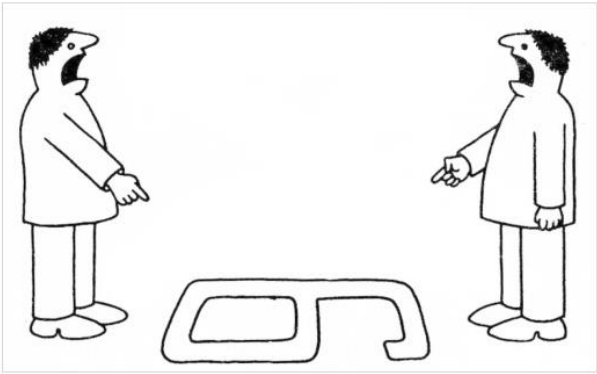
Coding Rules: the Power of Correct Names, Good and Bad Comments

How often have you had to dig into someone else's code? Instead of two hours, you may spend two days to simply understand the logic of what is happening. The funny thing is that for the person who wrote the code, everything is clear and entirely transparent. This is not surprising: after all, perfect code is a very vague concept, because each developer has the own vision of the world and of the code, too. More than once I have been in a situation when a coworker and I looked at the same code and had different opinions about its correctness and cleanness.



Sounds familiar, doesn't it? Still, there are some time-tested principles that should be adhered to. In the end, they will be advantageous for you, because if you leave your code in the state in which you yourself would like to receive it, then the world would become a little happier and cleaner.

Correct names

Correct names improve code readability, thus reducing the time required to familiarize yourself with the code, because using a method is much easier

when its name roughly describes its functionality. Everything in code consists of names (variables, methods, classes, objects, files, etc.), so this point becomes very important when creating correct, clean code. Based on the above, the name should convey meaning, for example, why the variable exists, what it does, and how it is used. I will note more than once that the best comment for a variable is to give it a good name.

from TV-series "Sherlock" (2010-2017)

Naming interfaces

Interfaces usually have names that start with a capital letter and are written in CamelCase. When writing an interface, it used to be considered good practice to add the prefix "I" to designate it as an interface (for example, IUserService), but that looks pretty ugly and distracting. In such cases, it is better to omit the prefix (UserService) and add "Impl" as a suffix to the name of its implementation (e.g. UserServiceImpl). Or possibly, as a last resort, add a "C" prefix to the name of the implementation (e.g. CUserService).

Class names

Just like interfaces, class names are capitalized and use CamelCase. It doesn't matter if we're facing a zombie apocalypse, it doesn't matter if the end is at hand — never, never, never should the name of a class be a verb! Class and object names must be nouns or compound nouns (UserController, UserDetails, UserAccount, and so on). You should not tack the abbreviation of the application onto the end of the name of each class, since that would only add unnecessary complexity. For example, if we have a User Data Migration application, then please don't add "UDM" to each class, i.e. UDMUserDetails, UDMUserAccount, UDMUserController.

Method names

Usually, method names begin with a lowercase letter, but they also use camel case style (camelCase). Above, we said that class names should never be verbs. Here the situation is just the opposite: the names of the methods should be verbs or verb phrases: findUserById, findAllUsers, createUser, and so on. When creating a method (as well as variables and classes), so use a consistent naming convention in order to avoid confusion. For example, to find a user, a method could be named getUserById or findUserById. And one more thing: don't use humor in the names of the methods, because others may not understand the joke. As a result, they may fail to grasp what the method does.

Variable names

In most cases, variable names begin with a lowercase letter and also use camelCase, except when the variable is a global constant. In such cases, all letters of the name are written in uppercase and the words are separated by an underscore ("_"). For convenience, you can use meaningful context when naming variables. In other words, when a variable exists as part of something larger, for example, firstName, lastName, or status. In such cases, you can add a prefix that indicates the object to which this variable belongs. For example: userFirstName, userLastName, userStatus. You should also avoid similar names for variables when they have completely different meanings. Here are some frequently encountered antonyms used in variable names:

- begin/end
- first/last
- locked/unlocked
- min/max
- next/previous
- old/new
- opened/closed
- visible/invisible
- source/target
- source/destination
- up/down

Short variable names

When we have variables like x or n or something like that, we don't immediately see the intent of the person who wrote the code. It's not obvious what n does. Figuring that out requires more careful contemplation (and this means time, time, time). For example, suppose we have a field that represents the id of the responsible user. Instead of some variable name like x or simply id, we will name this variable "responsibleUserId", which immediately improves readability and information content. That said, short names like n have a place as local variables in small methods, where the block of code involving this variable is just a couple of lines long, and the method name perfectly describes what happens there. Seeing such a variable, a developer understands that it is of secondary importance and has a very limited scope. As a result, the scope has a certain dependence on a variable name's length: the longer the name, the more global the variable and vice versa. As an example, here's a method to find the last saved user by date:

```
1  public User findLastUser() {
2     return findAllUsers().stream()
3             .sorted((x, y) -> -x.getCreatedDate().compareTo(y.getCreatedDate()))
4             .findFirst()
5             .orElseThrow(() -> new ResourceNotFoundException("No user exists"));
6  }
```

Here we use short-named variables x and y to sort the stream, and then we forget about them.

Optimal length

Let's continue with the topic of name length. The optimal name length is somewhere between n and maximumNumberOfUsersInTheCurrentGroup. In other words, short names suffer from a lack of meaning, while names that are too long lengthen the program without adding readability, and we're simply too lazy to write them every time. Apart from the case described above for variables with a short name like n, you should stick to a length of about 8-16 characters. This is not a strict rule, just a guideline.

Small differences

I cannot fail to mention subtle differences in names. This is also a bad practice, since these differences can be simply confusing or require spending a lot of extra time to notice them. For example, the difference between InvalidDataAccessApiUsageException and

InvalidDataAccessResourceUsageException is difficult to spot at a glance. Confusion can also often arise when using lowercase L and O, because they can be easily mistaken for 1 and 0. In some fonts the difference is more obvious, in some it is less.

The meaning

We need to make names meaningful, but not create ambiguity through synonyms, since, for example, UserData and UserInfo actually have the same meaning. In this case, we would have to dig deeper into the code to understand which particular object we need. Avoid words that do not convey helpful information. For example, in firstNameString, why do we need the word String? Could this really be a Date object? Of course not. So, we simply use firstName. I would also like to mention boolean variables. As an example, take a boolean named flagDeleted. The word flag has no meaning. It is more reasonable to call it isDeleted.

Disinformation

I would also like to say a few words about incorrect naming conventions. Let's say we have a variable named userActivityList, but instead of being a List, this object is some other container type or custom storage object. This could confuse the average programmer: it is better to call it something like userActivityGroup or userActivities.

Search

One of the drawbacks of short and simple names is that they are difficult to find in a large body of code — Which would be easier to find: "name" or "NAME_FOR_DEFAULT_USER"? The second option, of course. We should avoid frequently encountered words (letters) in names, since they will only increase the number of matching files during a search, which is not good. I would like to remind you that programmers spend more time reading code than writing it, so be smart about naming the elements of your application. But what if a good name just can't be found? What if the name of a method does not describe its functionality well? This is where comments enter the stage.

Comments

There is nothing better than a pertinent comment, but nothing clutters up a module like vacuous, outdated, or false comments. They can be a double-edged sword, no? Still, you shouldn't treat comments as unambiguously good, but rather as a lesser evil. After all, a comment is essentially a way to compensate for thinking that does not come through clearly in the code. For example, we use them to somehow convey the essence of a method, if the method itself turns out to be too confusing. In this situation, it is better to correctly refactor the code than to write descriptive notes. The older the comment, the worse the comment, because code tends to grow and evolve, but comments may remain the same. The more time that has passed since a comment was created, the more questionable it may be. Inaccurate comments are much worse than no comments at all, because they are confusing and deceptive, giving false expectations. And even if we have very tricky code, we should rewrite it rather than comment it.

Types of comments

- **Legal comments** — Comments at the beginning of each source file for legal reasons, for example:

```
1  * Copyright (c) 2007, 2013, Oracle and/or its affiliates. All rights reserved.
2  * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
```

Informative comments — Comments representing an explanation of the code (providing additional information or expounding on the intention of a given section of code).

For example:

```
/*
* Combines the user from the database with the one passed
for updating
```

```
* When a field in requestUser is empty, it is filled with
old data from foundUser
*/
private User mergeUser(User requestUser, User foundUser) {
        return new User(
        foundUser.getId(),
        requestUser.getFirstName() == null ?
requestUser.getFirstName() : foundUser.getFirstName(),
        requestUser.getMiddleName() == null ?
requestUser.getMiddleName() : foundUser.getMiddleName(),
        requestUser.getLastName() == null ?
requestUser.getLastName() : foundUser.getLastName(),
        requestUser.getAge() == null ? requestUser.getAge() :
foundUser.getAge()
        );
        }
```

- In this case, you can do without comments, since the name of the method and its parameters, coupled with very transparent functionality, describe themselves well.
- **Warning comments** — Comment intended to warn other developers about the undesirable consequences of an action (for example, warning them about why a test was marked as @Ignore):

```
1  // Takes too long to run
2  // Don't run if you don't have a lot of time
3  @Ignore
4  @Test
5  public void someIntegrationTest() {
6          ……
7          }
```

TODO — Comments that are a note about something that needs to be done in the future that but for some reason cannot be done now. This is a good practice, but such comments should be reviewed regularly in order to remove irrelevant ones and avoid clutter.

An example would be:

```
// TODO: Add a check for the current user ID (when the security context is created)

@Override
public Resource downloadFile(File file) {
	return fileManager.download(file);
	}
```

- Here we note the fact that we need to add a comparison of the user performing the download operation (whose ID we will extract from the security context) with the one who performed the save operation.
- **Reinforcing comments** — Comments emphasizing the importance of a circumstance that at first glance may seem insignificant.

  As an example, consider a piece of a method that fills a test database with some scripts:

```
Stream.of(IOUtils.resourceToString("/fill-scripts/" + x, StandardCharsets.UTF_8)
	.trim()
	.split(";"))
	.forEach(jdbcTemplate::update);
// The trim() call is very important. It removes possible spaces at the end of the script
// so that when we read and split into separate requests, we don't end up with empty ones
```

Javadoc comments — Comments that describe the API for certain functionality. There are probably the most useful comments, since the documented API is much easier to work with. That said, they can also be outdated like any other type of comment. So, never forget that the main contribution to documentation is made not by comments, but by good code.

Here's an example of a fairly common method for updating a user:

```
/**
* Updates the passed fields for a user based on its id.
	*
* @param id id of the user to be updated
* @param user user with populated fields for updating
* @return updated user
*/
	User update(Long id, User user);
```

Bad comments

- **muttering comment** — Comments that are usually written in a hurry and whose meaning is understandable only to the developer who

wrote them, since only he or she perceives the nuanced situation to which the comment refers.

Consider this example:

```
public void configureSomeSystem() {
       try{
       String configPath = filesLocation.concat("/").concat(CONFIGURATION_FILE);
       FileInputStream stream = new FileInputStream(configPath);
       } catch (FileNotFoundException e) {
       // If there is no configuration file, the default configuration is loaded
      }
}
```

- Who loads these settings? Have they already been loaded? Is this method supposed to catch exceptions and load default settings? Too many questions arise that can only be answered by delving into an investigation of other parts of the system.
- **Redundant comments** — Comments that don't carry any semantic load, since what is happening in a given section of the code is abundantly clear. In other words, the comment is no easier to read than the code.

Let's see an example:

```
public class JdbcConnection{
public class JdbcConnection{
   /**
    * The logger associated with the current class
    */
   private Logger log =
Logger.getLogger(JdbcConnection.class.getName());

   /**
    * Creates and returns a connection using the input
parameters
    */
   public static Connection buildConnection(String url,
String login, String password, String driver) throws
Exception {
       Class.forName(driver);
       connection = DriverManager.getConnection(url, login,
password);
       log.info("Created connection with db");
       return connection;
```

}

- What's the point of such comments? Everything they explain is already perfectly clear.
- **Unreliable comments** — Comments that are untrue and only misleading (disinformation). For example, here's one.

```java
/**
* Helper method. Closes the connection with the scanner if isNotUsing is true
*/
private void scanClose(Scanner scan, boolean isNotUsing) throws Exception {
   if (!isNotUsing) {
       throw new Exception("The scanner is still in use");
   } scan.close();
}
```

- What's wrong with this comment? The fact that it lies to us a little, in that the connection is closed if isNotUsing is false, not vice versa, as the comment informs us.
- **Obligatory comments** — Comments that are considered obligatory (e.g. Javadoc comments), but that in fact sometimes pile up excessively and are unreliable and unnecessary (you need to think about whether these comments are actually needed).

  Example:

```java
/**
* Create a user based on the parameters
* @param firstName first name of the created user
* @param middleName middle name of the created user
* @param lastName last name of the created user
* @param age age of the created user
* @param address address of the created user
* @return user that was created
*/
User createNewUser(String firstName, String middleName, String lastName, String age, String addre
```

  Would you be able to understand what the method does without these comments? Most likely, yes, so comments become pointless here.

- **Log comments** — Comments that are sometimes added to the beginning of a module each time it is edited (something like a change log).

```
/**
* Records kept since January 9, 2020;
*********************************************************************
* 9 Jan 2020: Providing a database connection using JDBC Connection;
* 15 Jan 2020: Adding DAO-level interfaces for working with the database;
* 23 Jan 2020: Adding integration tests for the database;
* 28 Jan 2020: Implementation of DAO-level interfaces;
* 1 Feb 2020: Development of interfaces for services,
* in accordance with the requirements specified in user stories;
* 16 Feb 2020: Implementation of service interfaces
* (implementation of business logic related to the work of the database);
* 25 Feb 2020: Adding tests for services;
* 8 Mar 2020: Celebration of International Women's Day (Terry is drunk again);
* 21 Mar 2020: Refactoring the service layer;
*/
```

- This approach was once justified, but with the advent of version control systems (for example, Git), it became an unnecessary clutter and complication of the code.
- **Authorship comments** — Comments whose purpose is to indicate the person who wrote the code, so you can contact him/her and discuss how, what, and why, e.g.:

```
* @author Bender Bending
```

- Once again, version control systems remember exactly who added any bit of code and when, so this approach is superfluous.
- **Commented-out code** — Code that was commented out for one reason or another. This is one of the worst habits, because what happens is you comment out something and forget it, and then other developers just don't have the courage to delete it (after all, what if it's something valuable?).

```
//    public void someMethod(SomeObject obj) {
//    .....
//    }
```

- As a result, commented-out code accumulates like trash. In no case should you leave such code. If you really need it, don't forget about the version control system.
- **Non-obvious comments** — Comments that describe something in an excessively complicated way.

```
/*
    * Start with an array large enough to store
    * all the data bytes (plus filter bytes) with a cushion, plus 300 bytes
    * for header data
    */
this.dataBytes = new byte[(this.size * (this.deep + 1) * 2)+300];
```

- A comment should explain the code. It should not itself need an explanation. So what's wrong here? What are "filter bytes"? What is that "+ 1" all about? Why exactly 300?

If you've already decided to write comments, here are a couple of tips:

1. Use styles that are easy to maintain: maintaining styles that are too fancy and exotic is annoying and time-consuming.
2. Do not use end-of-line comments that refer to single lines: the result is a large pile of comments. What's more, it is difficult to think up a meaningful comment for each line.
3. When you compose a comment, try to answer the question "why", not "how."
4. Avoid abridged information. As I said above, we don't need an explanation for a comment: the comment itself is the explanation.
5. You can use comments to make note of units and value ranges.
6. Place comments close to the code they describe.

Finally, I still want to remind you that the best comment is no comment, but rather the use of skillful naming throughout your application. As a rule, most of the time we will work with existing code, maintaining and extending it. It is much more convenient when this code is easy to read and understandable, since bad code is an obstacle. It's like throwing a wrench in the works, and haste is its faithful companion. And the more bad code we have, the more performance drops. This means we need to refactor from time to time. But if from the outset you try to write code that won't cause the next developers to want to find and kill you, then you won't need to refactor it as often. But it will still be necessary, since the product's conditions and requirements constantly change with the addition of new dependencies and connections.