http://www.yegor256.com/2014/11/07/how-immutability-helps.html

How Immutability Helps

7 November 2014 modified on 16 November 2014 Yegor Bugayenko

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 with `set`) evil. My argumentation was based mostly on metaphors and abstract examples. Apparently, this wasn't convincing enough for many of you — I received a few requests asking to provide more specific and practical examples.

Thus, in order to illustrate my strongly negative attitude to "mutability via setters", I took an existing commons-email Java library from Apache and re-designed it my way, without setters and with "object thinking" in mind. I released my library as part of the jcabi family — jcabi-email. Let's see what benefits we get from a "pure" object-oriented and immutable approach, without getters.

Here is how your code will look, if you send an email using commons-email:

```
Email email = new SimpleEmail();
email.setHostName("smtp.googlemail.com");
email.setSmtpPort(465);
email.setAuthenticator(new DefaultAuthenticator("user", "pwd"));
email.setFrom("yegor@teamed.io", "Yegor Bugayenko");
email.addTo("dude@jcabi.com");
email.setSubject("how are you?");
email.setMsg("Dude, how are you?");
email.send();
```

Here is how you do the same with jcabi-email:

```
Postman postman = new Postman.Default(
  new SMTP("smtp.googlemail.com", 465, "user", "pwd")
);
Envelope envelope = new Envelope.MIME(
  new Array<Stamp>(
    new StSender("Yegor Bugayenko <yegor@teamed.io>"),
    new StRecipient("dude@jcabi.com"),
    new StSubject("how are you?")
  ),
  new Array<Enclosure>(
    new EnPlain("Dude, how are you?")
  )
);
postman.send(envelope);
```

I think the difference is obvious.

In the first example, you're dealing with a monster class that can do everything for you, including sending your MIME message via SMTP, creating the message, configuring its parameters, adding MIME parts to it, etc. The `Email` class from commons-email is really a huge class — 33 private properties, over a hundred methods, about two thousands lines of code. First, you configure the class through a bunch of setters and then you ask it to `send()` an email for you.

In the second example, we have seven objects instantiated via seven `new` calls. `Postman` is responsible for packaging a MIME message; `SMTP` is responsible for sending it via SMTP; stamps (`StSender` , `StRecipient` , and `StSubject`) are responsible for configuring the MIME message before delivery; enclosure `EnPlain` is responsible for creating a MIME part for the message we're going to send. We construct these seven objects, encapsulating one into another, and then we ask the postman to `send()` the envelope for us.

What's Wrong With a Mutable Email?

From a user perspective, there is almost nothing wrong. `Email` is a powerful class with multiple controls — just hit the right one and the job gets done. However, from a developer perspective `Email` class is a nightmare. Mostly because the class is very big and difficult to maintain.

Because the class is so big, every time you want to extend it by introducing a new method, you're facing the fact that you're making the class even worse — longer, less cohesive, less readable, less maintainable, etc. You have a feeling that you're digging into something dirty and that there is no hope to make it cleaner, ever. I'm sure, you're familiar with this feeling — most legacy applications look that way. They have huge multi-line "classes" (in reality, COBOL programs written in Java) that were inherited from a few generations of programmers before you. When you start, you're full of energy, but after a few minutes of scrolling such a "class" you say — "screw it, it's almost Saturday".

Because the class is so big, there is no data hiding or encapsulation any more — 33 variables are accessible by over 100 methods. What is hidden? This `Email.java` file in reality is a big, procedural 2000-line script, called a "class" by mistake. Nothing is hidden, once you cross the border of the class by calling one of its methods. After that, you have full access to all the data you may need. Why is this bad? Well, why do we need encapsulation in the first place? In order to protect one programmer from another, aka defensive programming. While I'm busy changing the subject of the MIME message, I want to be sure that I'm not interfered with by some other method's activity, that is changing a sender and touching my subject by mistake. Encapsulation helps us narrow down the scope of the problem, while this `Email` class is doing exactly the opposite.

Because the class is so big, its unit testing is even more complicated than the class itself. Why? Because of multiple inter-dependencies between its

methods and properties. In order to test `setCharset()` you have to prepare the entire object by calling a few other methods, then you have to call `send()` to make sure the message being sent actually uses the encoding you specified. Thus, in order to test a one-line method `setCharset()` you run the entire integration testing scenario of sending a full MIME message through SMTP. Obviously, if something gets changed in one of the methods, almost every test method will be affected. In other words, tests are very fragile, unreliable and over-complicated.

I can go on and on with this "*because the class is so big*", but I think it is obvious that a small, cohesive class is always better than a big one. It is obvious to me, to you, and to any object-oriented programmer. But why is it not so obvious to the developers of Apache Commons Email? I don't think they are stupid or un-educated. What is it then?

How and Why Did It Happen?

This is how it always happens. You start to design a class as something cohesive, solid, and small. Your intentions are very positive. Very soon you realize that there is something else that this class has to do. Then, something else. Then, even more.

The best way to make your class more and more powerful is by adding setters that inject configuration parameters into the class so that it can process them inside, isn't it?

This is the root cause of the problem! The root cause is our ability to **insert** data into mutable objects via configuration methods, also known as "setters". When an object is mutable and allows us to add setters whenever we want, we will do it without limits.

Let me put it this way — **mutable classes tend to grow in size and lose cohesiveness**.

If commons-email authors made this `Email` class immutable in the beginning, they wouldn't have been able to add so many methods into it and encapsulate so many properties. They wouldn't be able to turn it into a monster. Why? Because an immutable object only accepts a state through a constructor. Can you imagine a 33-argument constructor? Of course, not.

When you make your class immutable in the first place, you are forced to keep it cohesive, small, solid and robust. Because you can't encapsulate too much and you can't modify what's encapsulated. Just two or three arguments of a constructor and you're done.

How Did I Design An Immutable Email?

When I was designing jcabi-email I started with a small and simple class: `Postman`. Well, it is an interface, since I never make interface-less classes. So, `Postman` is... a post man. He is delivering messages to other people. First, I created a default version of it (I omit the ctor, for the sake of brevity):

```
import javax.mail.Message;
@Immutable
class Postman.Default implements Postman {
  private final String host;
  private final int port;
  private final String user;
  private final String password;
  @Override
  void send(Message msg) {
    // create SMTP session
    // create transport
    // transport.connect(this.host, this.port, etc.)
    // transport.send(msg)
    // transport.close();
  }
}
```

Good start, it works. What now? Well, the `Message` is difficult to construct. It is a complex class from JDK that requires some manipulations before it can become a nice HTML email. So I created an envelope, which will build this complex object for me (pay attention, both `Postman` and `Envelope` are immutable and annotated with @Immutable from jcabi-aspects):

```
@Immutable
interface Envelope {
  Message unwrap();
}
```

I also refactor the `Postman` to accept an envelope, not a message:

```
@Immutable
interface Postman {
  void send(Envelope env);
}
```

So far, so good. Now let's try to create a simple implementation of `Envelope`:

```
@Immutable
class MIME implements Envelope {
  @Override
  public Message unwrap() {
    return new MimeMessage(
      Session.getDefaultInstance(new Properties())
    );
  }
}
```

It works, but it does nothing useful yet. It only creates an absolutely empty MIME message and returns it. How about adding a subject to it and both `To:` and `From:` addresses (pay attention, `MIME` class is also immutable):

```
@Immutable
class Envelope.MIME implements Envelope {
  private final String subject;
  private final String from;
  private final Array<String> to;
  public MIME(String subj, String sender, Iterable<String> rcpts) {
    this.subject = subj;
    this.from = sender;
    this.to = new Array<String>(rcpts);
  }
  @Override
  public Message unwrap() {
    Message msg = new MimeMessage(
      Session.getDefaultInstance(new Properties())
    );
    msg.setSubject(this.subject);
    msg.setFrom(new InternetAddress(this.from));
    for (String email : this.to) {
      msg.setRecipient(
        Message.RecipientType.TO,
        new InternetAddress(email)
      );
    }
    return msg;
  }
}
```

Looks correct and it works. But it is still too primitive. How about `CC:` and `BCC:`? What about email text? How about PDF enclosures? What if I want to specify the encoding of the message? What about `Reply-To`?

Can I add all these parameters to the constructor? Remember, the class is immutable and I can't introduce the `setReplyTo()` method. I have to pass the `replyTo` argument into its constructor. It's impossible, because the constructor will have too many arguments, and nobody will be able to use it.

So, what do I do?

Well, I started to think: how can we break the concept of an "envelope" into

smaller concepts — and this what I invented. Like a real-life envelope, my `MIME` object will have stamps. Stamps will be responsible for configuring an object `Message` (again, `Stamp` is immutable, as well as all its implementors):

```
@Immutable
interface Stamp {
  void attach(Message message);
}
```

Now, I can simplify my `MIME` class to the following:

```
@Immutable
class Envelope.MIME implements Envelope {
  private final Array<Stamp> stamps;
  public MIME(Iterable<Stamp> stmps) {
    this.stamps = new Array<Stamp>(stmps);
  }
  @Override
  public Message unwrap() {
    Message msg = new MimeMessage(
      Session.getDefaultInstance(new Properties())
    );
    for (Stamp stamp : this.stamps) {
      stamp.attach(msg);
    }
    return msg;
  }
}
```

Now, I will create stamps for the subject, for `To:`, for `From:`, for `CC:`, for `BCC:`, etc. As many stamps as I like. The class `MIME` will stay the same — small, cohesive, readable, solid, etc.

What is important here is why I made the decision to refactor while the class was relatively small. Indeed, I started to worry about these stamp classes when my `MIME` class was just 25 lines in size.

That is exactly the point of this article — **immutability forces you to design small and cohesive objects**.

Without immutability, I would have gone the same direction as commons-email. My `MIME` class would grow in size and sooner or later would become as big as `Email` from commons-email. The only thing that stopped me was the necessity to refactor it, because I wasn't able to pass all arguments through a constructor.

Without immutability, I wouldn't have had that motivator and I would have done what Apache developers did with commons-email — bloat the class and turn it into an unmaintainable monster.

That's jcabi-email. I hope this example was illustrative enough and that you will start writing cleaner code with immutable objects.