Introduction

Source code is a form of expression and so quality-of-expression must be a primary concern. Poor quality-of-expression creates ongoing costs and impacts the productivity (hence the reputation) of the team that produces it. Therefore software authors must learn to clearly express the intention of variables, methods, classes, and modules.

Furthermore, maintenance errors occur when a programmer does not correctly understand the code he must modify. The diligent programmer may expend considerable effort to accurately decipher the true intent of a program, even though its purpose was obvious to the original author. Indeed, it takes much more time to determine how to change the code than to make and test the change.

Some efforts to improve maintainability of code have actually exacerbated the problem by making names less intuitive, less obvious, and less natural. They make it easier to glean the information that *was not* lost (like type information and scope, still present in variable declarations), without making it easier to discover the more important information that *was* lost.

An expressive name for a software object must be clear, precise, and small. This is a guide to better clarity, precision, and terseness in naming. It is the author's hope that you will find it useful.

Use Intention-revealing Names

We often see comments used where better naming would be appropriate:

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

Such a comment is an excuse for not using a better variable name. The name 'd' doesn't evoke a sense of time, nor does the comment describe what time interval it represents. It requires a change:

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

The simple act of using a better name instead of applying a better comment can reduce the difficulty of working with the code we write.

What is the purpose of this python code?

```
list1 = []
for x in theList:
    if x[0] == 4:
        list1 += x;
return list1
```

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

The problem isn't the simplicity of the code but the *implicity* of the code: the degree to which the context is not explicit in the code itself. The code requires me to answer questions such as:

- What kinds of things are in `theList`?
- What is the significance of the zeroeth subscript of an item in `theList`?
- What is the significance of the value `4`?
- How would I use the list being returned?

This information is not present in the code sample, but it 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 `theBoard`.

Each cell on the board is represented by a simple array. We further find that the zeroeth 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:

```
flaggedCells = []
for cell in theBoard:
    if cell[STATUS_VALUE] == FLAGGED:
        flaggedCells += cell
return flaggedCells
```

Notice that the simplicity of the code is not changed. It still has exactly the same number of operators and constants, with exactly the same number of nesting levels.

We can go further and write a simple class for cells instead of using an array of ints. The expression used to check that a cell has been flagged can be renamed by adding an intention-revealing function (call it `isFlagged`) to hide the magic numbers. It results in a new version of the function:

```
flaggedCells = []
for cell in theBoard:
    if cell.isFlagged():
        flaggedCells += cell
return flaggedCells
```

or more tersely:

```
return [ cell for cell in theBoard if cell.isFlagged() ]
```

Even with the function collapsed to a list comprehension, it's not difficult to understand. My original four questions are answered fully, and the *implicity* of the code is reduced. This is the power of naming.

Avoid Disinformation

A software author must avoid leaving false clues which 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 is 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 CS people. If the container holding the accounts is not actually a list, it may lead to false conclusions. `AccountGroup` or `BunchOfAccounts` would have been better.

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 shape.

With modern Java environments, you have automatic code completion. You will write a few characters of a name and press some hot key combination (if that) and you will be greeted with a list of possible completions for that name. It is nice if names for very similar things sort together alphabetically, and if the differences are very, very obvious since 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 dis-informative 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 the author has examined code where such things were abundant. It's a great technique for shrouding code. 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 if an author performs a simple renaming.

Make Meaningful Distinctions

A problem arises from writing code solely to satisfy a compiler or interpreter. One can't have the same name referring to two things in the same scope, so one name is changed them 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.

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. Without being disinformative, number series names provide no clue to the intention of the author. Naming by intention and by domain may lend one to use names like `lvalue` and `rvalue` or `source` and `destination` rather than `string1` and `string2`.

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".

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 I know of where this is illustrated. I've changed the names to protect the guilty, but the exact form of the error is:

```
        getSomething();
        getSomethings();
        getSomethingInfo();
```

Consider context. In the absence of a class named Denomination, `MoneyAmount` is no better than `money`. `CustomerInfo` is no better than `Customer`.

Disambiguate in such a way that the reader knows what the different versions offer her, instead of merely that they're different.

Use Pronounceable Names

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.

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

```
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 somehow correspond to the size of its scope. 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.

This rule is less important if the entire team always uses an intelligent IDE that can show you all instances of a given variable without doing a text search, but you will find that even such projects often resort to less elegant search solutions, especially when combating strange build errors.

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


// Cluttered, but more obvious
int realDaysPerIdealDay = 4;
const int WORK_DAYS_PER_WEEK = 5;
/*...*/
for (int j=0; j < NUMBER_OF_TASKS; j++) {
    int realDays = estimate[j] * realDaysPerIdealDay;
    int realWeeks = (realdays/WORK_DAYS_PER_WEEK);
    sum+= realWeeks;
}
```

Note that "sum", above, is not a particularly useful name, but at least is searchable. The intentionally-named code makes for a longer and possibly more cluttered 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.

Avoid Encodings

Encoded names require deciphering. It hardly seems reasonable to require each new employee to learn an encoding "language" in addition to learning the (usually considerable) body of code that they'll be working in. It is an unnecessary mental burden when trying to solve a problem. Encoded names are seldom pronounceable and are easy to mistype.

In days of old, when you worked in name-length-challenged languages, you probably violated this rule with impunity and regret. Fortran forced it by basing type on the first letter, making the first letter a code for the type. Early versions of BASIC allowed only a letter plus one digit. Hungarian Notation (HN) has taken this to a whole new level

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). In *modern* languages we have much richer type systems. There is a trend toward smaller classes and shorter functions so that one can always see the point of declaration of each variable they're using.

Modern languages don't need type encoding. Objects are strongly (and often dynamically) typed, and editing environments have advanced. You may feel naked without type encoding, but with "duck typing" your type-encoded variable names will usually be wrong anyway. You don't need it, can't do it reliably, and can learn to live without it.

You don't need to prefix member variables with "m_" anymore.

```
class Item{
     String m_psz; // The textual description
     void setName(String psz) {
        m_psz = psz;
     }
  }
```

A reasonable replacement might be:

```
class Item {
     String description;
     void setDescription(String text) {
        this.description = text;
     }
  }
```

The other problem with prefix encoding is that people quickly learn to look past the prefix (or stop short of the suffix) to the meaningful part of the name. Those carefully crafted name systems end up being ignored (almost) totally. Eventually the prefixes are left unmaintained just as comments are left unmaintained.

```
  std::wstring pszName = L"hello"; // name not changed when type changed!
```

Avoid Mental Mapping

Readers shouldn't have to mentally translate your names into other names they already know. This problem generally arises from a choice to use neither problem domain terms nor solution domain terms.

This is a problem with single-letter variable names. Certainly a loop counter may be named `i` or `j` or `k` (though never `l`!) if its scope is very, very small and no other names can conflict with it. These are allowable because those are traditional solution-domain names. But generally, a single letter name *stands for* something else, or else has no meaning at all. There can be no worse reason for using the name `'c'` than *because* `a` *and* `b` *were already taken*.

This has been a a problem with really smart programmers who can manipulate more than the standard 7-plus-or-minus-two symbols easily in their heads. They map so easily among twelve or fifteen arbitrarily-named objects that they can't image anything easier than the simple fact that r is the lower-cased version of the url with the host and scheme removed.

Smart is overrated. Clarity is king. The very smart must use their talent to write code that others are less likely to misunderstand.

Use Noun and Verb Phrases

Classes and objects should have noun or noun phrase names.

There are some methods (commonly called "accessors") which calculate and/or return a value. These reasonably may have noun names. This way accessing a person's first name can read as follows:

```
        string x = person.name();
```
because it is as obvious as:
```
        string wholeName = person.getName();
```
Other methods (sometimes called "mutators", though not so commonly anymore) cause something to happen. These represent a small "transaction" on the object (and generally should be a complete action). Mutators should have verb or verb-phrase names. This way, changing a name would read:
```
        fred.changeNameTo("mike");
```

You will notice that the above line reads more like a sentence than a lot of code. It leaves a dangling "to", which is completed by the parameter. The intention is to make the parameter list more obvious so that it is harder to make foolish errors.

Another trend is to use a named creation function, rather than yet another overloaded constructor. It can be more obvious to read the creation of a complex number using
```
    Complex.FromRealNumber(23.0);
```
than using the constructor version
```
    new Complex(23.0);
```

As a class designer, does this sound boringly unimportant? If so, then go write code that uses your classes. The best way to test an interface is to use it and look for ugly, contrived, or confusing text. The most popular way to do this in the 21st century is to write *Unit Tests* for the module. If you have trouble reading the tests (or your partners do) then rework is in order.

Over time we've found that this rule extends even to constructors. Rather than having a lot of overloaded constructors and having to chose among them by their parameter lists, we frequently create named creation functions as class (static) methods.

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 they remember the joke. Will the people 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. Save the comedy for comments, where busy people don't have to read them.

Choose clarity over entertainment value.

Pick One Word Per Concept

Pick one word for one abstract function and stick with it. I hear that the Eiffel libraries excel at this, and I know that the C++ STL is very consistent. Sometimes the names seem a little odd (like `pop_front` for a list), but being consistent will reduce the overall learning curve for the whole library.

For instance, it's confusing to have `fetch`, `retrieve` and `get` as equivalent methods of the different classes. How do you remember which method name goes with which class? Sadly, you often have to remember which company, group, or individual wrote the library or class in order to remember which term was used. Otherwise, you spend an awful lot of time browsing through headers and previous code samples.

It's not as bad as it used to be. Modern editing environments like Eclipse, IntelliJ, Eric, and the like will provide hints to you. You type the name of an object and a dot, and up pops a list of methods you can call. But note that the list doesn't give you the comments you wrote around your function names and parameter lists. You are lucky if it gives the parameter names from function declarations. The function names have to stand alone, and they have to be consistent in order for you to pick the correct method without any additional exploration.

Likewise, it's confusing to have a `controller` and a `manager` and a `driver` in the same process. What is the essential difference between a DeviceManager and a ProtocolController? Why are both not controllers, or both not managers? Are they both Drivers, really? The name leads you to expect two objects that have very different *type* as well as having different *classes.*

A consistent lexicon is a great boon to programmers who must use your classes, even if it may seem like a pain while developing the classes. If you have to, you can write it into a wiki page or a document, but then it must be maintained. Don't create documents lightly.

Don't Pun

Try to never use the same word for two purposes.

After implementing the previous rule, you may have many classes with an `add` method. As long as the parameter lists are semantically equal and the desired result is the same, all is well.

However one might decide to use the word `add` for "consistency" when they are not in fact *adding* in the same sense. Perhaps in many classes "add" will create a new value by adding or concatenating two existing value. It might seem consistent to call your new method `add` even though it really only puts its single parameter into a collection. In this case, the semantics are different, and so there should be two different words used (perhaps `add` and either `insert` or `append`). Using the same term for two different ideas is essentially a *pun*.

However surprising it may seem to say so, you want your readers to afford some lazy reading and assumptions. You want your code to be a quick skim, not an intense study. You want to use the popular paperback model whereby the author is responsible for making himself clear and not the academic model where it is the scholar's job to dig the meaning out of the paper.

Again, you may need to write up an official project glossary if you start to have problems keeping the set of names and concepts clear. You might need to consider using more meaningful distinctions.

Use Rich Name Sources

Use Solution Domain Names

Go ahead, use computer science (CS) terms, algorithm names, pattern names, math terms, etc. It may seem a heretical notion, but you don't want your developers having to run back and forth to the customer asking what every name means if they already know the concept by a different name.

We're talking about code here, and you're more likely to have your code maintained by an informed programmer than by a domain expert with no programming background. End users of a system very seldom read the code, but the maintainers have to.

Solution domain names are appropriate only if you are working at a low-level where the solution domain terms completely describe the work you are doing. For work at a higher level of abstraction, you should Use Problem Domain Names (below).

Use Problem Domain Names

When the solution domain names do not reasonably describe the business effect you are trying to produce, you must use problem domain names for clarity. Consider these two examples:

```
policy.covers(vehicle)

policy.list.find(pair.first) != policy.list.end()
```

While the second is more verbose, it is certainly not more clear. The operation of determining that a vehicle is covered by a policy belongs in the problem domain, not the solution domain. Therefore, we must use problem domain naming. Rather than struggle with more meaningful names for the terms of the 'find' operation, we create clarity by renaming the entire expression.

Likewise, using pair or tuple does not help explain this high-level operation. Instead, it is better that problem-domain entities and terms such as vehicle and covers are exposed instead of solution-domain details.

Inside the method 'policy.covers', it is okay that we use solution domain terms, because these are details of implementation which are not of business-level interest.

```
bool covers(Vehicle vehicle) {
   return carlist.find(vehicle.id) !=  carlist.end();
}
```

In the chapter on writing clean functions, you will see further discussion on writing at an appropriate level of abstraction.

Make Context Meaningful

Add Meaningful Context

There are few names which are meaningful in and of themselves. Most, however are not. Instead, you need to place names in context for your reader by enclosing them in well-named classes, functions, or namespaces. When all else fails, then prefixing the name may be necessary as a last resort.

If you have a number of variables with the same prefix (address_firstName, address_lastname, address_Street), it can be a pretty clear clue that you need to create a class for them to live in. As a short-term measure and a means to an end, prefixing might not be intolerable. After all, every road out of Rome is also a road into Rome.

The term `tree' needs some disambiguation if it appears in a forestry application. You may have syntax trees, red-black or b-trees, and also elms, oaks, and pines. The word `tree' is a good word, and is not to be avoided, but it must be placed in context every place it is used.

If you review a program or enter into a conversation where the word "tree" could mean either, and you aren't sure, then the author (speaker) will have to clarify. Sadly, source won't explain itself to us, so it must be made clear.

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? Otherwise, the regular developer will have to learn to ignore the GSD at the front.

Likewise, say you invented a `Mailing Address' 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? 10 of 17 characters are redundant or irrelevant

Shorter names are generally better than longer ones, if they are clear. Add no more context to a name than is necessary.

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 I need to differentiate between MAC addresses, port addresses, and web addresses, I might consider PostalAddress, MAC, and URI. The resulting names are *more* precise, and isn't precision 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. This is a teaching issue, rather than a technical, business, or management issue.

People are also afraid of renaming things, for fear that some other developers will object. I have been working with people who do not share that fear, and I find that I am actually grateful when names change (for the better). Most of the time, I don't really memorize the names of classes and methods like I thought I did. I use the modern tools to deal with details like that, and I focus on whether my code reads like paragraphs and sentences, or at least like tables and data structure (a sentence isn't always the best way to display data). You will probably end up surprising someone when you rename, just like you might with any other code *improvement*, don't let it stop you in your tracks.

Follow some of these rules, and see if you don't improve the readability of your code. If you are maintaining someone else's code, use refactoring tools to help resolve these problems. It will pay off in the short term, and continue to pay in the long run.