“Object-oriented programming offers a sustainable way to write spaghetti code”[1]

—Paul Graham
The Hundred-Year Language

[1]It was said here: `http://www.paulgraham.com/hundred.html`. Do I agree? Yes, I do. The object-oriented programming we’re using right now is indeed a perfect tool to create unreadable and unmaintainable code. Does it mean it is dead and can’t be fixed? I don’t think so.

Contents

Acknowledgements **9**

Preface **11**

5 Gray **13**

5.1 Compound names 14
5.2 Validators and assertions 24
5.3 Printers instead of getters 36
5.3.1 Single Responsibility Principle 41
5.3.2 God object 43
5.3.3 Code duplication 44
5.3.4 Utility objects 46
5.3.5 Polymorphic media 48
5.4 Private static literals 54
5.5 Configurable objects 63
5.6 Temporal coupling 70
5.7 Inheritance 83
5.7.1 Subtyping 83
5.7.2 Implementation inheritance 86
5.8 Gradients of immutability 89
5.9 Law of Demeter 101
5.10 Algorithms 106

6 Dark **125**

6.1 Annotations . . . 126

6.1.1 Marshalling . . . 128

6.1.2 Dependency injection . . . 133

6.1.3 Access control . . . 136

6.1.4 AOP and aspects . . . 138

6.2 MVC . . . 147

6.3 Dependency injection containers . . . 158

6.4 Reflection . . . 166

6.4.1 Type introspection . . . 167

6.4.2 Type casting . . . 169

6.4.3 Annotations . . . 173

6.4.4 Instantiating by class name . . . 173

6.4.5 Method invocation by name . . . 175

6.4.6 Setting and getting attributes . . . 176

6.4.7 Serialization . . . 176

6.4.8 Byte code manipulations . . . 178

6.5 ORM . . . 179

6.5.1 Horizontal DoR . . . 180

6.5.2 Vertical DoR . . . 184

6.5.3 Mapping engine . . . 185

6.5.4 DTO . . . 190

6.5.5 SQL speaking objects . . . 191

6.5.6 Too many round trips . . . 195

6.5.7 Duplicated SQL manipulations . . . 198

6.5.8 “JOIN” and foreign keys . . . 200

6.5.9 “INSERT” . . . 203

6.5.10 “UPDATE” . . . 204

6.5.11 Transactions . . . 206

Epilogue **209**

Preface

Object-oriented programming (OOP) unfortunately is not an exact science. This means that there is no clear definition anywhere of what an object, a class, a method, a variable, or a program is. There is no "formula" to put all these pieces together and *prove* their consistency[1]. Functional and logical programming paradigms have that, while the object-oriented one doesn't. In OOP, in most cases, the formula still is: "if it works, don't touch it."

Despite this lack of formality in OOP, the majority of software we use now consists of classes, objects, and methods. The authors think that their software is object-oriented. Can they prove that? They can't. Can I prove the opposite and tell them that they misunderstand the object-oriented paradigm because some of their methods are static while others are getters? I can't.

I have tried, however. In the first volume of this book there were a number of principles suggested, which were supposed to make software *more* object-oriented. They were not strictly scientific and didn't have formal proofs, but they did help many programmers to increase the maintainability of their source code—to make it easier to understand and extend. I can tell that by the feedback received through the blog, at the conferences, and simply by email.

Thanks to this feedback you now hold this book in your hands. It contains another portion of those "magic" principles, which are not really scientific but may be rather helpful. To what extent

[1]Martin Abadi and Luca Cardelli, *A Theory of Objects*, Springer, 1998. This book attempts to formalize an *object calculi*, however repeating many mistakes made in object-oriented programming earlier, like classes, traits, inheritance, and others.

you will be able to apply them to your code depends on many factors, including your willingness to change and the amount of legacy code you have to work with.

The book starts with Chapter 5, to match consistently with the first volume, where you will find the first four chapters.

Chapter 5

Gray

There are sixteen sections in this book, grouped into two chapters, with almost no grouping principle. Well, there is a principle, but it is very artificial. The first chapter is about things that *hurt* object-oriented programming, while the second chapter is about things that *kill* it. That's why the first one is called "Gray," and the second one "Dark." The techniques in the Gray chapter are not fatal, but are rather bad. The ones in the second chapter are absolutely deadly, and you must stay away from them as far as possible.

5.1 Compound names

In any object-oriented software written in any programming language, there are three things that we have to give names to: classes, methods, and variables[1]. They are the building blocks of the language, and they must be named properly in order to make the code readable and maintainable.

Some principles of naming classes and methods were discussed in Sections 1.1 and 2.4 of the first volume of Elegant Objects. Variable names are also very important, and have to be chosen well. We will discuss them in this section.

I'm aware of two popular notations: snake_case, and camelCase, which are both wrong and confusing. There are more of them (like Hungarian notation, positional notation, or kebab-case), but they are less popular.

First, let's see how it all started. When the first computer program was created, I hadn't been born yet, but I know that at that time computers were rather slow. Simply typing the code for a program was a time-consuming operation. For that reason, and to save memory space, it was important to keep listings as short as possible. For example, this is a program written in Assembly language for an x86 CPU:

```
MOV AX, 16h
DEC BX
JZ  done
```

[1]Well, there are also namespaces, packages, files, directories, modules, libraries, and many others, but they are irrelevant to the problem we're discussing.

It doesn't do anything valuable, but do pay attention to the variable names it uses: `AX` and `BX`. They are very short. They have absolutely no semantic. Technically, they are not even variables, as "storage locators." They are registers in the CPU. They will have these names in any program written for the x86 CPU. A programmer must remember what `AX` means at any particular point in a program. In the first directive `MOV`, we're storing a hexadecimal value `16` to `AX`. What does `16` mean? It's not clear from this listing. For how long will it stay there? No idea. How is it related to other registers? No clue. Only the author of this code could tell us. Perhaps an author would add a few code comments attached to some lines of code to make them more readable. The code itself is not at all descriptive.

Higher level languages, like FORTRAN, BASIC, and COBOL, introduced variables with semantically richer names. For example, this is BASIC:

```
10 FOR i=1 TO 10
20 PRINT i
30 NEXT
```

Here, the variable name is `i`. It is still short, it is easy to type, it is easy to understand, but it means "index." Most variables in these early languages were named the same way, using just one letter: `x` and `y` for coordinates, `i`, `j`, and `k` for looping indexes, `p` for pointers, and simply `a`, `b`, and `c` in all other cases.

What is wrong with giving short one-letter names to variables? Actually, not much. Seriously, I don't think it's an evil notation. I find it less evil than the other two we will discuss now. Look at the BASIC example above one more time. Do you understand what the `i` variable is for? I do. Because the scope of visibility is rather small! If that piece of code had hundreds of lines and used a few dozen one-letter variables, I would have a hard time understanding and remembering their meanings. But it is rather short, and contains only three simple lines. Thanks to that, this single-letter name is a perfect choice.

However, in the 1960s, programs were rather long, imperative, and badly written. In most cases, the scope of visibility included thousands of lines of code. I can't prove this statement, I'm just guessing, but it's not really important. Look at a popular open-source Java library, and there will be many classes that contain thousands of lines. We haven't changed a lot, in this respect, since last century.

Here are some "champions" (measured on April 14, 2017):

Class	Product	LoC
`o.s.a.MethodWriter`	Spring Framework 4.7	2,393
`o.h.c.AnnotationBinder`	Hibernate 5.2	3,434
`o.h.j.JdbcResultSet`	H2 Database 1.4.194	3,849
`c.g.c.c.LocalCache`	Guava 21.0	5,176
`j.u.r.Pattern`	OpenJDK 8	5,856
`o.a.h.h.s.n.FSNamesystem`	Hadoop 3.0	7,216
`j.a.Component`	OpenJDK 8	10,157
`a.v.View`	Android SDK	23,934

Unix and C were the technologies that changed the world of programming. C inherited one-letter variables from older languages, but also introduced a "snake_case" notation[1]. Here is an example of C code from a famous book[2], page 111:

[1]The name "snake_case" was introduced much later, in 2004 in Ruby community by Gavin Kistner: `https://goo.gl/jQNA4t`.

[2]Brian Kernighan and Dennis Ritchie, *The C Programming Language*, 2nd Edition, Prentice Hall, 1988.

```
static char daytab[2][13] = {
  {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
  {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};
int day_of_year(int year, int month, int day) {
  int i, leap;
  leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
  for (i = 1; i < month; i++)
    day += daytab[leap][i];
  return day;
}
```

Pay attention to the names of the global array `daytab` and the function `day_of_year`. The name of the array is not really in snake_case, but very close. Just the underscore is omitted. The authors of the book are using both formats (with and without underscore), without any particular reason. What is important to notice here is that these names are *compound*—each of them consists of a few words: "day" + "tab," "day" + "of" + "year." That's how the authors use the code itself to explain the meaning of the variables. And it works well.

In some languages, the underscore symbol can be replaced by a hyphen. For example, in Lisp and COBOL, we could name the variable `day-tab`. This is not technically snake_case, but it's very close.

A more compact—and a better looking, in my opinion—notation is camelCase, which is the same as "PascalCase" with the first letter lowercase. It became popular with the appearance of object-oriented languages like Object Pascal and Java. It is very commonly used in Java for naming classes, methods, and variables. For example, this code is quoted from a very popular

Java book[1], page 284:

```
LocalDate today = LocalDate.now();
Month currentMonth = today.getMonth();
Month firstMonthofQuarter =
  currentMonth.firstMonthOfQuarter();
```

Here, both variables and names use camelCase notation. Their first words start with small letters, while all of the other words start with capital letters (I have no idea why "of" in the third variable doesn't start with a capital "O," probably a typo in the book). Just like snake_case, this notation helps developers explain to each other what each particular variable is for. In other words, longer variable names deliver more semantic information than shorter ones.

But, this is doing us a very bad favor.

Both snake_case and camelCase notations are great *workarounds*, but they are not really *solutions* to the problem. Moreover, as with almost every workaround, we are only making the situation worse by concealing the problem instead of fixing it. Now, finally, let's see what the problem is, and why it needs a solution.

A proper variable naming notation is supposed to solve a problem of low *readability*. We don't want to name our variables something that isn't descriptive, like `var1`, `var2`, `var3`, etc. Instead, we want to give them more meaningful names. But why? The compiler will understand the code no matter what. Why do we care about naming? Because we want to make our code more *readable*. We want to be able to *understand* it in a week, in a

[1]Benjamin Evans and David Flanagan, *Java in a Nutshell*, 6th Edition, O'Reilly Media, 2015.

month, and in five years. It will be difficult to tell the difference between `a1` and `a2`, while the difference between `encoded_array` and `decoded_array` will be much more obvious.

That is the problem. Actually, it is not exactly a problem, but a goal: we need to find a way to make our code more readable. And we will use as *descriptive* variable names as possible in order to help our readers understand the code!

Stop. Why are we trying to achieve this goal by using variable names? How about instead of that, we make the *surrounding* code more readable in the first place, and then the meaning of variables will become obvious as a result.

Just like in the first BASIC example above, where the variable `i` is just an index of a loop. Was that code readable? Definitely. Was it maintainable? Absolutely. Was our variable name descriptive enough? Yes! It was descriptive enough when it appeared in that small piece of code.

If the amount of code starts growing, we could make the variable names longer by introducing one of the variable naming notations. Or, we could take this as a signal that our code is too big. If we need to explain our code to a reader via variable names, it needs refactoring. Short and simple variable names should be enough to explain the code, otherwise the code is too long in the first place.

Using descriptive variable names is a patch, not a cure. They do make code more readable, but they don't solve the root cause of the problem—its high complexity. Because of high complexity, we need to use variable names like `utfEncodedFile` or `receivedByteArray`. We simply wouldn't be able to understand what they were for if their names were `file` and `array`. Since there are so many other files and arrays around, we need to explain to our readers what this specific file or array is for—what it is and what it does. The reader won't understand it otherwise, because the complexity of the code is too high.

But if we do things this way, we don't solve the complexity problem, we only hide it. The code becomes more readable in the short-term, but the bigger and more fundamental problem remains—it is too complex to be understood without long and descriptive variable names. By using composite variable names like `hashCode` or `extra_bytes` programmers can easily get away with badly written code, but do we really want them to get away with that? I mean, do we want to *ignore* the bigger problem and get away with short-term solutions?

Instead of hiding the complexity issue, we need to make it more visible. Mostly, we have to do this to force everyone to solve it as soon as possible. We simply need to *prohibit* long variable names. Every variable has to have a name consisting of a single noun, for example `file`, `array`, `index`, `user`, `url`, `target`, and `stream`; or even better `position`, `jeff`, `google`, `config`, and `log`. Even `i` and `j` are better than `firstArrayIndex` and `secondArrayIndex`, because they force a programmer to make code short and clear in the first place.

The code must be descriptive enough by itself. Each and every scope of visibility (a method, a class, or a script) must only have as many variables as it can have without making the names longer than single or plural nouns. When it becomes necessary to make names longer, the scope has to be broken down into smaller ones. Longer and more descriptive names will definitely help, but we must not use them. Instead, we must stop and think why a simple `text` is not enough. Why do we need to call it `saved_text` or `text_with_quotes`? The honest answer to this question would, in most cases, be "because otherwise it's unclear what `text` means."

Let's look at the examples above one more time. In that C code by Kernighan and Ritchie, why did they name that global array `daytab`? Why not just `tab` or simply `days`? Because the scope of its visibility was huge—it was actually as big as the entire application. The variable was global, and every other C function was able to see it. If they called it `days`, there would have been many semantic conflicts with other variables that could mean something else that were also called `days`. That's why they had to be specific enough, to prevent possible semantic collisions. The same is true for the function `day_of_year`—they couldn't name it `day`, because it was declared in the global scope. Many other functions and modules simply wouldn't understand what `day` meant, and might also try to overload this name with some other functions, doing something completely different.

When the scope is too big, we inevitably must make our names descriptive enough. We're doing that instead of fixing the bloated scope problem. We're simply closing our eyes to the bigger problem, pretending to make the code more readable by using `fileName` instead of `name`. That's just wrong, simply because the real problem remains with the code—it is still unreadable because it is too big. Eventually, all variables will be 30-40 characters long and the code will be very difficult to read.

Look at this code from `java.lang.String` in Oracle JDK (I removed a few comments and fixed indentation inconsistencies):

```
static int lastIndexOf(char[] source, int sourceOffset,
    int sourceCount, char[] target, int targetOffset,
    int targetCount, int fromIndex) {
```

```
  int rightIndex = sourceCount - targetCount;
  if (fromIndex < 0) {
    return -1;
  }
  if (fromIndex > rightIndex) {
    fromIndex = rightIndex;
  }
  if (targetCount == 0) {
    return fromIndex;
  }
  int strLastIndex = targetOffset + targetCount - 1;
  char strLastChar = target[strLastIndex];
  int min = sourceOffset + targetCount - 1;
  int i = min + fromIndex;
  startSearchForLastChar:
  while (true) {
    while (i >= min && source[i] != strLastChar) {
      i--;
    }
    if (i < min) {
      return -1;
    }
    int j = i - 1;
    int start = j - (targetCount - 1);
    int k = strLastIndex - 1;
    while (j > start) {
      if (source[j--] != target[k--]) {
        i--;
        continue startSearchForLastChar;
      }
    }
    return start - sourceOffset + 1;
  }
}
```

Authors of this method were trying to make it more readable by making variable names longer, but did they really manage to achieve anything? Do you understand what this method does and how? I tried to read it a few times, to no avail. Don't repeat their mistake. If the code is not readable, it has to be fixed. Longer and compound variable names are not how we fix it.

Thus, a simple rule of thumb I'm suggesting is the following—don't use any compound names anywhere in the code. If you can't explain your code using just single and plural nouns, refactor the code.

There could be exceptions where a single noun would not be enough, mostly because it would lose its meaning if the adjective is removed. For example: `timeZone`, `side-effect`, `MicroService`, `ChangingRoom`, `washing_machine`, `busStop`, `laughing-gas`, etc.

This rule is applicable to variables only. Method naming was discussed in Section 2.4 of the first volume and class naming was discussed in Section 1.1 of the first volume.