Entity Naming in Code, Files, and Everywhere

Last update 26 Aug 2016 [MLS]

[NOTE: As with any compendium of public knowledge this document is under constant revision. Don't be surprised to find something changed from the last time you read it. And don't be hesitant about making improvements as you come across better ideas.]

Entity names are important in conveying the use and context of a thing, whether it is a file, class, function, variable, list, or any other component of a product. Pronounceable, easy to remember names are important to the continuous communication that must take place between all development team members during the life of the project and beyond. When consistent entity naming is applied across the entire project regardless of entity type then finding related bits and pieces becomes easy and less prone to error. This further reduces errors in maintenance, porting, and feature enhancement.

Whether the project is a new product or an extension of an existing product these guidelines must apply. Names are meaningless to a computer, but they are everything to the people required to do the work. Therefore, retrofit meaningful names into legacy code so that this code becomes more readable and maintainable.

Be Clear, Precise, and Concise

Clarity is the ultimate goal of all entity naming. If the name does not convey the purpose of the entity then all other considerations are moot.

Precision is the second goal. Accuracy of description helps to avoid misunderstandings and obviates much discussion about meaning.

Conciseness is the last step. Entities must be described using the minimum of words, but just like poetry the complete name must cause each word used to connote significance beyond its immediate meaning.

General Rules

Create a glossary of terms for the product (see "Where Software Comes From"). Involve all those who have an investment in what terms mean and how they are going to be used. Compile a list of standard entity names for products and product groups. Use systematic rules to determine how names are applied. Once you have a list of terms, then you can start to name entities in a reasonable and predictive manner.

Team members shouldn't have to mentally translate entity names into other names they already know. This problem generally arises from a choice to use arbitrary names that refer to neither the problem nor solution domain.

Choose names that all team members will remember. Also, use names that will remind team members why those names were chosen. What is really being named? Use words that describe that. Find a name or phrase that accurately projects a mental image of the entity without conveying more context than necessary.

Names carry connotation as well as identity, so words that project a larger realm should be used for entities that have a matching impact on the system. Use vast words for vast ideas, humble words for humble ideas. Names should imply scope.

Don't use the same word for multiple purposes. Distinguish between similar things with words that describe their differences or uniqueness. Fully understand the nature of the things being named so that you can use names based on their deeper attributes, not just what comes immediately to mind. However, don't use names that are awkward and cumbersome just because they are different from each other.

Use names that describe entities uniquely within a context. Names like Buffer, Output, and Enable are perfectly valid as long as they only apply to one particular context. For a shared context use names that identify the entities uniquely [InputBuffer, OutputBuffer, SafetyEnable, MainEnable]. Each entity needs to be identified in such a way that their distinction is obvious just by the name.

Use full words, initialized, and concatenated, aka UpperCamelCase. By using the full words that make up the name there is no chance for later errors in abbreviations. So, use InputBuffer as opposed to InpBuf or InBfr or InpBfr; use KeyboardEncoder versus KeybdEnc or KeyEncdr or KeybrdEncodr. And just as any proper name is capitalized, variable names should begin with a capital.

Abbreviations are not expressly forbidden; however, in order to be acceptable, abbreviations must be clear and unambiguous [MainRAMTest, USBPort1].

Manifest Constant (Symbol) Naming

[NOTE: This portion is subject to change in local convention.]

Manifest constants are specific values expressed as a symbol. These names are always nouns or noun phrases composed of full words, all caps, joined by underscores. Note that leading underscores are not allowed in C and C++ code by ANSI convention. Symbolic values should be declared like this:

```
#define DAYS_PER_WEEK 7
```

Variable Naming

Variables are nouns. Variables represent quantities, states, conditions, and other records. They sometimes model the real world or stand in for remote system elements. These things are all supposed to be essentially tangible quanta of the system, so verb names do not make sense for variables.

Describe usage in the name. Instead of using a very terse name that must be explained in a comment, like so:

```
int Days;  // elapsed days since last change
```

use names that will explain the variable's use. Otherwise, the developer will need to mentally "look up" the use of the variable every time it is mentioned in the code. With a self-explanatory name there is no need to perform this on-the-fly translation effort, as so:

```
int ElapsedDaysSinceModification;
int DaysSinceModification;
int FileAgeInDays;
```

The simple act of using a better name instead of a comment can improve productivity.

Don't use meaningless "i" names for loop counters, transitory variables, or anything else. Use names that indicate what is being counted or swapped. This discipline will pay off during searches and maintenance.

Don't obscure the meaning of a name by using misleading terms. Don't use List, Table, or Array as part of the name when those terms don't accurately describe the entity. Don't refer to a grouping of filters as a FilterList unless it's actually a list. The word "list" means something specific to developers. If the container holding the filters is not actually a list, it may lead to false conclusions. Describe the container accurately.

Make meaningful distinctions. Don't use arbitrarily different names just to have multiple entities. Different alone is insufficient. If names must be different, then they should also mean something different.

Number-series naming (A1, A2,..An) is the complete opposite of informational naming. Number series names provide no clue to the use of the entity. How is A1 different from A2? How does their use differ? If entities are different then their use must be different; their names must indicate this difference in context.

If, however, an entity consists of a group of like elements whose only difference is position or sequence (i.e., port pins, unassigned outputs, etc.) then such naming is useful and prudent.

Avoid disinformation. Avoid words whose common usage differs from the intended meaning. For example, CPU, RAM, and USB would be poor variable names because they are well known, generic names

of hardware. Unless you are naming an entity in a highly restricted environment such as a small microcontroller, you should not use such names. However, as class names these would be all right.

Use searchable names. Magic numbers may be intrinsically obvious in context, but they are difficult to search out when used in code. If a constant is used in multiple places it must be given a search-friendly name. For example, this code is very succinct, but nearly illegible:

```
for (int j = 1; j < 34; j++) {
    S += ((t[j] * 4) / 5);
}
```

This nearly compiler-identical code seems cluttered, but reads almost like a document:

```
#define REAL_DAYS_PER_IDEAL_WEEK 4

const int WorkDaysPerWeek = 5;
const int NumberOfTasks = 34;

for (int Task = 1; Task < NumberOfTasks; Task++) {
    RealDays = EstimateWeeks[Task] * REAL_DAYS_PER_IDEAL_WEEK;
    RealWeeks = (RealDays / WorkDaysPerWeek);
    TotalWeeks += RealWeeks;
}
```

The informational-named code is verbose, but the search for a symbol is much more constrained than that for a number. Also, the intention is immediately clear from the name itself, rather than having to derive the intention from analyzing the code at every read.

Don’t add Gratuitous Context. It is a bad idea to prefix every entity within a particular context with an unchanging prefix (ValveControl, ValveStatus, ValveFlow, ValveShutoff, yada yada yada) or suffix. (Note that Linux code and much Open Source is chock full of this usage.) This is as bad as Hungarian Notation and equates to noise in the name. Shorter names are generally better than longer ones, if they are clear. Add no more context to a name than is necessary. The larger context should suffice to distinguish the immediate entities from other entities in the system which have identical names.

Eliminate noise. Names like Product and ProductInfo and ProductData are different only in semantics, not in meaning. Info and Data are indistinct noise words like “a”, “an” and “the”.

Don’t use names which vary only slightly. How long does it take to spot the subtle difference between a ControllerForHandlingOfWater in one place and a ControllerForHeatingOfWater elsewhere? The names are very similar. In this case the invariant parts of the names are noise that mask the true intent of the name.

Function (Method) Naming

Functions are actions and therefore should be given verb names. As with variables, eliminate noise words, redundant words, and invariant words from function names. NameString is not better than Name. Name already implies a string, not a floating point number or a table. Terrible example:

```
GetSomething();
GetSomethings();
GetSomethingInfo();
```

These names give no indication of what is different between these functions, and therefore no clues as to their use. So don’t use a blanket prefix or postfix (invariant) that adds no content to the name. Also, don’t use underscores or dashes between words. *(NOTE: In this document convention manifest constants are exempt from this restriction.)*

Booleans

Booleans are used for logic level decisions. These decisions are based on a single state, condition, or result of a test. Use names that describe the single state being determined [IsEnabled(), IsReady(), IsError()]. This greatly simplifies the code and makes it read like straight text, as such:

```
If (IsFuelEnabled()  &&  IsIgnitionReady()) {
    StartEngine();
}
```

Singletons

In the case of singletons the use and the domain are the same; choose names that reflect the exactly one use of this singleton. [Keyboard::Strobe(), Keyboard.KeyCode, Keyboard::SetEnable(State)]

Structure (Class) Naming vs Instance Naming

Classes and instances should have noun or noun phrase names. Nouns phrases also refer to attributes, conditions, and other storage of a class. Verbs phrases refer to methods of acting on the storage and changing the behavior of a class.

Determine what are states [conditions] and behaviors [operations] for any old generic instance of a class. Name the class entities after these generics. However, if a class has multiple entities that may have the same state or operation, use sense in determining if it is better to use discrete names or create subclasses (pseudocode example):

```
Class House {
    ThisThing Door;
    ThatThing Window;
    Void DoorOpen();
    Void DoorClose();
    Void WindowOpen();
    Void WindowClose();
}
```

or

```
Class House {
    Class Door {
        Void Open();
        Void Close();
    }
    Class Window {
        Void Open();
        Void Close();
    }
}
```

Both of these examples are perfectly valid constructs, but each conveys meaning a different way. Only thoughtful consideration will tell you which approach best fits your application.

While Address is a fine name for a class, ‘AccountAddress’ and ‘CustomerAddress’ are poor names for classes. These names are better suited as instances of the class Address. You might differentiate between MAC addresses, street addresses, and web addresses with the more concise instance names MAC, SnailMail, and URI.

Instances are named for a particular usage. Don’t use the class name as part of the instance name unless it is unavoidable. If you find yourself having to name entities with the class name as part of the instance name then you might have a design problem. Possibly the class names have been chosen badly. Possibly the context does not allow for a distinction between multiple instances of a class or between multiple classes. A rethink is called for in any case.

Wrapup

Entity naming is an art. As such, some team members will be more naturally skillful at this practice than others, but all team members can develop their skills with concerted effort. Although naming is like poetry in that conciseness is a talent, practice will improve anyone’s skill at naming entities.

Team members should not be afraid to change the names of entities. However, everyone must be circumspect with regard to the scope of name changes. At the lowest level a code developer can change the name of a variable at will, and the IDE will track the use of that name in the code. For the names of system concepts there needs to be a much broader agreement on the need for a change before that entity can be renamed. The scope of an entity's name should indicate the effort involved in changing it. That is why the naming of high level concepts should receive so much attention during the initial phases of the project.

Name refining will need to come to an end eventually. The project team leaders will need to set a cutoff point after which the names for certain entities will not change without serious cause. Even though these names will not be perfect (and never will be) they will be useful.

As always, it is wise for the project development notes to contain a record of why particular names were chosen and why others were rejected. This information is valuable in the long term by providing both the roadmap to success on the project as well as "The Road Not Traveled" for hindsight. All of this knowledge can then be applied to the next project.