“Step one in the transformation of a successful procedural developer into a successful object developer is a lobotomy”

–David West[1]

[1]David West, *Object Thinking*, Microsoft Press, 2004.

Contents

Preface

There have been many books written about Object-Oriented Programming (OOP). Why another? Because we are in trouble. We are getting further and further away from what OOP creators had in mind, and there is almost no hope of turning back. All existing OOP languages encourage us to treat objects as “data structures with attached procedures”, which is a totally wrong and dangerous misconception. We create new languages, but they do the same or even worse. As object-oriented programmers, we are forced to think like procedural programmers thought 40 years ago. In other words, think like computers, not objects.

This book is a collection of practical recommendations that I believe can change the situation and stop the degradation of OOP. I learned most recommendations from the publications listed in the bibliography at the end of the book. Some of them I just made up.

There are 23 strands of advice, grouped into four chapters: birth, school, employment, and retirement. We’ll talk about “Mr. Object,” an anthropomorphized entity in the object-oriented world. He will be born, go to school, get hired to do some job for us, and then retire. We’ll watch how it unfolds and try to learn something new. Together. Let’s go.

Wait. You know, before publishing, I sent this book to a dozen reviewers and almost all of them complained about the absence of an intro. They said that I threw them blindly into the first theme, with no overall context. They also said that it was difficult to digest my ideas, having a lot of previous experience in Java/C++ programming. They found that what they think OOP is contradicts with my understanding of it. Long story short, they all demanded that I write an intro. So, here it is.

I believe that OOP was designed as a solution for the problems we had in *procedural* programming, especially in languages like C and COBOL. The procedural style of writing code is very easy to understand for those who understand how a CPU works, processing instructions one by one, and letting them manipulate the data in memory. A piece of code in C, also known as a "function," is a set of statements, which have to be executed in a chronological order, basically moving data from one place in memory to another and making some in-fly transformations with them. That's how it worked for years and it still works. There is a ton of software written in that style, including all major Unix operating systems, for example.

Technically, this approach works—the code compiles and runs. However, there are problems with *maintainability.* The author of the code can more or less easily understand how it works while he or she is writing it. However, when you look at that code later, it's rather difficult to figure out what the intention of its author was. In other words, it is written for computers, not for humans. The best example of such a procedural and imperative language is probably Assembly. It stays as close as possible to the CPU, is very far from the language we speak in real life. There is no such thing as customer, file, rectangle, or price in Assembly. There are only registers, bytes, bits, and pointers—the things a CPU understands very well.

That's how it was years ago, when computers were big, slow, and in charge. We had to learn and speak their language, not the other way around. This was mostly because we had to make our software fast in order to be useful. We were fighting for each processing instruction, for each byte of memory. We didn't really care much about maintainability, more about speed and memory usage. It is important to mention that twenty years ago, programmers were much cheaper than computers. Excuse me that comparison, but it's true. Hiring a new programmer was cheaper than buying a bigger hard disc. Sometimes it was not even possible to solve a performance problem with extra hardware. There was no faster or larger hardware! But programmers were rather cheap (you can try to find statistics about our salaries twenty years ago). That's why we had to do what CPUs told us to.

Fortunately, the situation started to change some time ago and the problem of maintainability became more important than speed and memory. The lifecycle of software products also started to grow and it became obvious that Assembly code simply can't survive a transfer from one team to another—the new team would always want to re-write from scratch instead of figuring out how that 5000-line listing works. I believe that's how higher-level programming paradigms started to appear, including functional, logical and object-oriented (there are more, but these three are the most popular, I think). They all were shifting the focus from computers to humans. They allowed us to speak our own language instead of the language our CPU was used to. They helped us make the code much easier to understand and therefore much more maintainable. That was the intention.

However, historically, OOP inherited a lot from procedural programming. Well, not OOP as a paradigm, but the languages that became popular and were declared object-oriented.

I'm talking mostly about C++ and Java. Others, like Ruby, simply followed the direction set by these big two. Maybe that was the reason why C++ became popular—because it looked very similar to C and was consequently easy to learn. Java was also designed to simplify the transition from C++—its syntax looked very much like C++ and was easy for C++ programmers to learn. Because of these two big compromised transitions (from C to C++ and from C++ to Java) we have OOP that looks very much like procedural C.

Even though we have classes and objects, we still have instructions, statements, and their chronological execution. We don't deal any more with pointers, memory manipulations, and CPU registers, but the core principle is still in place—we instruct our CPU what to do and we manipulate data in memory. You may say, what's wrong with that? Nothing is wrong, if you want to stay in the procedural domain. Just like nothing was wrong with Assembly. Except the fact that the code was not really maintainable. This is the same problem we have today with Java/Ruby/Python/etc software—it is not maintainable, because it never was object-oriented.

Our code has classes, methods, objects, inheritance, and polymorphism, but it's not really object-oriented. What exactly is wrong with it? This is what I'm trying to explain in this book. It's really difficult to say what I have to say in just a few paragraphs. You really have to read it all, in order to grasp the idea and the mindset of pure OOP.

I tried to make the material as practical as possible and illustrate ideas with realistic code examples. Moreover, almost every section here has a blog post on the same or a very relevant subject. You can find links to these blog posts at the beginnings of sections. Feel free to post your comments there and I'll try to reply and discuss. Honestly, I don't think I'm right in everything I'm saying here. I was a procedural programmer myself for many years. It's really difficult to forget it all and start thinking in objects, instead of instructions and statements. Thus, I will appreciate your feedback.

That was the intro. I don't think it gave you a lot of information, but at least you know now what we'll be talking about in the next two hundred pages. Be ready for a lot of controversy. And be brave to challenge yourself. Have fun!

Chapter 1

Birth

An object is a living organism—let's start from this. From the first page on, we'll do as much as we can to *anthropomorphize* it. We will be treating it as a human being, in other words. That's why I will use *he* to refer to an object. And my dear female readers, please don't be offended. I may sometimes be rude to a poor object and don't want to be rude to a woman. So in this book, an object is a male, a "he."

To start with, he lives inside his *scope of visibility.* For example, (I'm mainly using Java and will continue to do so; I hope you understand it):

```
if (price < 100) {
  Cash extra = new Cash(5);
  price.add(extra);
}
```

Object `extra` is visible only inside the `if` block—that is his scope of visibility. Why is that important now? Because an object is a living organism. We should define what his living environment is and will be before we breathe life into him. What remains within him and what lies outside? In this example, `price` is outside and the number `5` is inside, right?

By the way, before we continue, let me assure you that everything you are going to read in this book is very practical and pragmatic. Rather than waxing on about philosophy, the majority revolves around the practical application of object-oriented programming to real-life problems. The main goal I'm trying to achieve with this writing is to increase the *maintainability* of your code. Of our code.

Maintainability is an important quality of any kind of software, and it may be measured as the time required for me to understand your code. The longer it takes, the lower the maintainability and therefore the worse your code is. I would even say that *if I don't understand you, it's your fault*. By understanding objects and their roles in OOP, you will increase your code's maintainability. Your code will become shorter, easier to digest, more modular, more cohesive, etc. It will become better, which in most real-life projects means cheaper; that's it.

So please don't be surprised by my seemingly too philosophical and abstract discussions. They are indeed very practical.

Now, back to the scope of visibility. If I'm `extra`, then `price` is the world around me. Number `5` is inside me, and is my inner world. Well, this is not exactly right. For now, it is enough to agree that `price` is outside and `5` is inside. We'll get back to this again, though a bit later in Section 3.4.

1.1 Never use -er names

The first job, after you know the scope of visibility for a future object, is to invent a good name for its class.

But wait, let me step aside for a few minutes and discuss something else—the difference between an object and a class. I'm sure you understand what it is. A class is a *factory* of objects. Let me convince you, it's important.

A class makes objects, though we usually phrase that by saying a class *instantiates* them:

```
class Cash {
  public Cash(int dollars) {
    //...
  }
}
Cash five = new Cash(5);
```

This is different from what we call the Factory Pattern, but only because this `new` operator in Java is not as powerful as it could be. The only thing you can use it for is to make an instance—an object. If we ask class `Cash` to make a new object, we get a new object. There is no check for whether similar objects already exist and can be reused, there are no parameters that would modify the behavior of `new`, etc.

`new` is a primitive control for a factory of objects. In C++, there is also a `delete` operator that allows us to delete an object from the factory. In Java and many other "more advanced" languages, unfortunately, we don't have that. In C++, we can ask a factory to make an object for us, use it, and then ask the same factory to destroy it:

```
class Cash {
public:
  explicit Cash(int dollars);
};
Cash* five = new Cash(5); // making an object
std::cout << *five;
delete five; // destroying it
```

In Ruby, this idea of “a class as a factory” is most properly expressed in the following way:

```
class Cash
  def initialize(dollars)
    # ...
  end
end
Cash five = Cash.new(5)
```

`new` is a static method of class `Cash`, and when it’s called, the class obtains control and makes object `five` for us. This object encapsulates number `5` and behaves like an integer.

Thus, a well-known Factory Pattern is a more powerful alternative to operator `new`, but conceptually they are the same. A class is a factory of objects. A class makes objects, keeps track of them, destroys them when necessary, etc. Most of these features, in most languages, are implemented by the runtime engine, not the code in the class, but it doesn’t really matter. What we see on the surface is a class that can give us objects when we ask for them. You may wonder, what about utility classes, which don't have any objects. We will talk more about them later, in Section 3.2.3.

The Factory Pattern, in Java, works like an extension to the `new` operator. It makes it more flexible and powerful, by adding an extra logic in front of it. For example:

```
class Shapes {
  public Shape make(String name) {
    if (name.equals("circle")) {
      return new Circle();
    }
    if (name.equals("rectangle")) {
      return new Rectangle();
    }
    throw new IllegalArgumentException("not found");
  }
}
```

This is a typical factory in Java that helps us instantiate objects, using textual names of their types. In the end, we still use the `new` operator. My point is that conceptually, there is not much difference between Factory Pattern and `new` operator. In a perfect OOP language this functionality would be available in the `new` operator. I want you to think of a class as a warehouse of objects, which we can get from there when needed and return when not needed any more.

Sometimes a class is explained as a template of an object. This is absolutely wrong, because this definition makes a class a passive, brainless listing of code that is simply being copied somewhere when the time comes. Even though it may technically look like this for you, try not to *think* like this. A class is a *factory* of objects, period. By the way, I'm not promoting the Factory Pattern here. I'm actually not a big fan of it, although it is technically a valid concept. I'm saying that we should think about a class as an active manager of objects. We may also call it a storage unit or warehouse—a place where we get our objects and where we return them back.

Actually, having in mind that an object is a living creature, his class is his mother. That would be the most accurate metaphor.

Now, back to the main subject of this section—the problem of how to choose a good name for the class. There are basically two approaches: the right way and the wrong way. The wrong one would be to look at what our class objects are *doing* and give their class a name based on functionality. For example, here is a class named with this thinking in mind:

```
class CashFormatter {
  private int dollars;
  CashFormatter(int dlr) {
    this.dollars = dlr;
  }
  public String format() {
    return String.format("$ %d", this.dollars);
  }
}
```

When I have an object of class `CashFormatter`, what does it do for me? It formats dollar amounts to text. We should call it a formatter, right? Isn't that obvious?...

You probably noticed that I didn't call this `CashFormatter` a "he." This is because it is not an object I would respect. It is not something I can anthropomorphize and treat as a respectful citizen in my code.

This naming principle is very wrong and very popular. I encourage you not to follow this way of thinking. The name of a class should not originate from the functionality that its objects expose! Instead, a class should be named by what he *is*, not what he *does*. This `CashFormatter` must be renamed to `Cash` or `USDCash` or `CashInUSD`, etc. Method `format()` should be renamed to `usd()`. For example:

```java
class Cash {
  private int dollars;
  Cash(int dlr) {
    this.dollars = dlr;
  }
  public String usd() {
    return String.format("$ %d", this.dollars);
  }
}
```

In other words, objects must be characterized by their capabilities. What I am is manifested by what I can do, not by my attributes, like my height, weight, or color.

The evil ingredient here is the "-er" suffix.

There are many examples of classes named like this, and all of them have that "-er" suffix, including Manager, Controller, Helper, Handler, Writer, Reader, Converter, Validator ("-or" is also evil), Router, Dispatcher, Observer, Listener, Sorter, Encoder, and Decoder. All these names are wrong. I'm sure you've seen many of them before. Here are a few counter examples: `Target`, `EncodedText`, `DecodedData`, `Content`, `SortedLines`, `ValidPage`, `Source`, etc.

The rule has exceptions though. Some English nouns have -er suffix, which originally was placed there in order to indicate that these nouns were performers of activities, but that was long time ago. For example, computer or user. We don't think anymore about a user as of something that is literally "using" something. It's more like a person who our system interacts with. We don't understand computer as something that "computes", instead it's an electronic device that is, well, a computer. But there are not so many exceptions like that.

An object is not a connector between his outside world and his inner world. An object is not a collection of procedures we can call in order to manipulate the data encapsulated inside him. Absolutely not! Instead, an object is a representative of his encapsulated data. See the difference?

A *connector* is not respected, because it just passes information through without actually being strong or smart enough to modify it or do something on its own. To the contrary, a *representative* is a self-sufficient entity who is capable of making his own decisions and acting on his own. Objects must be representatives, not connectors.

A class name that ends in "-er" tells us that this creature is not really an object but a collection of procedures that can manipulate some data. It is a procedural way of thinking that is inherited by many object-oriented developers from C, COBOL, Basic, and other languages. We are using Java and Ruby now, but we are still thinking in terms of data and procedures.

So, how do we name a class properly?

Look at what its objects will encapsulate and come up with a name for this group—simple as that. Let's say you have a list of numbers and an algorithm that tells you which number is prime. If you want to see a list of only prime numbers, don't call this class `Primer` or `PrimeFinder` or `PrimeChooser` or `PrimeHelper`. Instead, name him `PrimeNumbers` (this is Ruby, just for diversity):

```
class PrimeNumbers
  def initialize(origin)
    @origin = origin
  end
  def each
    @origin
      .select { i prime? i }
      .each { i yield i }
  end
  def prime?(x)
    # ...
  end
end
```

See what I mean? The class `PrimeNumbers` behaves like a list of numbers but returns only prime ones. Here is how we would design similar functionality in C, using a purely procedural style:

```
void find_prime_numbers(int* origin,
  int* primes, int size) {
  for (int i = 0; i < size; ++i) {
    primes[i] = (int) is_prime(origin[i]);
  }
}
```

Here, we have a procedure called `find_prime_numbers` that accepts two arrays of integers, goes through the first array to find all prime numbers, and marks their positions in the second array. There is no object involved. This is a purely *procedural* approach, and it is wrong. Well, it is still right in procedural languages, but we are in the OOP world.

This procedure is a connector between two pieces of data: an original list of numbers and a list of prime numbers. An object is something different. An object is not a connection; he is a representative of other objects and their combinations. In the example above, we create an object of class `PrimeNumbers` that behaves like a collection of numbers, but only prime numbers will be seen in the collection.

When your object is, in reality, the procedure `find_prime_numbers`, you are in trouble. An object is not supposed to work as a collection of procedures, even though technically he may look very similar. While the `PrimeNumbers` class encapsulates a list of numbers, it doesn't allow me to manage that list or find something in it. Instead, he says, "I am the list now!" When I need to do something with the list, I ask my object, and he decides what to do with my request. If he wants, he will get some data from the original list. If not, it's up to him.

`PrimeNumbers` *is* a list of numbers, not a bunch of methods that can help me manipulate the list. He is a *list*!

Let's summarize this section: When it's time to give a name to a new class, think what he *is*, not what he *does*. He is a list, and he can pick an element by number. He is an SQL record, and he can fetch a single cell as an integer. He is a pixel, and he can change his color. He is a file, and he can read his contents from the disc.

He is an encoding algorithm, and he can encode. He is an HTML document, and he can be rendered.

What I do and who I am are two different things.

Also, yet another bad class name is the one that ends with `Util` or `Utils`. They are so called “utility classes” and will be discussed in Section 3.2.3.