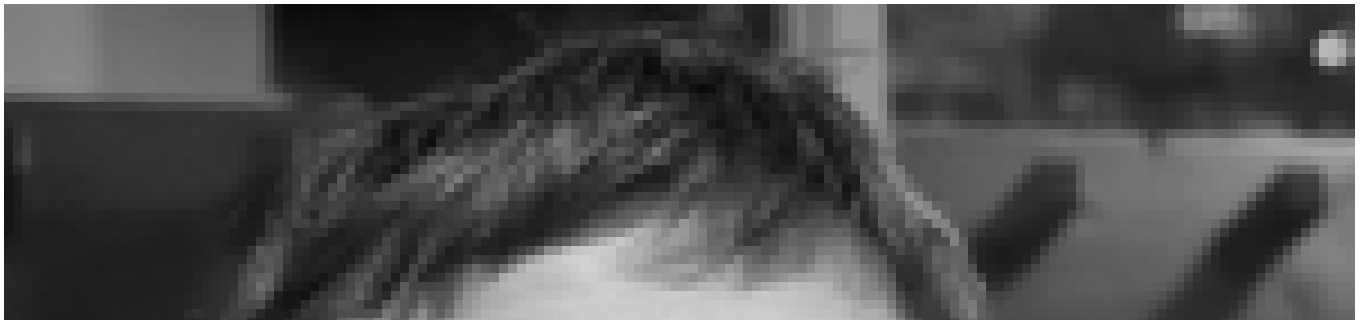
Community

yegor256.com

**49 Comments** • Created 7 months ago



# How Immutability Helps

In a few recent posts, including "Getters/Setters. Evil. Period.", "Objects Should Be Immutable", and "Dependency Injection Containers are Code Polluters", I universally labelled all mutable objects with "setters" (object methods starting…

(yegor256.com)

## 49 Comments



**Lambda Pool** • 4 months ago

Probably the best article here, State is the root of all Evil says the inventor of Erlang, Joe Armstrong.
I believe on it. Concurrency problems usually are due incorrect use of mutable objects for passing messages! Some people are just crazy about mutability they think its right when its completely wrong!!!!!



**Invisible Arrow** • 5 months ago

This question is more around organizing interfaces and their implementations.
The 'Envelope' interface in jcabi-email has 3 implementations defined within the interface itself. (I've sometimes used this pattern when I had just one default implementation which would be used most of the time)
But most of the other interfaces have implementations in separate files under another package.

Is there any specific criteria which is used for preferring one kind of organization over the other?

• Reply • Share ›

**Lambda Pool** ➚ Invisible Arrow • 4 months ago

no its not, not even close. The second example is thread safe due constructor creation of object, the second one are pure trash. for example, if I have a queue for processing those emails and send the first object probably it will be corrupted by another thread sometime, the second one never will be corrupted.

• Reply • Share ›

**Yegor Bugayenko** author ➚ Invisible Arrow • 5 months ago

I still can't find a rule to follow :) In general, if the class is small, put it into the interface. Or maybe, if the class doesn't need any external dependencies (other classes), put it into the interface. I'm not yet sure.

1 • Reply • Share ›

**Hans-Peter Störr** • 5 months ago

The design of jcabi-email discussed here seems indeed nicely done! I still like to mention that you could easily do the just the same with mutable objects, too, if you care about proper splitting of functionality. I very much trust you would do better than commons-email also without the limitations of Java constructors. ;-)

(Please notice that I'm not arguing against immutability here - I agree that this is a very important design strategy that should be used much more. And indeed, the discussed incentive to have less attributes is an interesting benefit of immutable objects in Java. Never thought about that.)

• Reply • Share ›

**Yegor Bugayenko** author ➚ Hans-Peter Störr • 5 months ago

Immutability restricted me, that's the point. With mutable classes I can easily make a wrong design with multiple setters, etc. Immutability restricts me by making my constructors too big as soon as I go the wrong direction. Immutability tells me to stop and refactor much sooner. That's the idea. And looks like you got it :)

1 • Reply • Share ›

**Marcos Douglas Santos** ➚ Yegor Bugayenko • 5 months ago

I fully agree. Restriction is the key about immutability.
I rewrote my AWS S3 lib using immutable objects -- a small project -- only to test this approach and this restriction helped me design better project.

1 • Reply • Share ›

**Invisible Arrow** • 6 months ago

This is an excellent article. I now realize better how this will lead to smaller components interacting with other and will not grow into monolithic pieces over time.
I do have a couple of questions mainly on designing complex objects.

1. Are factories and builders anti-patterns? Because the way I see it after reading this article, you don't really need a builder because the constructor essentially builds the components and "higher" level objects simply use these smaller constructed objects in their respective constructors.

Something like

```
new Bed(new Pillow(Color.WHITE), new Mattress(Color.BLUE, MattessType.FOAMY))
```

In case, I get a fancy bed (which has a book holder, say), I just have a new class FancyBed extending Bed.
However I do have a question regarding it's construction. What would be a better construction mechanism for the FancyBed?

```
new FancyBed(new Pillow(Color.WHITE), new Mattress(Color.BLUE, MattessType.FOAMY), new B
```

or

```
new FancyBed(theNormalBed, new BookHolder())
```

2. Also what about changing objects that represent, say, users in a system. Ex. when a user's email changes, do we always construct a new user object copying from the old one and changing just the email? Also after reading the another article of yours, I realized this kind of user is just a "data holder".
How do we model this kind of an object?

Would love to hear your thoughts on this.

• Reply • Share ›

**Yegor Bugayenko** author ➚ Invisible Arrow • 6 months ago

1. Definitely the second option. Fancy bed should look like a normal bed, but act differently. It is a decorator pattern, see https://en.wikipedia.org/wiki/.... You may also like this article about this very subject: http://www.yegor256.com/2014/1...

2. We should clearly separate object state from object behavior. If email is a uniquely identifying characteristic of a user - it is his state. In this case, every time you change an email, you get a new user object. If email is a behavior of a user and it's identity is defined by ID (for example), then we don't change the object. I'm planning to write an article about it soon.

• Reply • Share ›

**Invisible Arrow** ➚ Yegor Bugayenko • 6 months ago

Thanks! That helps a lot. Will wait for the article for the 2nd point :)

• Reply • Share ›

**Yegor Bugayenko** author ➚ Invisible Arrow • 6 months ago

Here is the answer to the second question:
http://www.yegor256.com/2014/1...

• Reply • Share ›

**Martin** • 7 months ago

I read your articles about getters, setters and immutability. I disagree with several of your points.

I'll start with the getter/setter argument. You don't like getters because no one "gets" the weight from a dog; they ask the dog to inform its weight. There's absolutely no conceptual difference between asking the Dog object to inform its weight through a method called "getWeight()" and through a method called "weight()". None. You might infer that the designer naming the method was not thinking in an object-oriented fashion, but that inference does not follow from the method name. "getWeight()", "weight()", "kindlyInformMeYourWeight()" are all exactly the same, because the concept behind them is the same. So, the use of the verb "get" is not wrong per-se; it's a matter of taste, and you made it clear you dislike it, but that's a long way from proving there's something conceptually wrong with it.

A different matter is the case of setters. Your analogy here is that no one changes the weight of a dog; the dog is fed, and its behaviour is to change its weight accordingly. And since the analogy does not allow the weight of the dog to be changed from "outside the dog", then that must be universally true, right? Again, you're wrong here. The analogy should only be used to illustrate the conceptual point of encapsulation, and it should end there. That's the goal of using an analogy: to illustrate a point. It's not the goal of an analogy to use them as the observable object from which we infer universal laws.

If we projected the dog analogy to other aspects of OOD (as you project it towards the immutability postulate), you could say something like this:

No one "creates" a dog. The dow "grows". So it's wrong to have class constructors, because it couples the creator to the knowledge of what it takes to produce a dog. Objects should be grown naturally from phenotipic expressions of genes through protein-induced chemical reactions.

The dog analogy stops, for you, at the weight example. But grabbing the analogy and using it to perform thought-experiments involving other situations, it's perfectly natural to "set" the name of a dog. In fact, I've never met a dog which named itself. By no means a new dog is created after you name your puppy "Pluto", nor a magical "snapshot" of the unnamed dog materializes before you. If the analogy makes the concept true (like you seem to claim), then it should be possible for a person to set the name of his/her pet. So why not have a setter?

Unfortunatelly you conclude that since setting the weight is wrong, then setting anything is wrong, thus all objects should be immutable. Incidentally, any mutable object is "impure".

There's no real world analogy to "create a new dog when you want to change its name", but you seem OK with implementing your code that way. The "snapshot" explanation is nothing more than a forced justification, again, with no equivalent either in practical or conceptual terms. It seems you try to invent analogies that'd support your assertion that all objects should be immutable.

Let me explain this again. You assert that all objects should be immutable, always, and your only conceptual justification is a set of cherry-picked analogies. The analogies you chose are tailored to fit your personal prefference to use immutable objects, and you disregard equivalent analogies saying that they don't apply to your made-up concepts (in concrete, the talk about "snapshots").

It's a case or circular logic: you cannot use setters because objects should be immutable because you cannot use setters.

The usage of "never" is usually a warning that something is fishy. It's the phylosophic equivalent to a code-smell.

You offer other explanations: problems that are solved or don't exist when you use immutable object. But that only justifies the use of immutable object when you want to solve or avoid those problems. It's not justification enough to claim that objects should always be immutable and that if you include a setter in a method you're not thinking in pure-OOD.

I agree with those commenters who offered examples of setter-like methods, when those examples refer to valid scenarios. I find no conceptual problem with "setting" the front-right tire of my car. In fact, that's exactly what I do when I get a flat tire. I don't create a new car. I don't generate a snapshot of the car. I change the tire and that's that.

Where do I, personally, draw the line between the analogy and the concept? In the exact point in which the analogy fails to illustrate the concept. There seems to be little point to be made for the assertion "all objects should be immutable". On the other hand, there's a lot to be said about "immutable objects solve or avoid these problems".

7 ^ | v • Reply • Share ›

**Yegor Bugayenko** author → Martin • 7 months ago

First of all, thanks for your comment! You're right, I can't prove my statements with an exact math. I can't give you a formula that will show that an immutable object is always better. Mostly because you can always show me millions lines of code written with mutable objects and they work, for decades. The point of my article(s) and my analogies is to show that there is a better way to thinking about software and developing it. I'm going to publish a new article next week, about "what is a good object and what are its merits". In there I'll try to explain again, what I believe is a properly designed class/object. It will be just a personal opinion of myself, and there won't be any exact proofs.

Programming is art, I believe. In art there can't be exact rules, but one picture you find beautiful, another one is merely an average. What is the difference? Can we make an explicit rule about it and prove it? So, don't expect me to prove my points of view with math. I can't. All I can say is that the code from Apache commons-email is ugly and my code is beautiful :)

2 ^ | v • Reply • Share ›

**Lambda Pool** → Yegor Bugayenko • 4 months ago

You should figure out why those code "works"... it works on a very mediocre and no safe way rely usually on thread pools, locks and application servers

which make things done underground.
That's why those old legacy code looks like working but in fact, it is stressing hardware for running!

• Reply • Share ›

**Thomas Eyde** • 7 months ago

I rephrase my question: I got the immutable part and how it solves concurrency, but how do you handle persistence from multiple threads? I can't see how to make the database immutable...

• Reply • Share ›

**Yegor Bugayenko** author → Thomas Eyde • 7 months ago

Look at these classes: https://github.com/aintshy/hub... They are all immutable and all represent database entities

• Reply • Share ›

**Dan Howard** → Yegor Bugayenko • 7 months ago

I don't know you think these classes are good example of anything. They are a mix of a data object, iterable and JDBC junk. This is not how to write single responsibility classes. You can write a bean without setters and use an ORM which supports Object initialization without them (like http://persism.sourceforge.net....

You could also use a "fluid bean" style looking like this:

```
public final class Postman {

    private  String host;
    private  int port;
    private  String user;
    private  String password;

    public String host() {
        return host;
    }

    public int port() {
        return port;
    }

    public String user() {
        return user;
    }

    public String password() {
        return password;
    }
```

```
    public Postman host(String host) {
        this.host = host;
        return this;
    }

    public Postman port(int port) {
        this.port = port;
        return this;
    }

    public Postman user(String user) {
        this.user = user;
        return this;
    }

    public Postman password(String password) {
        this.password = password;
        return this;
    }

    @Override
    public String toString() {
        return "Postman{" +
                "host='" + host + '\'' +
                ", port=" + port +
                ", user='" + user + '\'' +
                ", password='" + password + '\'' +
                '}';
    }
}

// ---
        Postman postman = new Postman().
                host("blah").
                port(80).
                user("x").
                password("123");
```

1 ▲ | ▼ • Reply • Share ›

**Lambda Pool** ➔ Dan Howard • 4 months ago

ORM sux

▲ | ▼ • Reply • Share ›

**Yegor Bugayenko** author ➔ Dan Howard • 7 months ago

Of course you can, but this article explains why you shouldn't :)

▲ | ▼ • Reply • Share ›

**Thomas Eyde** • 7 months ago

What about scenarios where objects aren't immutable by nature, like something which includes people? People can change name, move to a different address, change their phone number and so on. Would an immutable design like this fit equally well?

1 ^ | v • Reply • Share ›

**Yegor Bugayenko** author ➚ Thomas Eyde • 7 months ago

Look at these articles, they are about this very subject: http://www.yegor256.com/2014/0... and http://www.yegor256.com/2014/0.... In short, people/objects are **immutable** while their behavior is **not constant**. A good analogy is a file. Its content is not constant, while its identity is the same - it is one and the same file, no matter what's inside. When you rename the file, you create a new immutable object.

^ | v • Reply • Share ›

**Thomas Eyde** ➚ Yegor Bugayenko • 7 months ago

I stumbled upon these articles before I read your reply. Although I read good arguments, I still fail to see the implementation. Maybe some context is in order: I have a few domain classes, one of them is the Employee. An Employee can, among other things, ChangeName(), ChangeAddress and ChangeUserName(). These changes are persisted to an EventStore. The Employee's internal state is built from all previous events.

If I should make the Employee immutable, whenever I do something with it, I would have to copy the state, pass it on to a new instance, and then apply the changes to this new instance. I could always copy the events and introduce a copy-constructor for that. But then what? Now it seems like I end up with ApplyX-methods for every X(), like: ChangeName(), ApplyChangeName() and so on.

Not very elegant. Even if I should find a better way, the EventStore isn't immutable. So the concurrency problem fixed with the immutable Employee still remain in the EventStore.

I'm not arguing against immutable objects here, just trying to understand...

^ | v • Reply • Share ›

**Grzegorz Gajos** ➚ Thomas Eyde • 7 months ago

If previous state will be immutable you can create new immutable employee with updated fields and store reference to previous, also immutable employee object. You don't have to copy over everything, you have to maintain only immutability. I think that your question is similar to thread safe array list solution https://docs.oracle.com/javase... In jcabi-email you can find very elegant solution to applyX for every X(). Look how builder is made https://github.com/jcabi/jcabi.... X represented as object can simplify your implementation and make it very elegant.

• Reply • Share ›

**Kanstantsin Kamkou** • 7 months ago

I have one question here. Yegor, you've created the `Postman` interface. And the same file contains `Default` and `NoDrafts` classes. Then you've marked them as **final**. Is it something more usual for Java, or why can't we create standalone classes which I can extend if needed? In your case the interface has functionality which is 100% unclaimed if I were decide to create my own implementation like `MyPostmanBlackhole`. Is it worth to have them there?

• Reply • Share ›

**Yegor Bugayenko** author ➚ Kanstantsin Kamkou • 7 months ago

I believe that each class (in any OOP language) should be either abstract or final (see this answer too: http://stackoverflow.com/quest.... When/if you want to extend a final class, use decorators.

• Reply • Share ›

**Kanstantsin Kamkou** ➚ Yegor Bugayenko • 7 months ago

Thank you for the link, I've read your answer. My question is more about why did you decided to go with `new Postman.Default` instead of `new PostmanDefault`?

• Reply • Share ›

**Yegor Bugayenko** author ➚ Kanstantsin Kamkou • 7 months ago

Ah, it's a question of style, I guess. I like to keep classes inside interfaces when they are closely related and will never exist without each other. This `Postman.Default` is not just a default implementation of postman, but also the most preferred one. It's sort of a message to the users of postman - "use this one, it's the best for you". When class is outside of interface it means there will be more and they are interchangeable. Something like that. Again, it's a matter of taste.

• Reply • Share ›

**Grzegorz Gajos** ➚ Yegor Bugayenko • 7 months ago

Isn't also good idea to replace "StSubject" with "Stamp.Subject", or if file become too big, just follow standard Java convention "SubjectStamp"? Isn't actually "StSubject" a naming violation?

• Reply • Share ›

**Yegor Bugayenko** author ➚ Grzegorz Gajos • 7 months ago

It is a valid remark, you're right. The best name would be just `Subject`, but this will lead to clashes with many other classes in Java world. That's why I added a prefix, but tried to save the idea that it is just a stamp that is a subject. `SubjectStamp` is wrong, since it duplicates the information already provided by the `implements` keyword. If tomorrow I'll rename `Stamp` to something else, I will have to rename all child classes implementing it? That would be a bad design.

**Grzegorz Gajos** ➚ Yegor Bugayenko • 7 months ago

So what what do you think about putting every /St.*/ classes inside Stamp interface? File will grow, that's true. But is it bad? From usage perspective it will give syntactic sugar. IDE autcomplete popup also like this kind of structure.

 • Reply • Share ›

**Yegor Bugayenko** author ➚ Grzegorz Gajos • 7 months ago

Not a bad idea, but it will make the class very big. There are 7 stamps now in the library, but should be many more. I implemented just the most important ones. But in general I agree, what you propose would look nicer.

 • Reply • Share ›

**Grzegorz Gajos** ➚ Yegor Bugayenko • 7 months ago

Just wondering if there's argument against big file? We shouldn't create big classes because it's hard to maintain them but is there any argument against big file with multiple small classes? For example is there a downside of Stamp interface that will contain let say 50 classes?

 • Reply • Share ›

**Yegor Bugayenko** author ➚ Grzegorz Gajos • 7 months ago

In an active development by many programmers a long file will cause many more merge conflicts, especially if people will move parts of the file. This is the main problem. Besides that, of course, readability is lower if a file is long. It's difficult to find what is what inside it.

 • Reply • Share ›

**Grzegorz Gajos** ➚ Yegor Bugayenko • 7 months ago

True, good point, big files are bad. What about double link. Little overhead but gives syntactic sugar and group similar classes.

```
interface Stamp {
  class Subject extends StSubject {};
  class Recipient extends StRecipient {};
}

class StSubject implements Stamp { /* implementation */ }
class StRecipient implements Stamp { /* implementation */ }
```

It gives also additional flexibility, you can change internal library names without compromising end client code. Plus, you can switch implementation of Subject by just changing superclass of Subject.

// edit:

Just realized that all c-tors have to be copied over. On the other hand

factory methods are accessible, maybe it's better then to avoid public c-tors in favor of factory methods?



**Yegor Bugayenko** author ↗ Grzegorz Gajos • 7 months ago

It looks interesting, indeed! However, there are a few downsides. First, all my classes are `final` - won't be possible to keep them final in this design. Second, all static analyzers will complain about that method-less classes. So, I would not do it. But, again, it is very innovative :)



**Grzegorz Gajos** ↗ Yegor Bugayenko • 7 months ago

Thank you. My last idea was to create Stamps (smth like Collections) factory class but it would force additional delegation code. Too much hassle I think.

```
class Stamps {
  static StSubject subject(String subject) { return new StSubject(
  static StRecipient recipient(String recipient) { return new StRe
}
// usage with static imports
new Stamp[] {
  subject("Another subject"),
  recipient("recipient@of.email.com")
};
```

In the end, looks like your prefix solution is the best possible approach :).



**Yegor Bugayenko** author ↗ Grzegorz Gajos • 7 months ago

Keep in mind that static methods is an evil by itself. Try to avoid them at all cost. See http://www.yegor256.com/2014/0...



**Grzegorz Gajos** ↗ Yegor Bugayenko • 7 months ago

Well I have to disagree. I'm using them very often as named constructors to make code more readable. For example this code:

```
new Token("string", "string")
```

is more cryptic than this:

```
Token.fromUserAndSalt("string", "string")
```

**Kanstantsin Kamkou** ➚ Grzegorz Gajos • 7 months ago

I've asked the same question ^^. IMHO `Token token = new Token(new User());`. In Java it looks more obvious.

• Reply • Share ›

**Guest** ➚ Yegor Bugayenko • 7 months ago

Ok, I got it. Thank you!

• Reply • Share ›

**Grzegorz Gajos** • 7 months ago

What happen if I pass multiple topics by mistake. "new StSubject("1"), new StSubject("2")". Should it throw IllegalArgumentException? Last topic win? Isn't better to create builder class with methods like "withTopic(String topic)", which will in the end ".build()" produce immutable object?

• Reply • Share ›

**Yegor Bugayenko** author ➚ Grzegorz Gajos • 7 months ago

Current implementation of `StSubject` just replaces the subject, without any validation. If you want to add a validation, just create your own instance of Message (decorator) and catch any duplicate attempts to set subject into it (unfortunately Message in JDK is very bloated and such a decorator will look ugly, but that's not my fault :)

The builder already exists in the `Envelope.MIME`, check this code: https://github.com/jcabi/jcabi...

• Reply • Share ›

**Grzegorz Gajos** ➚ Yegor Bugayenko • 7 months ago

Didn't notice Envelope.MIME.with(), even all iterables are transferred to new arrays in c-tors. I didn't realize that builder can be also immutable. Great design!

• Reply • Share ›

**Kanstantsin Kamkou** ➚ Grzegorz Gajos • 7 months ago

I think it is up to you to decide which functionality shall be responsible for the linkage and the builder pattern is one of the possible options. As I understand the whole concept neither the `Postman` or `Envelope` is responsible for validation of meta-fields.

• Reply • Share ›

**Grzegorz Gajos** ➚ Kanstantsin Kamkou • 7 months ago

Actually I found code responsible for that. Envelop.unwrap() method is performing validation. Especially nice is "Strict" implementation.

• Reply • Share ›

**Kanstantsin Kamkou** ➚ Grzegorz Gajos • 7 months ago

`Envelop.unwrap` validates fields according RFC standard. What you're looking for is a custom validation rule. In case of *multiple topics by*

*mistake* I'm not sure that the check is needed.

• Reply • Share ›

## More from yegor256.com

**yegor256.com** recently published

### How to Implement an Iterating Adapter

13 Comments Recommend

**yegor256.com** recently published

### Software Quality Award - Yegor Bugayenko

10 Comments Recommend

**yegor256.com** recently published

### Constructors Must Be Code-Free

68 Comments Recommend