maybe the problem is not with DI containers, but with encapsulating the creation responsibility. Maybe somewhere in the middle there's some creation pattern missing, or maybe Java DI containers are missing some way to bind object creation to your own factory / builder objects.

Cheers!

1 ^ | v · Reply · Share ›



Yegor Bugayenko author → [Guido Contreras Woda](#) · 3 months ago

Yes, this "[Agents.java](#)" file is basically a map of the entire application. It defines what the application consists of and how its components are composed together into a single solid unit. Someone in the project has to be able to understand this composition. An architect, I guess. And this composition is in a single file. Once someone, even a newbie, wants to get an idea of how the app works, from the highest level of abstraction, the file is there. In case of DI containers, one has to investigate multiple files and still the entire idea will be vague.

^ | v · Reply · Share ›



Lubos Krnac · 4 months ago

Which DI containers have you been using and how long? Have you heard about Spring's concept of component scanning?

^ | v · Reply · Share ›



Yegor Bugayenko author → [Lubos Krnac](#) · 4 months ago

I tried Guice and Spring DI. Not for a long time. Yes, I'm aware of Spring component scanning.

^ | v · Reply · Share ›



Lubos Krnac → [Yegor Bugayenko](#) · 4 months ago

So what are your thoughts on that? Component scanning practically

So what are your thoughts on that? Component scanning practically replaced explicit wiring in Spring world. Your article doesn't mention that at all.

^ | v · Reply · Share ›



Yegor Bugayenko author → Lubos Krnac · 4 months ago

I don't like it. Mostly because it's implicit. I want dependencies to be provided explicitly to objects, via constructors. Also, the mechanism of constructing objects should be explicit, via `new` operator. Everything that replaces this mechanism is basically evil. This is what the article is about. We want construction of objects to happen explicitly and obviously. All these Spring-ish techniques were invented for lazy and clueless programmers, who don't care about objects and their composition, but care about getting the job done as soon as possible.

2 ^ | v · Reply · Share ›



Lubos Krnac → Yegor Bugayenko · 4 months ago

Pretty arrogant statements from somebody who has little experience with DI. I wish you luck in your verbose coding. You will need it.

^ | v · Reply · Share ›



Konstantin Mikheev · 5 months ago

Honestly, I see no difference providing a dependency with constructor vs injection from the OO perspective. Object should execute a task or serve some function. It must not care about where to get all requirements.

Injection is 100 times more flexible. Injection is a life saver on large systems. Without injection... oh my gosh, I had few years of Java without injection, but only after trying Dagger I started to breathe. No more pain of passing 100500 arguments. Refactoring is 100 times easier and I need it less frequently now. On some systems (like Android) it is impossible to call constructor on your object because it is called by the system. No, I will never-never-never come back.

You say that your [Agents.java](#) is beautiful? Sorry, but it is a spaghetti-code.

12 ^ | v · Reply · Share ›



dreadwolf · 7 months ago

Impressive, you say? I can only agree. To come up with something this fugly is nothing short of impressive. I've read a few more articles from this blog, and I can honestly say I appreciate my coworkers more as a result, as even the least competent one has yet to come up with something this "impressive". So... thanks for that, I guess.

15 ^ | v · Reply · Share ›



Marcin Deptuła · 7 months ago

Hello, firstly I'd like to say it's a good article. I like whole line of your articles in which you talk about, what I think are OO's deficiencies (or improper use) like for example obsessive object mutability. When I read those I think you are pushing your programming style towards functional programming, as many of those things are resolved / improved in languages like F# and Scala.

languages like F# and Scala.

As for this article, I miss an answer to 1 potential problem I know I would have in some of my projects if I would convert IoC approach to what you described here. I would like to hear what would be your take on it. Let's say I have an object that behaves as unit of work, for example Entity Framework's contexts work like that. First thing that might be obvious is that I need to create those objects dynamically. This can be solved with creating some sort of provider which could create those for me (or maybe there is other way you would do that?). But then, there's a second issue that IoC can solve, let's say I have a controller, which will be created for any request:

```
//Controller ctor
public NewsController(PInfContext ctx, Configuration cfg, NewsConverter newsConverter) {

//Helper converter ctor
public NewsConverter(PInfContext ctx) { ... }
```

Now let's assume that NewsConverter is just a helper which also depends on PInfContext and because this implements UoW pattern, if I will create 2 objects it will break (changes inside NewsController won't be recognized by those in NewsConverter and vice versa). How IoC helps here? It can create one context object and when it will resolve NewsController, it can inject the same object into it's constructor and it's dependencies constructors.

I could of course move creation of helpers inside main controller:

```
//Controller ctor
public NewsController(PInfContext ctx, Configuration cfg)
{
    this.converter = new NewsConverter(ctx);
}
```

but this is also kind of pollution isn't it? How would you implement this, having those constraints mentioned above?

^ | v · Reply · Share ›



Yegor Bugayenko author → Marcin Deptuła · 7 months ago

The way "controllers" are created in modern MVC frameworks is terrible. Not only in MVC, in all frameworks (correct me if you know any framework that works differently). You declare a class, put into a package and let your framework make an instance of it when a new request arrives (finding your class through reflection). This is wrong, very wrong. Instead, a framework should let you create a single instance of "controllers factory" (not the best name, but in this context let's call it that way) and give into a single constructor of a "HTTP facade".

Thus, it's not possible to just convert an existing Spring or Play application into DI-container-less architecture. It is technically not possible for that very reason you mentioned above. Because the framework itself is responsible for object instantiation. The framework is designed in a wrong way.

So, I don't have an answer to this question. You can't live without a DI container in, say, Spring or Play. You need it. And this is our common misfortune :)

^ | v · Reply · Share ›



Derek William Stavis → Yegor Bugayenko · a month ago

This misfortune is also valid for Android. So sad :(

^ | v · Reply · Share ›



Grzegorz Gajos · 7 months ago

Great article, very brave one :). The only missing part here is hard argument against DI containers. Example from bottom of the article is Rultor DSL, very nice one I must admit. However, the real benefit of DI is to hide object creation, acquisition and increase code readability. Using your example: if QnlfContains class have to get data from database to resolve some logic during execution, you would need to pass database reference. Things going even more complex if those nested objects are creating other objects and passing those parameters around is going crazy. For those environment aware classes you can use DI to create and locate resources you need. The real problem I believe is when DI is fully taking control of object creation. Again, in Rultor example, to Tweets class you might want to inject github reference via DI but pass custom second parameter and things go crazy again, you create factories to satisfy DI and parametrize creation. Thanks for this post, it forced me to think again about those "trivial" stuff :)

2 ^ | v · Reply · Share ›



Yegor Bugayenko author → Grzegorz Gajos · 7 months ago

Thanks for reading and sharing your thoughts, appreciate it. "if those nested objects are creating other objects" - this should never happen, ideally. Creating (using `new` operator) objects should be done only in the main entry-point method/class of the application. That class will look big, but it will be the only place where complexity will be concentrated.

^ | v · Reply · Share ›



Grzegorz Gajos → Yegor Bugayenko · 7 months ago

Ok, what about logic that is modifying multiple classes at the same time, for example transactions. This is common example why spring is so popular. In raw case, we have to worry about transaction open/submit, connection pool, exceptions (rollbacks handling). DI+AOP is able to augment service classes for you, so you can focus entirely on business logic, very clean. When db/app model is getting bigger and bigger those features are saving lots of dev time. Other example is lazy bean evaluation, useful especially in tests context (speeds up test start), when DI context create only those classes that are needed.

^ | v · Reply · Share ›



Yegor Bugayenko author → Grzegorz Gajos · 7 months ago

Yes, that's why I don't like Spring. It flattens the entire object hierarchy breaking all encapsulation barriers and letting you manipulate object internals (their encapsulated state) in any possible

manipulate object internals (their encapsulated state) in any possible way. When db/app model is getting bigger we should refactor it and make it smaller again, breaking it down into sub-systems, for example. Spring is a pure evil.. Some day I'll write a detailed post about it :)

^ | v · Reply · Share ›



Grzegorz Gajos → Yegor Bugayenko · 7 months ago

Well, It's hard for me to argue when we're thinking the same way. Spring let you patch bad design but It's even better when you add on top of it some dynamic language like groovy ;). On the other hand there is very little people that made great designs and without such flexible tools, many hit dead end. There is very little knowledge about how to do this right (apart of great designed OO). I feel that there is huge gap between app/lib design and design patterns. Anyway, looking forward to read more posts about it :)

^ | v · Reply · Share ›



Tom J Nowell · 7 months ago

Root Composition on steroids!

^ | v · Reply · Share ›



Yegor Bugayenko author → Tom J Nowell · 7 months ago

Exactly :)

^ | v · Reply · Share ›



Duilio · 7 months ago

What if you know want to use a different db implementation ? For example you are using an API repository in production but you want to use a Array Repository in testing environment and a MySQL repository in local, would you give us an example of that?

^ | v · Reply · Share ›



Yegor Bugayenko author → Duilio · 7 months ago

Very good question, thanks. I would create a "switching" decorator (pseudo-code):

```
class OptDB implements DB {
    private final Mode mode;
    private final DB[] options;
    public OptDB(final Model mde, final DB[] opts) {
        this.mode = mde;
        this.options = opts;
    }
    @Override
    public String cell(final String query) {
        return this.opts[this.mode.read()].cell(query);
    }
}
```

Then, in your application-construction method, you use it like this:

```
new App(  
  new OptDB(  
    new ModeByEnvironmentVariable("APPLICATION_ENV"),  
    new DB[] {  
      new ApiDB(), // for production, mode=0  
      new ArrayDB(), // for testing, mode=1  
      new MySQLDB() // for local, mode=2  
    }  
  )  
).run();
```

Got the idea?

^ | v · Reply · Share ›



fluminis → Yegor Bugayenko · 5 months ago

Thanks for the article, but I do not agree with this example. Why creating objects you do not need at runtime? DI is probably not perfect and obfuscate your code at some points, I do agree with that and I agree with the fact that DI make a lot of Dark Magic. But... it could improve your productivity and help you to do unit testing by inject mock objects, etc.

I do not want to totally avoid the new keyword in my code, but for some key classes (DB, transaction are good examples) DI is a very powerfull tool.

^ | v · Reply · Share ›



TitouanGalopin · 7 months ago

I don't agree with you. IMO, you are wrong when you say: "And how does that help us? Why do we need it? Why can't we use plain old new in the main class of the application?".

The main principle of DI are the services. Service and objects are not the same, you can not replace a service by an object creation. Why? Because you can change a service declaration from external code. You can't for an object creation.

DI is extremely useful for external libraries as you can replace a single class by your own (without messing with the library code). In a world where every application would be created from scartch, I agree with your way. But DI is just the easiest and best way to use external libraries.

I think your analysis is wrong because you didn't get the main point of DI: it's flexible inheritance.

^ | v · Reply · Share ›



Yegor Bugayenko author → TitouanGalopin · 7 months ago

Replacing a single class in an external library? Doesn't it sound dirty? For me, it does. A library is just a collection of classes that can instantiate objects. How these objects are instantiated and what arguments their constructors will get - completely up to my application.

Yes, DI containers are extremely useful for badly designed libraries. This is a valid argument.

2 ^ | v · Reply · Share ›



TitouanGalopin → Yegor Bugayenko · 7 months ago

Its not about the bad or well designed libraries, it's about the abstraction a library provide to its users. When you use a library, you want abstraction. With DI, you just reference a single string corresponding to the objected created by the library. With your way, you have to create the whole library object: you cannot abstract the classes names or constructors. You have to read all the library code.

After reading your other answers, I think you have a specific idea of development in small applications that interacts, and I understand DI has not a place in it. I just don't have the same opinion as you :) .

^ | v · Reply · Share ›



Yegor Bugayenko author → TitouanGalopin · 7 months ago

Yes, I'm in general in favor of small apps (<100 classes, <20K lines of code). I think they are maintainable and easy to modify. Big apps should be broken down into smaller ones. When I'm consulting a client that has a 1M LoC enterprise-size app, my first recommendation is to break it down into pieces. So, yes, I agree here. If you have a huge monster app, DI container is an inevitable practice.

^ | v · Reply · Share ›



koolak82 → Yegor Bugayenko · 2 months ago

And your idea about "small apps" is orthogonal to DI / composition whatever. Also debatable. Instead of "small app" it could be micro services / or better yet which I like that every layer as module. That scales. Like Data layer is an "app" then it works. And it's a bigger stuff than DI / composition.

I don't argue with your ideas in general. They have merit too.

But consider this too, Service Locator /JNDI, all has its pros and cons like DI. Wont you have a container there?

^ | v · Reply · Share ›



Andrew Glassman · 7 months ago


Beware any advice that starts with always or never.

Yes, DI Frameworks may be code polluters for smaller apps, but they are life savers for larger apps, and families of apps at the enterprise level. Adhering to a common DI framework across a family of applications gives developers a starting point for figuring an app out. Also, DI frameworks such as Spring come with loads of detailed documentation and built in functionality. I wouldn't write them off as evil.

6 ^ | v · Reply · Share ›



"Larger apps" is what I'm also against. An app should be maintainable and changeable. If there are over 100k lines of code in a piece of software, it is not maintainable any more (well, there could be exceptions, of course). When your apps starts to need a dependency injection container, it's time to break it down into pieces.



"A Budget doesn't know what kind of database it is working with." And not that rare, the programmer will find himself in the same way as Budget does. Not once did I have to read classes and classes and XMLs upon XMLs just to find where's that injection defined. It's like having a framework that does integer addition.



That's exactly my point! :)



According Robert C. Martin a source code should be read like a poem with words easy to understand, in the source code the words are the variables and procedures, one of the problems in object oriented programming are the constructors because have the name of classes and the developers do not know how the class will construct the object, as a solution he proposed build the objects from static method, for example "SalesReport.fromDates(start, end)", this for me is easy to understand against "new SalesReport(start, end)" when the class SalesReport contains to many constructors. This convinces me but I always have questions about the "philosophy" of software. Perhaps the solution is not to apply all rules so rigid, because constructing from static method is antipattern, DI is an antipattern, get/set is antipattern, but the static method helps to developers to understand the code in some cases, DI helps to construct very configurable systems, get/set helps to understand when a property is readable or writable.



You're right, all these anti-patterns help developers to survive in their offices. When the job has to be done and there is not enough motivation/courage/intellect to find the right way, we easily fall for something simple and common. Especially if the whole "OOP" world around us is full of that solutions. And it's only sad that Robert C. Martin is making this situation worse.



10/15



Robert C. Martin is making this situation worse ? I think you should read his books and posts... He said many years before you what you say in this article : <https://sites.google.com/site/...> He has always been a Dependency Injection advocate (and a clean code advocate generally speaking) and not a big fan of third party framework (although admitting they can be useful if used with precaution : <http://blog.8thlight.com/uncle...>

I think his approach is more realistic and less categorical than yours...

^ | v · Reply · Share ›



Lorenzo → Yegor Bugayenko · 7 months ago

I don't think the lack of intellect is the reason of the use of "right way" of OOP, the right way for some people is the wrong way for others and both sides can be very smart. Anyway I bought the books you cite in your posts...

1 ^ | v · Reply · Share ›



Crak Lod Mes · 8 months ago

your "The Right Way" code looks ugly

11 ^ | v · Reply · Share ›



Yegor Bugayenko author → Crak Lod Mes · 8 months ago

For me it looks beautiful. It's a matter of what you're used to. Try to spend more lines on object composition (like in my example) and less lines for method calls. You will see how beautiful your code will become and how ugly your current code will look to you.

^ | v · Reply · Share ›



Stefano Fago · 8 months ago

The Post is very interesting and the problem exposed is a reality!
(for short: a DI container alter the way you design your software).

Some other reflection are:

1] DI Container are also a business (open source business model) too...so (in some way) we need them.

2] If you're not the software factory but a controller/coordinator of different consultants, you need to use DI Container because you need fast (pre-assembled) code and controls.

3] with your approach, that i like, you preserve injection, construction order but there are problem with dynamic

configurations or if you prefer with metadata.

Making a generic approach, we'll arrive to a micro framework made by:

StaticHolder, DynamicHolder, HoldersRegistry

We need these elements to make measures, statistics, tracking and dynamic configuration.

So, for example, i can have:

```
new CSV(  
new StaticHolder<report>(new Report(  
new DynamicHolder<dbconnector>( new MySQL() )  
  
)  
).print();  
);
```

Make it sense?

^ | v · Reply · Share ›



Yegor Bugayenko · author → Stefano Fago · 8 months ago

Well, a "holder" or a "registry" is something we invented in OOP because we wanted to have an ability to manage data the way we did it before, in C or Assembly. We want to fetch the data and then manipulate with them. Classic code and data separation, which exists in all x86 processors (and only there). To the contrary, in object-oriented programming, we should imagine our objects as humans. We should anthropomorphize them. Thus, instead of naming it a StaticHolder think about what this object actually is. And give it a real name. Maybe Jeff or Frank :) But not StaticHolder.

1 ^ | v · Reply · Share ›



Tommy Long · 8 months ago

This pattern of presenting some well known / documented facts to onboard the audience before coming out with total rubbish is getting old. If you consider a 90 line single composition statement good or "impressive" you need to dig into composition patterns a bit more. I don't come from the world of java, but if the examples you have provided are the only or best way to use certain dependency injection containers, I suggest you shop around some more. Generic and Type based composition should allow specifying abstraction to concrete bindings, and the container will "buildup" the composition itself.

If your concretes require unabstracted dependencies, you lose the benefits of loose coupling and test mocking, let alone your composition issues.

12 ^ | v · Reply · Share ›



Michel Daviot · 8 months ago

Hi ! I tend to agree with the general idea in the article, even if the last exemple is a bit extreme (don't say this is readable, easy to maintain code !).

What is your opinion on the macwire framework ? <http://di-in-scala.github.io/#...> This is what I use in an opensource project, it help constructor dependency injection but does not prevent you from using it in a manual way.

^ | v · Reply · Share ›



Yegor Bugayenko · author → Michel Daviot · 8 months ago

Well, I consider the last example a very readable and maintainable. Maybe it's a matter of habit, but for me it looks clean. It's a pure object composition, purely

declarative style, 100% object oriented, if you wish :)

I took a look at macwire, and I don't like it. Mostly because it looks procedural, instead of object oriented. This is procedural code:

```
DB db = new MySQL("...");
Report report = new Report(db);
CSV csv = new CSV(report);
System.out.println(csv.print());
```

This style is inherited from Assembly, COBOL, C, etc. This is object composition (the same code, but different style):

```
System.out.println(
    new CSV(
        new Report(
            new MySQL("...")
        )
    ).print()
);
```

See the difference? We're NOT giving instructions to the computer. We're composing an object and asking it to do something for us.

1 ^ | v · Reply · Share ›



Willem Salembier → Yegor Bugayenko · 8 months ago

Do you realize that you've written the exact same code twice? Once using local variables, and once without.

2 ^ | v · Reply · Share ›



Yegor Bugayenko author → Willem Salembier · 8 months ago

Yes, I do :) The first example allows me to do intermediary manipulations with objects I create, while the second one allows me to communicate only with a single object, responsible for the task at hands. See the difference?

^ | v · Reply · Share ›



Willem Salembier → Yegor Bugayenko · 8 months ago

So how would you share the db object between two report objects? I don't get your point. Why hide object references? The point is an object should expose public methods and hides its internal implementation. This principle is not violated by storing it in a local variable. Plus, the scope of local variables is already limited to the surrounding method.

3 ^ | v · Reply · Share ›



Yegor Bugayenko author → Willem Salembier · 8 months ago

I think that in general (there can be exceptions) we should try to generate object constructing from object manipulating. In an ideal

separate object constructing from object manipulating. In an ideal OOP world all new operators should be called in one file only. We compose a huge big object, encapsulating everything into it, and then give it full control. Something like:

```
final App app = new App(
    new Records(
        new MySQL(),
        //... just like my example above
    )
);
app.run();
```

Now the question is why local variable use is discouraged. Because they are used only/mostly as a temporary cache for the objects we create. And this caching only pollutes the code. This is how I would do:

```
final class Main {
    public void main(String... args) {
        new App(
            new Projects(this.db()),
            new Employees(this.db())
        ).run;
    }
    @Cacheable
    private Db db() {
        return new MySQL(/*...*/);
    }
}
```

Pay attention to the use of @Cacheable annotation (it is AOP, with jcabi-aspects). The code is clean and caching is done behind the scene, but AOP engine.

1 ^ | v · Reply · Share ›



Ivano Pagano → Yegor Bugayenko · 8 months ago

I may want to reuse one of the component objects to create different instances of other dependents.

```
DB mydb = new MyDB(...);

doMeAReport(
    new CSV(
        new Report(mydb);
    );
);
```



```
doMeAnotherReport(  
    new Xml(  
        new Report(mydb);
```

More from [yegor256.com](#)

256

[yegor256.com](#) recently published

Software Quality Award - Yegor Bugayenko

10 Comments  Recommend 

256

[yegor256.com](#) recently published

How to Protect a Business Idea While Outsourcing

8 Comments  Recommend 

256

[yegor256.com](#) recently published

My Favorite Software Books - Yegor Bugayenko

13 Comments  Recommend 