Clean Code

A Handbook of Agile Software Craftsmanship

The Object Mentors:

Robert C. Martin

Michael C. Feathers Timothy R. Ottinger
Jeffrey J. Langr Brett L. Schuchert
James W. Grenning Kevin Dean Wampler
Object Mentor Inc.

Writing clean code is what you must do in order to call yourself a professional. There is no reasonable excuse for doing anything less than your best.

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

2

Meaningful Names

by Tim Ottinger

Introduction

Names are everywhere in software. We name our variables, our functions, our arguments, classes, and packages. We name our source files and the directories that contain them. We name our jar files and war files and ear files. We name and name and name. Because we do

so much of it, we'd better do it well. What follows are some simple rules for creating good names.

Use Intention-Revealing Names

It is easy to say that names should reveal intent. What we want to impress upon you is that we are *serious* about this. Choosing good names takes time but saves more than it takes. So take care with your names and change them when you find better ones. Everyone who reads your code (including you) will be happier if you do.

The name of a variable, function, or class, should answer all the big questions. It should tell you why it exists, what it does, and how it is used. If a name requires a comment, then the name does not reveal its intent.

```
int d; // elapsed time in days
```

The name `d` reveals nothing. It does not evoke a sense of elapsed time, nor of days. We should choose a name that specifies what is being measured and the unit of that measurement:

```
int elapsedTimeInDays;
int daysSinceCreation;
int daysSinceModification;
int fileAgeInDays;
```

Choosing names that reveal intent can make it much easier to understand and change code. What is the purpose of this code?

```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
  for (int[] x : theList)
    if (x[0] == 4)
      list1.add(x);
  return list1;
}
```

Why is it hard to tell what this code is doing? There are no complex expressions. Spacing and indentation are reasonable. There are only three variables and two constants mentioned. There aren't even any fancy classes or polymorphic methods, just a list of arrays (or so it seems).

The problem isn't the simplicity of the code but the *implicity* of the code (to coin a phrase): the degree to which the context is not explicit in the code itself. The code implicitly requires that we know the answers to questions such as:

1. What kinds of things are in `theList`?
2. What is the significance of the zeroth subscript of an item in `theList`?
3. What is the significance of the value `4`?
4. How would I use the list being returned?

The answers to these questions are not present in the code sample, *but they could have been*. Say that we're working in a mine sweeper game. We find that the board is a list of cells called `theList`. Let's rename that to `gameBoard`.

Each cell on the board is represented by a simple array. We further find that the zeroth subscript is the location of a status value and that a status value of 4 means "flagged." Just by giving these concepts names we can improve the code considerably:

```
public List<int[]> getFlaggedCells() {
  List<int[]> flaggedCells = new ArrayList<int[]>();
  for (int[] cell : gameBoard)
    if (cell[STATUS_VALUE] == FLAGGED)
      flaggedCells.add(cell);
  return flaggedCells;
}
```

Notice that the simplicity of the code has not changed. It still has exactly the same number of operators and constants, with exactly the same number of nesting levels. But the code has become much more explicit.

We can go further and write a simple class for cells instead of using an array of `int`s. It can include an intention-revealing function (call it `isFlagged`) to hide the magic numbers. It results in a new version of the function:

```
public List<Cell> getFlaggedCells() {
  List<Cell> flaggedCells = new ArrayList<Cell>();
  for (Cell cell : gameBoard)
    if (cell.isFlagged())
      flaggedCells.add(cell);
  return flaggedCells;
}
```

With these simple name changes, it's not difficult to understand what's going on. This is the power of choosing good names.

Avoid Disinformation

Programmers must avoid leaving false clues that obscure the meaning of code. We should avoid words whose entrenched meanings vary from our intended meaning. For example, `hp`, `aix`, and `sco` would be poor variable names because they are the names of Unix platforms or variants. Even if you are coding a hypotenuse and `hp` looks like a good abbreviation, it could be disinformative.

Do not refer to a grouping of accounts as an `accountList` unless it's actually a `List`. The word list means something specific to programmers. If the container holding the accounts is not actually a `List`, it may lead to false conclusions.[1] So `accountGroup` or `bunchOfAccounts` or just plain `accounts` would be better.

1. As we'll see later on, even if the container *is* a `List`, it's probably better not to encode the container type into the name.

Beware of using names which vary in small ways. How long does it take to spot the subtle difference between a `XYZControllerForEfficientHandlingOfStrings` in one module and, somewhere a little more distant, `XYZControllerForEfficientStorageOfStrings`? The words have frightfully similar shapes.

Spelling similar concepts similarly is *information*. Using inconsistent spellings is *disinformation*. With modern Java environments we enjoy automatic code completion. We write a few characters of a name and press some hotkey combination (if that) and are rewarded with a list of possible completions for that name. It is very helpful if names for very similar things sort together alphabetically and if the differences are very obvious, because the developer is likely to pick an object by name without seeing your copious comments or even the list of methods supplied by that class.

A truly awful example of disinformative names would be the use of lower-case `L` or uppercase `O` as variable names, especially in combination. The problem, of course, is that they look almost entirely like the constants one and zero, respectively.

```
int a = l;
if ( O == l )
  a = O1;
else
  l = 01;
```

The reader may think this a contrivance, but we have examined code where such things were abundant. In one case the author of the code suggested using a different font so that the differences were more obvious, a solution that would have to be passed down to all future developers as oral tradition or in a written document. The problem is conquered with finality and without creating new work products by a simple renaming.

Make Meaningful Distinctions

Programmers create problems for themselves when they write code solely to satisfy a compiler or interpreter. For example, because you can't use the same name to refer to two different things in the same scope, you might be tempted to change one name in an arbitrary way. Sometimes this is done by misspelling one, leading to the surprising situation where correcting spelling errors leads to an inability to compile.[2]

It is not sufficient to add number series or noise words, even though the compiler is satisfied. If names must be different, then they should also mean something different.

2. Consider, for example, the truly hideous practice of creating a variable named `klass` just because the name `class` was used for something else.

Number-series naming (`a1, a2, .. aN`) is the opposite of intentional naming. Such names are not disinformative—they are noninformative; they provide no clue to the author's intention. Consider:

```
public static void copyChars(char a1[], char a2[]) {
  for (int i = 0; i < a1.length; i++) {
    a2[i] = a1[i];
  }
}
```

This function reads much better when `source` and `destination` are used for the argument names.

Noise words are another meaningless distinction. Imagine that you have a `Product` class. If you have another called `ProductInfo` or `ProductData`, you have made the names different without making them mean anything different. `Info` and `Data` are indistinct noise words like `a`, `an`, and `the`.

Note that there is nothing wrong with using prefix conventions like `a` and `the` so long as they make a meaningful distinction. For example you might use `a` for all local variables and `the` for all function arguments.[3] The problem comes in when you decide to call a variable `theZork` because you already have another variable named `zork`.

Noise words are redundant. The word `variable` should never appear in a variable name. The word `table` should never appear in a table name. How is `NameString` better than `Name`? Would a `Name` ever be a floating point number? If so, it breaks an earlier rule about disinformation. Imagine finding one class named `Customer` and another named `CustomerObject`. What should you understand as the distinction? Which one will represent the best path to a customer's payment history?

There is an application we know of where this is illustrated. we've changed the names to protect the guilty, but here's the exact form of the error:

```
getActiveAccount();
getActiveAccounts();
getActiveAccountInfo();
```

How are the programmers in this project supposed to know which of these functions to call?

In the absence of specific conventions, the variable `moneyAmount` is indistinguishable from `money`, `customerInfo` is indistinguishable from `customer`, `accountData` is indistinguishable from `account`, and `theMessage` is indistinguishable from `message`. Distinguish names in such a way that the reader knows what the differences offer.

Use Pronounceable Names

Humans are good at words. A significant part of our brains is dedicated to the concept of words. And words are, by definition, pronounceable. It would be a shame not to take

3. Uncle Bob used to do this in C++ but has given up the practice because modern IDEs make it unnecessary.

advantage of that huge portion of our brains that has evolved to deal with spoken language. So make your names pronounceable.

If you can't pronounce it, you can't discuss it without sounding like an idiot. "Well, over here on the bee cee arr three cee enn tee we have a pee ess zee kyew int, see?" This matters because programming is a social activity.

A company I know has `genymdhms` (generation date, year, month, day, hour, minute, and second) so they walked around saying "gen why emm dee aich emm ess". I have an annoying habit of pronouncing everything as written, so I started saying "gen-yah-mudda-hims." It later was being called this by a host of designers and analysts, and we still sounded silly. But we were in on the joke, so it was fun. Fun or not, we were tolerating poor naming. New developers had to have the variables explained to them, and then they spoke about it in silly made-up words instead of using proper English terms. Compare

```
class DtaRcrd102 {
  private Date genymdhms;
  private Date modymdhms;
  private final String pszqint = "102";
  /* ... */
};
```

to

```
class Customer {
  private Date generationTimestamp;
  private Date modificationTimestamp;;
  private final String recordId = "102";
  /* ... */
};
```

Intelligent conversation is now possible: "Hey, Mikey, take a look at this record! The generation timestamp is set to tomorrow's date! How can that be?"

Use Searchable Names

Single-letter names and numeric constants have a particular problem in that they are not easy to locate across a body of text.

One might easily grep for `MAX_CLASSES_PER_STUDENT`, but the number 7 could be more troublesome. Searches may turn up the digit as part of file names, other constant definitions, and in various expressions where the value is used with different intent. It is even worse when a constant is a long number and someone might have transposed digits, thereby creating a bug while simultaneously evading the programmer's search.

Likewise, the name `e` is a poor choice for any variable for which a programmer might need to search. It is the most common letter in the English language and likely to show up in every passage of text in every program. In this regard, longer names trump shorter names, and any searchable name trumps a constant in code.

My personal preference is that single-letter names can ONLY be used as local variables inside short methods. *The length of a name should correspond to the size of its scope*

[N5]. If a variable or constant might be seen or used in multiple places in a body of code, it is imperative to give it a search-friendly name. Once again compare

```
for (int j=0; j<34; j++) {
  s += (t[j]*4)/5;
}
```

to

```
int realDaysPerIdealDay = 4;
const int WORK_DAYS_PER_WEEK = 5;
int sum = 0;
for (int j=0; j < NUMBER_OF_TASKS; j++) {
  int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;
  int realTaskWeeks = (realdays / WORK_DAYS_PER_WEEK);
  sum += realTaskWeeks;
}
```

Note that `sum`, above, is not a particularly useful name but at least is searchable. The intentionally named code makes for a longer function, but consider how much easier it will be to find `WORK_DAYS_PER_WEEK` than to find all the places where 5 was used and filter the list down to just the instances with the intended meaning.