4

Meaningful Names

by Tim Ottinger and Uncle Bob

Beginning in the late ’90s I ran a company named Object Mentor Inc. Our website sported a number of technical documents that we offered to the software community. The most downloaded of all those documents was Tim Ottinger’s rules for names. What you are about to read is the descendant and natural evolution of Tim’s classic document.

—Uncle Bob

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 `dll` files and `gem` files. We name and name and name; and because we do so much of it, we’d better do it well.

It might seem that names are so subjective that defining the principles of "goodness" is a fool's errand. However, despite their subjectivity, we know of good restaurants, good movies, good paintings, and good books.

Subjectivity is tempered by the realization that any work is good if it is accepted by its audience. This means that we aren't dealing with individual preferences and prejudices, but rather, with the desires of a collective audience.

The audience for this book is the set of programmers who are interested enough in improving their code craft to purchase, borrow, or in some other way acquire a copy of this book. If the writing reaches you and helps you, then we've written good-enough words in a good-enough order.

As an author of code, your audience is the set of people who will work in your codebase. You are not only an author, but also a member of the codebase community. The audience is larger than you are, and they collectively will decide what is welcome and what is unwelcome in the codebase.

Write for your audience, not for yourself. You must remember that your experience of writing the code is less important and more transient than their experience of reading that code.

That said, we find there is a fairly consistent band of preferences across many audiences, and this generalized advice is intended to help you find that "sweet spot" with your current software audience.

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.

To help you in that effort, here are some of the more useful recommendations we have found.

Use Intention-Revealing Names

The name of a variable, function, or class should answer all the big questions. It should tell its audience why it exists, what it does, and how it is used.

If a name requires a comment to communicate to its audience, then the name does not reveal its intent.

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

The name `d` reveals very little. It does not evoke a sense of elapsed time; specifically, of days. We should choose a name that specifies what is being measured and the unit of that measurement. One of these might be better: `elapsedDays` , `daysSinceCreation` , `daysSinceModification` , or `fileAgeInDays` .

These examples are longer than the original name (`d`), but that doesn't necessarily mean that long names are better than short names, only that clear names are better than unclear ones. For example:

```
int d = getElapsedTimeInDays();
```

Here the name is perfectly clear since it is defined by the context and would be appropriate if the scope that contains the name is relatively small. (See "Use Names of Appropriate Length.")

Build a System of Names

Choosing names that reveal intent can make it much easier to understand and change code. What follows is an example that breaks this rule. Can you determine 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. The problem is that the code provides no context. The code implicitly requires that we know the answers to questions such as

- What kinds of things are in `theList` ?
- What does the zeroth subscript signify?
- What is the meaning of the value 4?
- How would I use the list being returned?

The answers to these questions are not present in the code sample. However, a good system of names would have made them clear.

Let’s say that code came from the old *Minesweeper* game.[1] The *Minesweeper* game board is a list of cells called `theList` . Let’s rename that to `gameBoard` .

---

1. A game from the ’90s that shipped with Windows 3.11.

Each cell on the board is represented by a primitive array. The zeroth subscript is the location of a status value, and 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. Only the clarity has improved.

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 through naming.

Note also that if this was the very first bit of code you came across, you would already know that this was a game with a board of cells that had statuses, one of which was “flagged.” A good system of names communicates a great deal about the application as a whole.

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. For example, 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 an array or a JSON document, it may lead to false conclusions.[2] So `accountGroup` or `bunchOfAccounts` or just plain `accounts` may be shorter (and thereby easier to take in at a glance) and more honest.

2. 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 that vary in small ways. How long does it take to spot the subtle difference between an

`XYZControllerForEfficientHandlingOfStrings` in one module and, somewhere a little more distant, an `XYZControllerForEfficientStorageOfStrings` ? The words have frightfully similar shapes. Would you realize that they were different if they weren't right next to each other?

Those names are doubly difficult because they are long and have similar beginnings. Can you spot the optical illusion below?

```
XYZControllerForEfficientHandlingOfStrings
XXZControllerForEfficientHandlingOfStrings
```

Spelling similar concepts similarly is *information*. Using inconsistent spellings is *disinformation*. With modern environments, we enjoy automatic code completion. We write a few characters of a name and are quickly rewarded with a list of possible completions for that name. Inconsistent spellings will confound that list.

Modern environments also help by using different fonts, styles, and colors for different syntactic elements. Still, it is wise to be careful of some of the optical illusions inherent in our fonts. For example, be careful with the use of lowercase *L* and uppercase *O*. In certain fonts, they can be indistinguishable from the constants 1 and 0, respectively.

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

Combining these letters with numbers and `x` can make it worse. Consider `Oxll` as a name.

Make Meaningful Distinctions

Programmers create problems for themselves when they write code solely to satisfy a compiler or interpreter. For example, when you want to 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 to keep them distinct.

Sometimes this is done by misspelling one, leading to the surprising situation where correcting spelling errors leads to an inability to compile.[3]

---

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

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.

Number-series naming (`a1` , `a2` , ... `aN`) is the opposite of intentional naming. Such names are not dis-informative, they are uninformative; 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.

In general, when you see variables named `X1` , `X2` , `X3` , the code is practically begging you for either an array or a data structure. Consider what makes them so similar that they have the same name and only a numeric difference, and look for some meaning that will inspire either different names or different structures entirely.

In the code above, it was just lazy naming. Changing those names to `source` and `destination` makes a better solution than trying to create a structure, class, or container. Not all problems are deep.

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.

They add no new information, because every class contains “data” and every class contains “info.”

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 likely 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?

Here are some disturbingly meaningless function names we found in one particular application.

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

How are the programmers in this project supposed to know which of these functions to call? How many active accounts are there? Which of them are returned by the first function? And what is the difference between the second and third functions? In any universe, this “system” of names is a nightmare.

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 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 conveniently discuss it. “Well, over here on the bee cee arr three cee enn tee we have a pee ess zee kyew int, see?” Convenience of discussion matters because programming is a social activity.

One company had the name `genymdhms` (generation date, year, month, day, hour, minute, and second), so they walked around saying “gen why emm dee aich emm ess.” This in inconvenient, so the pronunciation gradually evolved into “gen-yah-mudda-hims.” This might sound silly, but they were all in on the joke, so it was fun. Fun or not, they 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 generated;
  private Date modified;
  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 filenames, as 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 not a searchable name. It is the most common letter in the English language and likely to show up in every passage of text in every program. In general, short names are less searchable than long names, which is one more reason that when it comes to being searchable, longer names can trump shorter names, and any searchable name trumps a constant in code.

The name `e` is often used to denote the specific error that was raised in a `try` / `catch` block, especially in Python and Go. In this case, though, the name is familiar in its context to many programmers, and it occurs in such a small scope as to make searches unnecessary. Short names in limited and well-known circumstances generally don't confound searches.

Our preference is that very short, or single-letter, names ought *only* to be used as local variables inside very short scopes. If a variable or constant might be seen or used in multiple places in a body of code, it is best 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] * realDaysPerIc
   int realTaskWeeks = (realTaskDays / WORK_DAYS_PER_
   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.

Use Names of Appropriate Length

Our rules for the length of names may surprise you, but they have withstood the test of time.

Variable Names

We have a simple rule for the names of variables. The length of a variable’s name should be proportional to the length of the scope that contains it. For example:

```
for (int n=0; n<20; n++)
  System.out.println("" + n + " " + n*n);
```

The name `n` is appropriately short because its scope is two lines long. We would find it annoying if that variable were named `theNumberToBeSquared`.

On the other hand, in a longer scope, such small names can be opaque. We would usually find an instance variable named `n` annoying; especially if `m`, `p`, `i`, and `j` were also instance variables.

As the size of the scope increases, so the length of the variable names within that scope should increase. Arguments and local variables are scoped to a single function and so should have relatively short names; perhaps a word or two. Instance variables are scoped to a whole class. Their names should be relatively longer. Global variable names should be longer still.

Function Names

Our rule for functions is exactly the opposite. The length of their names should be inversely proportional to the length of their scope. Thus, global functions will have short names, while private methods within a class will have relatively long names. Tests, which have the shortest scope of all (effectively zero), have the longest names of all.

We justify this in two ways. First, the larger the scope of a function, the more frequently it will be used. So, for the sake of convenience, we’d like the name to be short. We would find it quite annoying if the function that opens a file was named `openFileAndThrowExceptionIfNotFound` instead of `open` .

Second, the larger the scope of a function, the more general it is. Again, `open` is a very general function. Methods within a class have a much narrower context and so will likely need more words to explain their intent. Private methods within a class have very small scopes and are called from very few places, or even just one. They are not general at all and may therefore require many words to describe properly. Their names should be correspondingly longer.

Class Names

Our rule for classes is the same as our rule for functions. The longer the scope, the shorter the name. Classes at the global scope should have relatively short names (a word or two), while derived classes, inner classes, and private classes should have longer names.

Property and Attribute Names

Some languages, like C++ and C#, have elements that look like variables but are actually functions. They look like variables because they can be put on the left of assignment statements. Therefore, they should follow the variable-naming rule. The length of their names should be proportional to the scope that contains them.

Namespace Names

Most languages now allow us to put our classes, functions, and variables into namespaces. Namespace names should be descriptive. Thus, they are likely to be longish and are often part of a hierarchy of namespaces. Consider, for example, the namespace from a space war game[4] Uncle Bob wrote a few years back: `spacewar.ui.tactical-scan` . Such long and qualified names are generally inconvenient, which is why most languages that have namespaces have some way to create namespace aliases that can be used within smaller scopes.

---

4. https://github.com/unclebob/spacewar

Thus, while the length of namespace names is not a function of the scope that contains them, the length of their aliases is. In a very short module, we might use the alias `ts`, whereas in a longer module, we might use `tac-scan`, and in a very long module (which ought to be rare), we might use `tactical-scan`.

Avoid Encodings

We have enough encodings to deal with without adding more to our burden. Encoding type or scope information into names simply adds an extra burden of deciphering. It hardly seems reasonable to require each new employee to learn yet another encoding "language" in addition to learning the (usually considerable) body of code that they'll be working in. On top of that, encoded names are seldom pronounceable and are easy to mistype.

In days of old, when we worked in name length–challenged languages, we violated this rule out of necessity and with regret. Fortran forced encodings by making the first letter a code for the type. Early versions of BASIC allowed only a letter plus one digit. Hungarian notation[5] (HN) took this to a whole new level.

5. So named because it was invented by Charles Simonyi, a Hungarian-born architect at Microsoft during the DOS era.

HN was considered to be pretty important back in the Windows C API, when everything was an integer handle or a long pointer or a void pointer, or one of several implementations of "string" (with different uses and attributes). The C compiler did not check types in those days, so the programmers needed a crutch to help them remember the types.

In modern languages, we have much richer type systems, and the compilers remember and enforce the types. What's more, there is a trend toward smaller classes and shorter functions so that people can usually see the point of declaration of each variable they're using.

Today's programmers get enough help from their compilers and IDEs that they don't usually need to encode types within their variable names. So nowadays, HN and other forms of type encoding are simply impediments.

They make it harder to change the name or type of a variable, function, or class. They make it harder to read the code. And they create the possibility that the encoding system will mislead the reader.

```
PhoneNumber phoneString;
// name not changed when type changed!
```

Member Prefixes

Prefixing member variables with something like `m_` is another form of type encoding that is no longer necessary. Your classes and functions should be small enough that you don’t need such a crutch. You should be using an editing environment that highlights or colorizes members to make them distinct.

```
public class Part {
  private String m_dsc; // The textual description
  void setName(String name) {
    m_dsc = name;
  }
}
```

```
public class Part {
  String description;
  void setDescription(String description) {
    this.description = description;
  }
}
```

Interfaces and Implementations

We do not recommend prefixing interfaces with `I`, because that’s just another kind of type encoding. For example, say you are building an Abstract Factory[6] for the creation of `Shape` s. This factory will be an interface and will be implemented by a concrete class. What should we name these classes? `IShapeFactory` and `ShapeFactory` ? We prefer to leave interfaces unadorned. The preceding `I` is a distraction at best and too much information at worst. After all, the whole point of an interface is that users don't need to know it's an interface. We just want them to know that it's a `ShapeFactory`. So, if we must encode either the interface or the implementation, we prefer to add a suffix to the implementation; for example, `ShapeFactoryImp`.

---

6. [GOF95].

Use Appropriate Parts of Speech

Some elements of our code are best represented by nouns, and others by verbs.

Class Names

Classes and objects should generally have noun or noun phrase names like `Customer`, `WikiPage`, `Account`, and `AddressParser`. Generally the noun should be singular and not plural. A class that represents customers should be named `Customer` and not `Customers`. On the other hand, a class that holds many `Customer` instances probably *should* be called `Customers`.

Avoid noise words like `Manager`, `Processor`, `Data`, or `Info` in the name of a class. We already know that classes manage and process data and info—you don't need to tell us.

Method Names

Methods should generally have verb or verb phrase names like `postPayment`, `deletePage`, or `save`.

Depending on your language, accessors, mutators, and predicates should be named according to the local convention. In Java, they are named for their value and prefixed with `get`, `set`, and `is` according to the JavaBeans standard.[7]

---

7. http://java.sun.com/products/javabeans/docs/spec.html

There is a de facto convention in some languages to name a function after the value it returns, that is, as a noun. For example, the function that returns the name of a customer might be called `name`. We do not prefer this convention, but we will follow it when working with teams that use it.

In most cases, it's best if a mutator is named for the mutation it causes, such as `add`, `remove`, `sort`, and so on. These tend to be verb names, and so the standard advice is to "name methods with verb names."

```
string name = employee.getName();
customer.setName("mike");
if (paycheck.isPosted())…
```

C++, Java, and C# have constructors that take the name of the class. This means you cannot give the constructor a meaningful name. When such constructors are overloaded, it is better to use static factory methods with names that describe the arguments. For example:

```
Complex fulcrumPoint = Complex.FromRealNumber(23.0);
```

is generally better than

```
Complex fulcrumPoint = new Complex(23.0);.
```

Consider enforcing their use by making the corresponding constructors private.

Consider Keyword Parameters

If your language allows keyword or hash map parameters, such as Python, C#, Ruby, and Clojure do, you may find it best to limit the positional parameters of a function and use explicit keyword-only parameters for most variations.

```
def sort(iterable, *, key=None, reverse=False ) :
  … # body elided
pets_sorted = sort(pets)
```

```
pets_reversed = sort(pets, reverse=True)
pets_by_age = sort(pets, key=lambda x: x.birthdate()
```

Don't Be Cute

If names are too clever, they will be memorable only to people who share the author's sense of humor, and only as long as these people remember the joke. Will they know what the function named `HolyHandGrenade` is supposed to do? Sure it's cute, but maybe in this case `DeleteItems` might be a better name. Choose clarity over entertainment value.

Cuteness in code often appears in the form of colloquialisms or slang. For example, don't use the name `whack()` to mean `kill()`. Don't tell little culture-dependent jokes like `eatMyShorts()` to mean `abort()`.

Say what you mean. Mean what you say.

Pick One Word per Concept

Pick one word for one concept, and stick with it. For example, don't use `amount` in one module, `fee` in another, `price` in yet another, and `cost` in still another. Pick a name for each concept and use that name throughout. Make sure that everyone on the team knows those names.

We will often audit the names in our systems for consistency. This is a small task, and often an IDE will have the option to provide an "outline view" or "object view," or else "fold/collapse all functions" so that you can look at method and variable names without scrolling. We highly recommend harmonizing all the names in a given class, module, or test suite. You will find it satisfying, we have no doubt.

A consistent lexicon (A Ubiquitous Language[8]) is a great boon to the programmers who must use your code.

---

8. [DDD].

Use Solution Domain Names

Remember that the people who read your code will be programmers. They will be familiar with solution domain terms.

When you are underneath the domain level and into pure implementation, go ahead and use computer science (CS) terms, algorithm names, pattern names, math terms, and so forth. It is not wise to draw every name from the problem domain, because we don't want our coworkers to have to run back and forth to the customer asking what every name means when they already know the concept by a different name.

The name `AccountVisitor` means a great deal to a programmer who is familiar with the Visitor[9] pattern. What programmer would not know what a `JobQueue` or `FIFO` is?

---

9. [GOF95].

There are lots of very technical things that programmers have to do. Choosing technical names for doing technical things is usually the most appropriate course.

Programs are often best described using programming. You don't have to make everything look like either English prose or mathematical formulae. Use the level of abstraction that works best for the algorithm and the audience.

Use Problem Domain Names

Use names from the problem domain. They are likely to be familiar and help set the context for developers on your team. If not, then at least the programmer who maintains your code can ask a domain expert what it means.

Separating solution and problem domain concepts is part of the job of a good programmer and designer. The code that has more to do with problem domain concepts should have names drawn from the problem domain. We don't force solution domain names on the problem, nor problem domain names on the solutions.

Add Meaningful Context

There are a few names that are meaningful in and of themselves—most are not. Instead, you need to place names in context for your reader by enclosing them in well-named classes, functions, or namespaces.

Imagine that you have variables named `firstName`, `lastName`, `street`, `houseNumber`, `city`, `state`, and `postcode`. Taken together, it's pretty clear that they form an address. But what if you just saw the `state` variable being used alone in a method? Would you automatically infer that it was part of an address, or would you infer that you are in a finite state machine and tracking which state that machine was currently in?

You can remove this ambiguity by creating a class or data structure named `Address`. Then, even the compiler will know that the variables belong to a bigger concept.

Programmers sometimes fall into the code smell of Primitive Obsession,[10] avoiding the creation of classes, structures, and module for too long. It is much better, for example, to work with `GlobalLocations` than to deal with primitive variables for `latitude` and `longitude` in hours, minutes, and seconds. It is far easier to work with two "locations" than with 12 floating-point integers, all similarly named.

---

10. One of the code smells in [Refactoring].

As another example, consider the following Python code. Do the variables need a more meaningful context? The function name provides only part of the context; the algorithm provides the rest.

```
private void printGuessStatistics(char candidate, int
    String number;
    String verb;
    String pluralModifier;
    if (count == 0) {
      number = "no";
      verb = "are";
      pluralModifier = "s";
```

```
        } else if (count == 1) {
          number = "1";
          verb = "is";
          pluralModifier = "";
        } else {
          number = Integer.toString(count);
          verb = "are";
          pluralModifier = "s";
        }
        String guessMessage = String.format(
          "There %s %s %s%s", verb, number, candidate, pl
        );
        print(guessMessage);
      }
```

Once you read through the function, you see that the three variables— `number` , `verb` , and `pluralModifier` —are part of the "guess statistics" message. Unfortunately, the context must be inferred. When you first look at the method, the meanings of the variables are opaque.

The function is a bit too long and the variables are used throughout. To split the function into smaller pieces we need to create a `GuessStatisticsMessage` class and make the three variables fields of this class. This provides a clear context for the three variables. They are definitively part of the `GuessStatisticsMessage` . The improvement of context also allows the algorithm to be made much cleaner by breaking it into many smaller functions.

```
    public class GuessStatisticsMessage {
      private String number;
      private String verb;
      private String pluralModifier;
      public String make(char candidate, int count) {
        createPluralDependentMessageParts(count);
        return String.format(
          "There %s %s %s%s",
           verb, number, candidate, pluralModifier );
      }
      private void createPluralDependentMessageParts(int
        if (count == 0) {
          thereAreNoLetters();
        } else if (count == 1) {
          thereIsOneLetter();
        } else {
          thereAreManyLetters(count);
        }
      }
      private void thereAreManyLetters(int count) {
        number = Integer.toString(count);
        verb = "are";
        pluralModifier = "s";
      }
      private void thereIsOneLetter() {
        number = "1";
        verb = "is";
        pluralModifier = "";
      }
      private void thereAreNoLetters() {
        number = "no";
        verb = "are";
        pluralModifier = "s";
      }
    }
```

Don’t Add Gratuitous Context

In an imaginary application called Gas Station Deluxe, it is a bad idea to prefix every class with `GSD`. Frankly, you are working against your tools. You type `G` and press the completion key and are rewarded with a mile-long list of every class in the system. Is that wise? Why make it hard for the IDE to help you?

Likewise, say you invented a `MailingAddress` class in `GSD`’s `accounting` module, and you named it `GSDAccountAddress`. Later, you need a mailing address for your customer contact application. Do you use `GSDAccountAddress`? Does it sound like the right name? Ten of 17 characters are redundant or irrelevant.

The names `accountAddress` and `customerAddress` are fine names for instances of the class `Address` but could be poor names for classes. `Address` is a fine name for a class. If we need to differentiate between MAC addresses, port addresses, and Web addresses, we might consider `PostalAddress`, `MAC`, and `URI`. The resulting names are more precise, which is the point of all naming.

Final Words

The hardest thing about choosing good names is that it requires good descriptive skills and a shared cultural background. It’s not a self-pleasing, universal, learn-one-time-and-run-on-autopilot kind of skill.

Each time we change codebases we have to start the process over. Who are the people who comprise our audience now? What does “everyone know” in this space? What is familiar and what seems “weird” or “alien” here?

There is a degree of subjectivity here, as you write for an audience. What you like personally may not be what they need and understand. As a result, many programmers don’t learn to do it very well. It requires contact with your user base, just like any good software development requires contact with real users.

Since it’s hard to get it right (even with these fine guidelines), we often have to iterate on our software names. Sometimes other developers are surprised that we, after having written so many times and so many words on naming, will often call a variable by an intentionally bad name temporarily, until a better name comes to mind.

We rename things. As Josh Kerievsky reminds us, *all good writing is based on revision*.

People may defer renaming things for fear that some other developers will object. We do not share that fear and find that we are actually grateful when names change (for the better).

Most of the time we don’t memorize the names of classes and methods and demand that they remain the same. Memorization is a waste of our time. We use modern tools to deal with details like that. We focus more on whether the code reads like paragraphs and sentences, or at least like equations, tables, and data structures.

You may surprise someone with a name change, just like you might with any other code improvement. Don’t let it stop you in your tracks.

Try these guidelines to see if they help in your situation.

So far, they seem to help most people most of the time. We’re confident that you will also find help here.