http://www.yegor256.com/2014/11/20/seven-virtues-of-good-object.html

Seven Virtues of a Good Object

20 November 2014 modified on 11 March 2015 Yegor Bugayenko

Martin Fowler says:

> *A library is essentially a set of functions that you can call, these days usually organized into classes.*

Functions organized into classes? With all due respect, this is wrong. And it is a very common misconception of a class in object-oriented programming. Classes are not organizers of functions. And objects are not data structures.

So what is a "proper" object? Which one is not a proper one? What is the difference? Even though it is a very polemic subject, it is very important. Unless we understand what an object is, how can we write object-oriented software? Well, thanks to Java, Ruby, and others, we can. But how good will it be? Unfortunately, this is not an exact science, and there are many opinions. Here is my list of qualities of a good object.

Class vs. Object

Before we start talking about objects, let's define what a *class* is. It is a place where objects are being born (a.k.a. *instantiated*). The main responsibility of a class is to *construct* new objects on demand and *destruct* them when they are not used anymore. A class knows how its children should look and how they should behave. In other words, it knows what *contracts* they should obey.

Sometimes I hear classes being called "object templates" (for example, Wikipedia says so). This definition is not correct because it places classes into a passive position. This definition assumes that someone will get a template and build an object by using it. This may be true, technically speaking, but conceptually it's wrong. Nobody else should be involved — there are only a class and its children. An object asks a class to create another object, and the class constructs it; that's it. Ruby expresses this concept much better than Java or C++:

```
photo = File.new('/tmp/photo.png')
```

The object `photo` is constructed by the class `File` (`new` is an entry point to the class). Once constructed, the object is acting on its own. It shouldn't know who constructed it and how many more brothers and sisters it has in the class. Yes, I mean that reflection is a terrible idea, but I'll write more about it in one of the next posts :) Now, let's talk about objects and their best and worst sides.

1. He Exists in Real Life

First of all, an object is a **living organism**. Moreover, an object should be anthropomorphized, i.e. treated like a human being (or a pet, if you like them more). By this I basically mean that an object is not a *data structure* or a collection of functions. Instead, it is an independent entity with its own life cycle, its own behavior, and its own habits.

An employee, a department, an HTTP request, a table in MySQL, a line in a file, or a file itself are proper objects — because they exist in real life, even when our software is turned off. To be more precise, an object is a *representative* of a real-life creature. It is a *proxy* of that real-life creature in front of all other objects. Without such a creature, there is — obviously — no object.

```
photo = File.new('/tmp/photo.png')
puts photo.width()
```

In this example, I'm asking `File` to construct a new object `photo`, which will be a representative of a real file on disk. You may say that a file is also something virtual and exists only when the computer is turned on. I would agree and refine the definition of "real life" as follows: It is everything that exists aside from the scope of the program the object lives in. The disk file is outside the scope of our program; that's why it is perfectly correct to create its representative inside the program.

A controller, a parser, a filter, a validator, a service locator, a singleton, or a factory are **not** good objects (yes, most GoF patterns are anti-patterns!). They don't exist apart from your software, in real life. They are invented just to tie other objects together. They are artificial and fake creatures. They don't represent anyone. Seriously, an XML parser — who does it represent? Nobody.

Some of them may become good if they change their names; others can never excuse their existence. For example, that XML parser can be renamed to "parseable XML" and start to represent an XML document that exists outside of our scope.

Always ask yourself, "What is the real-life entity behind my object?" If you can't find an answer, start thinking about refactoring.

2. He Works by Contracts

A good object always works by contracts. He expects to be hired not because of his personal merits but because he obeys the contracts. On the other hand, when we hire an object, we shouldn't discriminate and expect some specific object from a specific class to do the work for us. We should expect *any* object to do what our contract says. As long as the object does what we need, we should not be interested in his class of origin, his sex, or

his religion.

For example, I need to show a photo on the screen. I want that photo to be read from a file in PNG format. I'm contracting an object from class `DataFile` and asking him to give me the binary content of that image.

But wait, do I care where exactly the content will come from — the file on disk, or an HTTP request, or maybe a document in Dropbox? Actually, I don't. All I care about is that some object gives me a byte array with PNG content. So my contract would look like this:

```
interface Binary {
  byte[] read();
}
```

Now, any object from any class (not just `DataFile`) can work for me. All he has to do, in order to be eligible, is to obey the contract — by implementing the interface `Binary`.

The rule here is simple: every public method in a good object should implement his counterpart from an interface. If your object has public methods that are not inherited from any interface, he is badly designed.

There are two practical reasons for this. First, an object working without a contract is impossible to mock in a unit test. Second, a contractless object is impossible to extend via decoration.

3. He Is Unique

A good object should always encapsulate something in order to be unique. If there is nothing to encapsulate, an object may have identical clones, which I believe is bad. Here is an example of a bad object, which may have clones:

```
class HTTPStatus implements Status {
  private URL page = new URL("http://www.google.com");
  @Override
  public int read() throws IOException {
    return HttpURLConnection.class.cast(
      this.page.openConnection()
    ).getResponseCode();
  }
}
```

I can create a few instances of class `HTTPStatus`, and all of them will be equal to each other:

```
first = new HTTPStatus();
second = new HTTPStatus();
assert first.equals(second);
```

Obviously utility classes, which have only static methods, can't instantiate good objects. More generally, utility classes don't have any of the merits mentioned in this article and can't even be called "classes". They are simply terrible abusers of an object paradigm and exist in modern object-oriented languages only because their inventors enabled static methods.

4. He Is Immutable

A good object should never change his encapsulated state. Remember, an object is a representative of a real-life entity, and this entity should stay the same through the entire life of the object. In other words, an object should never betray those whom he represents. He should never change owners. :)

Be aware that immutability doesn't mean that all methods always return the same values. Instead, a good immutable object is very dynamic. However, he never changes his internal state. For example:

```
@Immutable
```

```
final class HTTPStatus implements Status {
  private URL page;
  public HTTPStatus(URL url) {
    this.page = url;
  }
  @Override
  public int read() throws IOException {
    return HttpURLConnection.class.cast(
      this.page.openConnection()
    ).getResponseCode();
  }
}
```

Even though the method `read()` may return different values, the object is immutable. He points to a certain web page and will never point anywhere else. He will never change his encapsulated state, and he will never betray the URL he represents.

Why is immutability a virtue? This article explains in detail: Objects Should Be Immutable. In a nutshell, immutable objects are better because:

- Immutable objects are simpler to construct, test, and use.
- Truly immutable objects are always thread-safe.
- They help avoid temporal coupling.
- Their usage is side-effect free (no defensive copies).
- They always have failure atomicity.
- They are much easier to cache.
- They prevent NULL references.

Of course, a good object doesn't have setters, which may change his state and force him to betray the URL. In other words, introducing a `setURL()` method would be a terrible mistake in class `HTTPStatus`.

Besides all that, immutable objects will force you to make more cohesive, solid, and understandable designs, as this article explains: How Immutability Helps.

5. His Class Doesn't Have Anything Static

A static method implements a behavior of a class, not an object. Let's say we have class `File`, and his children have method `size()`:

```
final class File implements Measurable {
  @Override
  public int size() {
    // calculate the size of the file and return
  }
}
```

So far, so good; the method `size()` is there because of the contract `Measurable`, and every object of class `File` will be able to measure his size. A terrible mistake would be to design this class with a static method instead (this design is also known as a utility class and is very popular in Java, Ruby, and almost every OOP language):

```
// TERRIBLE DESIGN, DON'T USE!
class File {
  public static int size(String file) {
    // calculate the size of the file and return
  }
}
```

This design runs completely against the object-oriented paradigm. Why? Because static methods turn object-oriented programming into "class-oriented" programming. This method, `size()`, exposes the behavior of the class, not of his objects. What's wrong with this, you may ask? Why can't we have both objects and classes as first-class citizens in our code? Why can't both of them have methods and properties?

The problem is that with class-oriented programming, decomposition doesn't work anymore. We can't break down a complex problem into parts, because only a single instance of a class exists in the entire program. The

power of OOP is that it allows us to use objects as an instrument for scope decomposition. When I instantiate an object inside a method, he is dedicated to my specific task. He is perfectly isolated from all other objects around the method. This object is a *local variable* in the scope of the method. A class, with his static methods, is always a *global variable* no matter where I use him. Because of that, I can't isolate my interaction with this variable from others.

Besides being conceptually against object-oriented principles, public static methods have a few practical drawbacks:

First, it's **impossible to mock** them (Well, you can use PowerMock, but this will then be the most terrible decision you could make in a Java project ... I made it once, a few years ago).

Second, they are **not thread-safe** by definition, because they always work with static variables, which are accessible from all threads. You can make them thread-safe, but this will always require explicit synchronization.

Every time you see a public static method, start rewriting immediately. I don't even want to mention how terrible static (or global) variables are. I think it is just obvious.

6. His Name Is Not a Job Title

The name of an object should tell us what this object **is**, not what it **does**, just like we name objects in real life: book instead of page aggregator, cup instead of water holder, T-shirt instead of body dresser. There are exceptions, of course, like printer or computer, but they were invented just recently and by those who didn't read this article. :)

For example, these names tell us who their owners are: an apple, a file, a series of HTTP requests, a socket, an XML document, a list of users, a regular expression, an integer, a PostgreSQL table, or Jeffrey Lebowski. A

properly named object is always possible to draw as a small picture. Even a regular expression can be drawn.

In the opposite, here is an example of names that tell us what their owners do: a file reader, a text parser, a URL validator, an XML printer, a service locator, a singleton, a script runner, or a Java programmer. Can you draw any of them? No, you can't. These names are not suitable for good objects. They are terrible names that lead to terrible design.

In general, avoid names that end with "-er" — most of them are bad.

"What is the alternative of a `FileReader`?" I hear you asking. What would be a better name? Let's see. We already have `File`, which is a representative of a real-world file on disk. This representative is not powerful enough for us, because he doesn't know how to read the content of the file. We want to create a more powerful one that will have that ability. What would we call him? Remember, the name should say what he is, not what he does. What is he? He is a file that has data; not just a file, like `File`, but a more sophisticated one, with data. So how about `FileWithData` or simply `DataFile`?

The same logic should be applicable to all other names. Always think about **what it is** rather than what it does. Give your objects real, meaningful names instead of job titles.

More about this in Don't Create Objects That End With -ER.

7. His Class Is Either Final or Abstract

A good object comes from either a final or abstract class. A `final` class is one that can't be extended via inheritance. An `abstract` class is one that can't have children. Simply put, a class should either say, "You can never break me; I'm a black box for you" or "I'm broken already; fix me first and then use".

There is nothing in between. A final class is a black box that you can't modify by any means. He works as he works, and you either use him or throw him away. You can't create another class that will inherit his properties. This is not allowed because of that `final` modifier. The only way to extend such a final class is through decoration of his children. Let's say I have the class `HTTPStatus` (see above), and I don't like him. Well, I like him, but he's not powerful enough for me. I want him to throw an exception if HTTP status is over 400. I want his method, `read()`, to do more that it does now. A traditional way would be to extend the class and overwrite his method:

```
class OnlyValidStatus extends HTTPStatus {
  public OnlyValidStatus(URL url) {
    super(url);
  }
  @Override
  public int read() throws IOException {
    int code = super.read();
    if (code > 400) {
      throw new RuntimException("unsuccessful HTTP code");
    }
    return code;
  }
}
```

Why is this wrong? It is very wrong because we risk breaking the logic of the entire parent class by overriding one of his methods. Remember, once we override the method `read()` in the child class, all methods from the parent class start to use his new version. We're literally injecting a new "piece of implementation" right into the class. Philosophically speaking, this is an offense.

On the other hand, to extend a final class, you have to treat him like a black box and decorate him with your own implementation (a.k.a. Decorator Pattern):

```
final class OnlyValidStatus implements Status {
  private final Status origin;
  public OnlyValidStatus(Status status) {
    this.origin = status;
  }
  @Override
  public int read() throws IOException {
    int code = this.origin.read();
    if (code > 400) {
      throw new RuntimException("unsuccessful HTTP code");
    }
    return code;
  }
}
```

Make sure that this class is implementing the same interface as the original one: `Status`. The instance of `HTTPStatus` will be passed into him through the constructor and encapsulated. Then every call will be intercepted and implemented in a different way, if necessary. In this design, we treat the original object as a black box and never touch his internal logic.

If you don't use that `final` keyword, anyone (including yourself) will be able to extend the class and ... offend him :(So a class without `final` is a bad design.

An abstract class is the exact oposite case — he tells us that he is incomplete and we can't use him "as is". We have to inject our custom implementation logic into him, but only into the places he allows us to touch. These places are explicitly marked as `abstract` methods. For example, our `HTTPStatus` may look like this:

```
abstract class ValidatedHTTPStatus implements Status {
  @Override
  public final int read() throws IOException {
    int code = this.origin.read();
    if (!this.isValid()) {
      throw new RuntimException("unsuccessful HTTP code");
    }
```

```
    return code;
  }
  protected abstract boolean isValid();
}
```

As you see, the class doesn't know how exactly to validate the HTTP code, and he expects us to inject that logic through inheritance and through overloading the method `isValid()`. We're not going to offend him with this inheritance, since he defended all other methods with `final` (pay attention to the modifiers of his methods). Thus, the class is ready for our offense and is perfectly guarded against it.

To summarize, your class should either be `final` or `abstract` — nothing in between.