Meaningful2 Names

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.

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

to

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

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.

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.

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.

EX:

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

Make Meaningful Distinctions:

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
EX:

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

you have made the names different without making them mean anything different.
make Distinguished 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
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.

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.
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
EX:

```
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;
}
```

Avoid Encodings:

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”

1. **Hungarian Notation:**
   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 compiler did not check types in those days, so the programmers needed a crutch to help them remember the types.
   nowadays HN and other forms of type encoding are simply impediments.

2. **Member Prefixes:**
   You also don’t need to prefix member variables with m_ anymore. Your classes and functions should be small enough that you don’t need them. And you should be using an editing environment that highlights or colorizes members to make them distinct.

Interfaces and Implementations:

say you are building an ABSTRACT FACTORY for the creation of shapes. This factory will be an interface and will be implemented by a concrete class. What should you name them? IShapeFactory and ShapeFactory? I prefer to leave interfaces unadorned.

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. One difference between a smart programmer and a professional programmer is that the professional understands that clarity is king. Professionals use their powers for good and write code that others can understand.

Class Names:

Classes and objects should have noun or noun phrase names like Customer, WikiPage, Account, and AddressParser. Avoid words like Manager, Processor, Data, or Info in the name of a class. A class name should not be a verb.

Method Names:

Methods should have verb or verb phrase names like postPayment , deletePage, or save.
Accessors, mutators, and predicates should be named for their value and prefixed with get, set, and is according to the javabean standard.

Don’t Be Cute:

Will 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. Choose clarity over entertainment value.

Pick One Word per Concept:

Pick one word for one abstract concept and stick with it. For instance, it's confusing to have fetch , retrieve, and get as equivalent methods of 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.

Don't Pun:

Avoid using the same word for two purposes. Using the same term for two different ideas is essentially a pun.

Our goal, as authors, is to make our code as easy as possible to understand. We want our code to be a quick skim, not an intense study. We 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.

Use Solution Domain Names:

Remember that the people who read your code will be programmers. So go ahead and use computer science (CS) terms, algorithm names, pattern names, math terms, and so forth.

Use Problem Domain Names:

When there is no "programmer-eese" for what you're doing, use the name from the problem domain. At least the programmer who maintains your code can ask a domain expert what it means.

Add Meaningful Context:

There are a few names which 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-namedclasses, functions, or namespaces. When all else fails, then prefixing the name may be necessary as a last resort.

Don't Add Gratuitous Context:

Shorter names are generally better than longer ones, so long as 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, 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. This is a teaching issue rather than a technical, business, or management issue. As a result many people in this field don't learn to do it very well.

Follow some of these rules and see whether 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.