Naming guidelines for professional programmers

Abstract

Programmers generally acknowledge the difficulty of naming things, whatever their experience level and wherever they work, but relatively few use explicit naming guidelines. Various authors have published different kinds of identifier naming guidelines, but these guidelines do little to make naming easier, in practice, due to their formulation. Meanwhile, professional programmers follow diverse conventions and disagree about key aspects of naming, such as acceptable name lengths. Tnese teams lack consistent standards.

Few teams write their own coding standards, let alone naming guidelines, but many teams use code review and pair programming to maintain code quality. We believe that these teams could use third-party naming guidelines to inform these reviews, and improve their coding style.

This paper examines various sources of naming guidelines, and reflects on them, in the context of the first author's twenty years' experience as a professional programmer. This paper then presents a consolidated set of naming guidelines that professional programmers can apply to the code they write.

Why naming matters

Several researchers have explored the importance of naming. For example, Deißenbock and Pizka conclude that identifier names are crucial to program comprehension:

> Research on the cognitive processes of language and text understanding show that it is the semantics inherent to words that determine the comprehension process [14]

Other authors agree; Caprile and Tonella [15] state that identifiers provide important information about program entities, because they give the programmer an initial idea of these entities' roles within the whole program. Deißenbock and Pizka [14] not only present their opinion on naming, they also perform measurements. They found that in the Eclipse code base, which consists of about 2 MLoC, 33 per cent of the tokens and 72 per cent of characters are devoted to identifiers.

Better identifier names have been known to correlate with improved program comprehension. For example, [17] reports on a study performed with over 100 programmers, who had to describe functions and rate their confidence in doing so. Their results show that using full word identifiers leads to better code comprehension than using single-letter identifiers, measured by both description rating and by confidence in understanding. However, they also found that in many cases there is no difference between words and abbreviations. Interestingly, this study also found that women comprehend more from abbreviations than men do.

Naming might have been found to matter for source code quality. Butler *et al.* [16] evaluated the quality of identifiers in eight large Java projects according to a number of naming style guidelines. They found that the occurrence of naming violations correlated with code issues as reported by FindBugs, a static analysis tool for Java. In particular, capitalisation errors, using non-dictionary words and using more than four words were correlated with issues.

Developers agree that naming matters. In an ethnographic study among twelve professional developers and eighteen third-year students [18], researchers found that both

students and professional developers find the use of naming guidelines important. The study also found a remarkable difference between professionals and students: professional developers pay more attention to the name of the identifiers than to source code comments. Could this be due to the fact that computer science courses tend to emphasise the importance of comments but largely neglect naming?

While developers agree that guidelines are important, in practice, we have observed that software development typically turns out to cost more and take longer than anyone expects. As Bugayenko writes [9], software development is 'a never-ending process' that will cost 'All of your money, and it won't be enough'. We see that the cost of continuous software development includes the cost of debugging, fixing and maintaining code. These activities clearly require programmers to read and understand existing code. As programmers, we can only understand code if we know what it means.

A thought experiment further illustrates that we rely on naming to understand what code means. Imagine trying to read code after someone has replaced every identifier name with an underscore followed by a random number. Although an identifier like `result` communicates relatively little intent, a name like `_42` explains even less.

Purpose of naming guidelines

In the above, we have established that naming is important, but also hard. As Karlton famously joked:

> 'There are only two hard things in Computer Science: cache invalidation and naming things.' - Phil Karlton

We think some programmers make the mistake of focusing too much on the executability of the code, rather than on the value of the code as a thing for humans to read, forgetting that other famous quote:

> 'Programs are meant to be read by humans and only incidentally for computers to execute.' - Donald Knuth

A good name helps a future reader of code to quickly understand what a value means, thus making code more readable and easier to understand.

However, programmers don't always try write code to be maintainable, and when they do they typically find it difficult to achieve. The very idea of 'maintenance' lacks a common industry definition that doesn't assume a specific (usually waterfall) software development method or software services business model. Similarly, computer science does not have a clear definition of 'maintainability', and instead focuses on proxies such as code comprehension and programmers' ability to discover and correct code errors. These related measures reduce 'maintainability' to 'readability'.

Readability requires good naming, because bad names obscure the programmer's intent. We report, above, that naming affects programmers' ability to read and understand source code. Unfortunately, programmers struggle to write readable code because they struggle to avoid using bad names. Naming guidelines aim to help programmers identify and avoid bad names, and to guide programmers towards good names.

not sure what to do with these, maybe better to leave maintenance out of it for now and focus on names? Then later in the paper in the conclusion we can revisit the broader impact. -

We see naming guidelines as a means to help programmers write more maintainable code, and to reduce the cost and difficulty of software development. Crucially, these benefits potentially apply to all software development, not just long-term maintenance of legacy systems.

Existing guidelines

In our experience, professional software developers don't use explicit naming guidelines extensively. The few written coding standards in common use, such as [6], limit their guidelines to formatting and name length, but offer little to clarify the difference between good names and bad names.

Some books for software developers include a section on naming. *Code Complete* [4] includes a 30-page chapter on *The Power of Data Names*. This includes fourteen guidelines for how to write better names, a discussion of various naming conventions, a list of eleven naming smells and a checklist that summarises these guidelines. For this chapter alone, we recommend that every professional programmer own a copy of this book, or at least study the collected guidelines we present below.

Clean Code [5] also has a whole chapter on *Meaningful Names*, which consists of eighteen guidelines. Most of these guidelines directly address the hardest part of naming: semantics. More recent programming books tend to devote fewer words to naming, perhaps because they have little to add.

Computer science research sometimes includes naming guidelines. Papers by Relf [2] and Arnaodova *et al.* [3] include collections of naming guidelines, which they evaluate in different ways. Computer scientists will likely continue publish papers that include naming guidelines, but professional programmers rarely have access to published papers, which makes them less directly useful in the software industry than books.

Importance of different guidelines to professional programmers

Professional software developers benefit more from some kinds of guidelines than from others. Guidelines like *Variable names should be short yet meaningful* [6] sound reasonable, but offer little practical help, either when choosing a name when coding or when evaluating a name during code review.

Some academic studies, such as Binkley [10], have compared the relative readability of different formatting conventions, such as camel-case (capitalised words) and snake-case (words separated by underscores). In principle, programming language designers could use this research when setting these conventions to design programming languages with a more productive developer experience, but professional programmers have little use for them.

Ken Arnold typifies the view that the responsibility for using this kind of research to choose a coding style lies with language designers rather than programmers. In his essay *Style is substance*, he argues that a programming language's specification should fix all aspects of coding style, so that compilers reject all violations:

> For almost any mature language […] coding style is an essentially solved problem, and we ought to stop worrying about it. […] the only way to get from where we are to a place where we stop worrying about style is to enforce it *as part of the language*. […]
>
> I want the owners of language standards to take this up. I want the next version of these languages to require any code that uses new features to conform to some style. [11]

He argues that programmers follow the name formatting convention that a programming language community adopts, and that these programmers have nothing to gain from this kind of research.

> For any given language, there are one or a few common coding styles. […]
>
> There is not now, nor will there ever be, a programming style whose benefit is significantly greater than any of the common styles. [11]

However, Binkley [10] concludes that not all 'common coding styles' deliver the same productivity, and that 'it becomes evident that the camel case style leads to better all around performance once a subject is trained on this style'. Either way, this remains a result for language designers to consider, not professional programmers.

Fortunately, some research has directly addressed different guidelines' usefulness. Relf, for example, concludes that:

> The identifier-naming style guidelines that proved to be the most useful to programmers required that identifier names should be composed of from two to four natural language words or project accepted acronyms; should not be composed only of abstract words; should not contain plural words; and should conform to the project naming conventions. [8]

Professional programmers can apply guidelines more readily than they can access and read the original scientific research, especially when stated this clearly. We therefore aim to

present a larger collection of naming guidelines from a number of sources in a form that makes them accessible to professional programmers.

Guideline styles

People who write naming guidelines phrase their guidelines in different ways. Some authors write prescriptive instructions, e.g. *Use intention-revealing names* [5], while some phrase them as code smells or naming problems, e.g. *Meaningless names* [1].

We examined a number of written naming guidelines [1-7]. On guideline styles, we conclude that a single naming guideline typically contains one or more of the following.

- Prescriptive instruction
- Naming smell name
- Correcting refactoring name
- Example guideline violation
- Example name that follows the guideline
- Explanation of why the guideline matters or how it works

Naming smells are 'code smells' that come from bad names. A code smell indicates where you can improve your code, and often points to some deeper problem. A particular code smell often has a corresponding refactoring that removes that particular smell, improving the code. Naming smells appear in many forms, but all have the same refactoring: *Rename*.

Needless to say, programmers find consistently-written guidelines easier to understand and apply. As well as consistency, multiple explanations help programmers apply a guideline in different scenarios. Naming smells help programmers identify violations during code review, while prescriptive instructions are easier to follow while writing code. Examples serve to explain both smells and instructions, whose abstract nature can make them hard to understand.

The remainder of this paper presents and discusses specific guidelines.

Syntax guidelines

Syntax guidelines address how identifiers are constructed from words and formatted. These guidelines are not concerned with which words names use, except for the guideline to use words in the first place.

Use naming conventions

Guideline. Follow the programming language's conventions for names. Programming languages usually have some conventions for how to write identifier names, or at least their specifications or communities do. Java programmers, for example, follow Sun Microsystems' original guidelines [6] for how to use upper and lower-case, nouns and verbs, in the names of classes, interfaces, methods, variables and constants.

Refactoring. Apply standard case with rigorous consistency, and use language-specific code inspection tools to enforce it.

Example violations. `appleCOUNT`, `apple_count` (when camel-case is standard).

References: [2], [6]

Replace numeric suffixes

Guideline. Don't add numbers to multiple identifiers with the same base name. If you already have an `employee` variable, then a name like `employee2` has as little meaning as `another_employee`.

Refactoring. Replace the numbers with additional words that describe the difference between multiple identifiers that might otherwise have the same name.

Example violations. `employee2`

References: [1], [2], [4]

Use dictionary words

Guideline. Only use correctly-spelled dictionary words and abbreviations that appear in a dictionary. Make exceptions for `id` and documented domain-specific language/ abbreviations. Spelling mistakes can render names ambiguous, and result in confusing inconsistency. Abbreviations introduce a different kind of ambiguity that the original programmer does not see because they know which word the abbreviation stands for, even if multiple words have that same abbreviation.

Refactoring. Write words out in full and define abbreviations for the bounded context. Use tools that identify spelling errors in identifier names.

Example violations. `acc`, `pos`, `char`, `mod`, `auth`, `appCnt`

References: [1], [4], [5]

Expand single-letter names

Guideline. Don't make exceptions to using dictionary words for single-letter names; use searchable names. Single-letter names, when used as abbreviations, introduce the maximum possible ambiguity. They end up being used with specific meanings, usually by unwritten convention, which makes the code harder to read for programmers when they first encounter the convention or who have to switch between conflicting conventions in different contexts.

One study of over 100 programmers that compared comprehension for single letters, 'well-formed' common abbreviations and full words supports this guideline:

> The results show that full-word identifiers lead to the best comprehension; however, in many cases, there is no statistical difference between using full words and abbreviations. [12]

Refactoring. Use dictionary words.

Example violations. `i`, `j`, `k`, `l`, `m`, `n`, `t`, `x`, `y`, `z`

References: [1], [2], [4], [5]

Articulate symbolic names

Guideline. Don't use ASCII art symbols instead of words, in programming languages that support it. Make very limited exceptions for documented domain-specific symbols, e.g. `+` in arithmetic. Ironically, programmers who encounter symbolic names in third-party libraries may invent their own names, but choose names based on what the symbol looks like, rather than what it means.

Refactoring. Use dictionary words.

Example violations. `>=>`, `<*>` - valid function identifiers in Scala, for example, colloquially named *fish* and *space ship*.

References: [1]

Name constant values

Guideline. Name what the constant represents, rather than its constant value. Don't construct numeric constant names from numbers' names.

Refactoring. Extract constant, for the *Magic number* code smell. Replace number names with either domain-specific names, such as `pi`, or a name that describes the concept that the number represents, such as `boiling_point`.

Example violations. `3.142591`, `ONE_HUNDRED`

References: [2], [4]

Only use one underscore at a time

Guideline. Don't use more than one consecutive underscore. Multiple underscores usually appear as a single line, which makes it hard to count them.

Refactoring. Replace with a single underscore. Use tools that warn when names contain multiple underscores.

Example violations. `APPLE__COUNT`

References: [2]

Only use underscores between words

Guideline. Don't use underscores as prefixes or suffixes. Underscores lack visual prominence, which makes them good word separators, but easy to misread before or after a word.

Refactoring. Trim underscores. Use tools that warn when names do not start with a letter.

Example violations. `_APPLE_COUNT`, `APPLE_COUNT_`

References: [2]

Limit name character length

Guideline. Keep name length within a twenty character maximum.

The results of one experiment involving 158 'programmers of varying degrees of experience':

> […] reinforce past proposals advocating the use of limited, consistent, and regular vocabulary in identifier names. In particular, good naming limits individual name length and reduces the need for specialized vocabulary. [13]

Refactoring. Simplify name, Extract variable.

Example violations. `ForeignDomesticAppleCount`

References: [2]

Limit name word count

Guideline. Keep name length within a four word maximum, and avoid gratuitous context. Limit names to the number of words that people can read at a glance. Don't unnecessarily use the same prefix, such as the software system's name, for all names.

Refactoring. Simplify name, Extract variable.

Example violations. `NewRedAppleSizeType`, `MyAppSizeType`

References: [2], [4], [5], [8], [16]

Qualify values with suffixes

Guideline. Use a suffix to describe what kind of value constant and variable values represent. Suffixes such as `minimum`, `count` and `average` relate a collection of values to a single derived value. Using a suffix, rather than a prefix, for the qualifier naturally links the name to other similar names.

Refactoring. Move the qualification to the end.

Example violations. `MINIMUM_APPLE_COUNT` (replace with `APPLE_COUNT_MINIMUM`).

References: [2], [4]

Make names unique

Guideline. Don't overwrite a name with a duplicate name in the same scope. In Java, for example, a local variable 'shadows' a class field that has the same name. Adopt a convention that prevents ambiguity in which name the programmer intended to refer to.

Refactoring. Add words to one of the names clarify the difference between contexts.

References: [2]

Vocabulary guidelines

Vocabulary guidelines address word choice, with the rationale that using the *right* word matters.

Describe meaning

Guideline. Use a descriptive name whose meaning describes a recognisable concept, with enough context. Avoid placeholder names that deliberately mean nothing more than `a_variable`.

Refactoring. Describe what the identifier represents.

Example violations. `foo`, `blah`, `flag`, `temp`

References: [1], [4], [5]

Be precise

Guideline. Identify a specific kind of information and its purpose. Imprecise words might apply equally to multiple identifiers, and therefore fail to distinguish them.

Refactoring. Replace vague words with more specific words that would only be correct for this name.

Example violations. `data`, `object`

References: [1]

Choose concrete words

Guideline. Use words that have a single clear meaning. Like imprecise words, abstract words might apply equally to multiple identifiers.

Refactoring. Replace with more specific words that narrow down the concept they refer to.

Example violations. `Manager` suffix, `get` prefix, `doIt`

References: [1], [2]

Use standard language

Guideline. Avoid being cute or funny when it results in a name that requires shared culture or more effort to understand. Like deliberately meaningless names, cute and funny names require the reader to understand some implicit context. While humour often relies on indirect references and ambiguity, these qualities do not improve code readability.

Refactoring. Replace indirect references and colloquial language with the corresponding explicit and standard language.

Example violations. `whack` instead of kill.

References: [5]

Use a large vocabulary

Guideline. Use a richer single word instead of multiple words that describe a well-known concept. Use the word that most accurately refers to the concept the identifier refers to.

Refactoring. Replace multiple words that describe a concept when 'there's a word for that'.

Example violations. `CompanyPerson` (replace with `Employee`).

References: [1]

Use problem domain terms

Guideline. Use the correct term in the problem domain's ubiquitous language, and only one term for each concept, within each bounded context. Consistently use the correct domain language terms that subject-matter experts use.

Refactoring. Rename identifiers to use the correct terminology.

Example violations. `Order` when you mean `Shipment`, in a supply-chain context, where it means something different.

References: [1], [4], [5]

Make names differ by more than one or two letters

Guideline. Don't use a name that barely differs from an existing name. Avoid words that you will probably mix up when reading the code.

Refactoring. Make the difference more explicit by adding or changing words.

Example violations. `appleCount` vs `appleCounts`

References: [2], [4], [5]

Make names differ by more than word order

Guideline. Don't use a name that only differs from an existing name in word order. Don't use two names that both combine the same set of words.

Refactoring. Make the difference more explicit by using different words rather than just different word order to communicate different meanings.

Example violations. `appleCount` vs `countApple`

References: [2]

Make names differ in meaning

Guideline. Don't use names that have the same meaning as each other. Avoid names that only differ by changing words for their synonyms.

Refactoring. Rename both variables with more explicit names.

Example violations. `input`/`inputValue`, `recordCount`/`numberOfRecords`

References: [4]

Make names differ phonetically

Guideline. Don't use names that sound the same when spoken. Aim to write code that another programmer could write down correctly if you read it out loud. Even though they don't transcribe code like that, as a rule, they often talk about code.

Refactoring. Replace a homophone with a synonym.

Example violations. `wrap`/`rap`

References: [4]

Data type guidelines

Data type guidelines extend vocabulary guidelines by addressing data type names in identifier names. Some of these guidelines only apply to languages whose type system allows code to explicitly identify data types, separately from identifier names. Code in other programming languages cannot always avoid the need to indicate types.

Omit type information

Guideline. Don't use prefixes or suffixes that encode the data type. Avoid Hungarian notation and its remnants. Don't prefix Boolean typed values and functions with `is`.

Refactoring. Remove words that duplicate the data type, either literally or indirectly.

Example violations. `isValid`, `dateCreated`, `iAppleCount` (replace with `valid`, `created`, `appleCount`).

References: [1], [2], [5]

Use singular names for values

Guideline. Don't pluralise names for single values.

Refactoring. Replace the plural with the singular form.

Example violations. `appleCounts` (replace with `appleCount`).

References: [2], [3]

Use plural names for collections

Guideline. Pluralise names for collection values, such as lists. Technically, this contradicts the guideline to avoid encoding type information in names, but English grammar requires it to make it possible to read the code normally, or out loud.

Refactoring. Use the plural form.

Example violations. `remainingApple` for a set of apples (replace with `remainingApples`).

References: [3]

Prefer collective nouns for collections

Guideline. If a collection's type has a collective noun, in the name's context, use it instead of a plural.

Refactoring. Use the collective noun, when possible, instead of a regular plural form.

Example violations. `appointments` (replace with `calendar`), `pickedApples` (replace with `harvest`).

References: [1]

Use opposites precisely

Guideline. Consistently use opposites in standard pairs with naming conventions. Typical pairs include add/remove, begin/end, create/destroy, destination/source, first/last, get/release, increment/decrement, insert/delete, lock/unlock, minimum/maximum, next/previous, old/new, old/new, open/close, put/get, show/hide, source/destination, start/stop, target/source, and up/down.

Refactoring. Use the correct opposite, and use it consistently.

Example violations. `first`/`end`

References: [4]

Use Boolean variable names that imply *true* or *false*

Guideline. Use names like `done` or `found` that describe Boolean values. Use conventional Boolean names, possibly from a code conventions list.

Refactoring. Replace Boolean names with names in the correct grammatical form.

Example violations. `status` for e.g. `started`

References: [4]

Use positive Boolean names

Guideline. Don't use negation in Boolean names. Don't use names that require a prefix like `not` that inverts the variable's truth value.

Refactoring. Invert the meaning and remove the prefix.

Example violations. `notSuccessful`

References: [4]

Class name guidelines

Class name guidelines specifically address names for classes in object-oriented programming languages.

Use a noun-phrase name

Guideline. Name a class with a noun phrase so you can use the class name to complete the phrase *This class' constructor returns a new…*. Follow object-oriented programming's grammatical conventions.

Refactoring. Add the missing noun, remembering to Choose concrete words.

Example violations. `Calculate` (replace with `DiscountRule`, for example).

References: [5], [6]

Use a name that allows all possible states

Guideline. Don't use class names that assume a particular state. If a class models something that can have multiple states, then avoid a name that would be inconsistent with the state that results from calling a method that changes that state.

Refactoring. Make the class name less specific to accommodate all possible states.

Example violations. A `disable` method that returns a `ControlEnableState` (rename class to `ControlState`).

References: [3]

Choose a name consistent with possible values

Guideline. Don't use a name that appears to contradict certain possible values. Some types aggregate multiple values of the same type, such as a line that has a `start` and an `end`, so use a name that applies equally to both values, such as `Extremity`, rather than naming the type after just one possible value, such as `Start`.

Refactoring. Make class name inclusive.

Example violations. start field has type MAssociationEnd (rename class to MAssociationExtremity).

References: [3]

Method name guidelines

Method name guidelines specifically address names for methods in object-oriented programming languages. Several of these guidelines apply to Java in particular, due to the bad habits the JavaBeans Specification [6] encouraged.

Use a verb-phrase name

Guideline. Make the method name an active verb phrase, except for accessor methods. As with the guideline to use noun phrases to name class, follow object-oriented programming's grammatical conventions. Some coding styles omit the verb from accessor methods, changing `Parcel.getWeight()` to `Parcel.weight()`. Another common style is to omit the verb from conversion methods, changing `Discount.convertToPercentage()` to `Discount.asPercentage()`.

Refactoring. Add the missing verb, remembering to Choose concrete words.

Example violations. `calculation()`

References: [5], [6]

Don't use `get`, `is` or `has` prefixes for methods with side-effects

Guideline. Use a verb phrase that suggests the side-effect, if there is one. Verbs like `create` and `convert` suggest a side-effect, while others suggest idempotence.

Refactoring. Replace 'get' with another verb.

Example violations. `getImageData` method that constructs a new object.

References: [3]

Only use `get`, `is` and `has` prefixes for methods that only perform field access

Guideline. Only use the conventional accessor method name prefixes for accessor methods that directly return a field value. In Java, the JavaBeans specification [7] requires these prefixes for certain methods. When some methods require a certain prefix, don't use the same prefixes for methods that do not require them.

Refactoring. Replace 'get' with another verb.

Example violations. `getScore` that performs calculation or accesses external data.

Only use `get` prefix for field accessors that return a value

Guideline. Don't use the `get` field accessor method name prefix for methods that don't return a value.

Refactoring. Replace 'get' with a verb that describes the side-effect.

Example violations. `getMethodBodies` populates the method bodies but doesn't return them.

References: [3]

Only use `is` and `has` prefixes for Boolean field accessors

Guideline. Don't use the conventional Boolean accessor method name prefixes for methods that don't return a Boolean value.

Refactoring. Replace prefix with `get` or remove the prefix altogether.

Example violations. `isValid` returns an `int` value.

References: [3]

Only use `set` prefix for field accessors that don't return a value

Guideline. Don't use the `set` field accessor method name prefix for methods that return a value.

Refactoring. Replace 'set' with another verb, or remove it in a 'fluent API' that chains method calls.

Example violations. `setBreadth` creates and returns a new object, or updates and returns `this` (fluent API).

References: [3]

Only use validation verbs for methods that provide the result

Guideline. Only use verbs like `validate`, `check` or `ensure` to name methods that either result or throw an exception when validation fails.

Refactoring. Return result.

Example violations. `validateSnaps` and `checkCurrentState` that return `void`.

References: [3]

Only use transformation verbs for methods that return a transformed value

Guideline. Only use verbs that suggest transformation, like `convert`, for methods that return the result.

Refactoring. Return result, or change the verb to indicate what the method transforms.

Example violations. `javaToNative` with return type `void`.

References: [3]

Rejected guidelines

To illustrate disagreement among programmers about which guidelines to use, the following paragraphs quote naming guidelines together with our rationale for why we do not 'strongly accept' them.

Use long names for long scopes

> When you give a variable a short name like `i`, the length itself says something about the variable - namely that the variable is a scratch value with a limited scope of operation. [4]
>
> The length of a name should be related to the length of the scope. You can use very short variable names for tiny scopes, but for big scopes you should use longer names. [5]

Although this guideline sounds reasonable and enjoys wide popularity among programmers, it contradicts other guidelines and encourages bad naming. This guideline essentially recommends that you *encode a variable's scope in its name length*. This contradicts the same authors' advice to avoid encodings in names. Even if that were a good idea, this would be hard to do consistently enough that someone reading the code could reliably infer a name's scope from its length. And even if that would be possible, this would impose an unrealistic maintenance burden to rename variables when their scope changes.

This guideline's popularity does not arise because programmers can overcome these challenges, but because they read it as an excuse to use bad names. Indeed, when a name

has a very small scope, a bad name does less damage. That does not mean that we should have a guideline to deliberately use bad names in that case. At best, programmers are using the actual guideline to *prioritise naming effort and use bad names when you can get away with it*. Even if this were a good idea, the person writing the code typically cannot judge what they can get away with.

Correctly deciding whether a scope is small enough for a variable to only need a single-letter name is harder than always choosing a better name. Ignoring this would-be guideline leads to more maintainable code.

Short Identifier Name

> [avoid] an identifier name shorter than eight characters, excluding: `i`, `j`, `k`, `l`, `m`, `n`, `t`, `x`, `y` or `z` [2]

> Variable names should be short yet meaningful. The choice of a variable name should be mnemonic - that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary “throwaway” variables. Common names for temporary variables are `i`, `j`, `k`, `m`, and `n` for integers; `c`, `d`, and `e` for characters. [6]

We don’t use this guideline, in practice, because we’re more concerned about avoiding abbreviations (see Use dictionary words) than that names should not be too short. In fact, we’d partly accept these guidelines, were it not for their exceptions for single-letter names, which we consider the worst kind of abbreviation.

Long Identifier Name

> [avoid] an identifier name longer than twenty characters [2]

We only partly accept this guideline, because we prefer names to be as long as necessary. However, we would also consider a name longer than twenty characters to be suspiciously long, and look for either a simpler name or extracting an intermediate declaration, which sometimes simplifies the thing with the long name.

Number of Words

> an identifier should consist of two, three or four words [2]

As with the previous guidelines, we don’t use this because we prefer to let the other guidelines determine length. However, in his 2007 doctoral thesis, Relf reveals the neuroscience for limiting identifiers to four words, which suggests that may be a good idea. We don’t know what the objection to one-word names might be, especially when the correct term in a bounded context’s vocabulary (a subject domain term) might be a single word, such as a ‘shipment’ in a supply chain context.

Class/Type Qualification

> class names and type names should be qualified to identify their nature […] e.g. `Fruit` (`FruitClass` and `Fruit_Tree` are considered more readable) [2]

We reject this guideline, as did the study participants (in [2]. We would consider adding a class name `Class` suffix redundant. Many languages use a capitalisation convention for type names, and a `class` keyword for declarations. Furthermore, professional software developers tend to use tools (IDEs) that indicate which identifiers are types, or support navigation to the declaration. We have never heard of anyone systematically adopting this guideline.

Constant/Variable Qualification

> numeric range constants should be fully qualified […] e.g. `Minimum_Apple_Count` (`Apple_Count_Minimum` is considered more readable) [2]

This guideline seems reasonable, but we probably prefer grammatical English word order sometimes.

Singular Word

> identifier names should be composed of words in the singular [2]

We reject this guideline because it doesn't consider collection types, but it's easily fixed. Only use singular names for single values, and only use plural names for collections.

Similar Identifier Names

> [avoid] the appearance of two similar identifier names both in scope [2]

We don't know whether to accept this guideline, because we don't experience this as a problem in practice, and don't know how onerous we would find it.

Enumeration Identifier Definition Order

> [avoid] enumeration constants declared in non-alphabetical order [2]

We partly accept this guideline, which at least requires an order and thus prevents (apparently) random order. However, some enumerations, such as weekdays, have their own non-alphabetical *natural* order. Fortunately, we don't follow guidelines blindly.

Enumeration Identifier Qualification

> [avoid] the non-qualification of enumeration constants to identify their base type [2]

Adding an enumeration type's name to its constants' name make as little sense as adding a class' name to its instances' names (or guideline come to that). Fortunately, we've never seen this in practice.

Further research

While developers agree guidelines are important [18], they remain underused in the software industry. In our experience, professional software developers do not always agree on which guidelines to use, or even that they are worthwhile. To further good naming practices, our industry would benefit from more rigorous answers to the following questions.

1 Which naming guidelines apply universally to all kinds of code?
2 Which naming guidelines have the most positive impact on code readability and maintainability?
3 Can we usefully reduce naming guidelines to a short checklist for use in code review?
4 Can mining large collections generate 'crowd-sourced' standard vocabularies for specific domains?
5 How should software developers *write* naming guidelines?
6 How should software developers *use* naming guidelines?
7 Is it possible to measure identifier name quality or naming guideline effectiveness?
8 What can we learn from a cost-benefit analysis of naming guidelines?
9 How do naming and naming guidelines relate to software documentation?
10 Does better naming reduce the need for code comments?
11 Do pair programming and mob programming significantly improve naming quality?
12 How can an English dictionary and thesaurus help programmers choose effective names?
13 Which techniques have a positive effect on improving programmers' naming skills?
14 Which tools to support naming have programmers not yet tested in an industrial setting?
15 How does the programmer's native language affect their approach to choosing English identifier names?

Needless to say, we hope that software engineering researchers address these questions in the future.

References

1 Peter Hilton - Naming smells (2016)

2 Phillip Relf - *Achieving Software Quality through Source Code Readability* (2004)

3 Venera Arnaoudova, Massimiliano Di Penta and Giuliano Antoniol - Linguistic Antipatterns: What They Are and How Developers Perceive Them (2015)

4 Steve McConnell - *Code Complete*, Microsoft Press (1993)

5 Robert C. Martin - *Clean Code*, Prentice Hall (2009)

6 Sun Microsystems - Code Conventions for the Java TM Programming Language (20 April 1999)

7 Sun Microsystems - JavaBeans - JavaBeans™ API specification version 1.01 (1997)

8 Phillip Relf - *Source Code Readability Improvement Using Heuristic-Based Dynamic Error Reporting During Editing* - doctoral thesis (2007)

9 Yegor Bugayenko - How Much For This Software?

10 Dave Binkley, Marcia Davis, Dawn Lawrie and Christopher Morrell - *To CamelCase or Under score* (2009)

11 Ken Arnold - *The Best Software Writing I* (2005, ed. Joel Spolsky), pp 1-6, previously published online as Style is Substance (2004)

12 Dawn Lawrie, Christopher Morrell, Henry Feild and David Binkley - *Effective identifier names for comprehension and memory* (2007)

13 Dave Binkley, Dawn Lawrie, Steve Maex and Christopher Morrell - *Identifier length and limited programmer memory* (2009)

14 F. Deißenböck and M. Pizka - *Concise and consistent naming* (2005) In Proceedings of IWPC 2005.

15 B. Caprile and P. Tonella - *Restructuring program identifier names* (2000) In Proceedings of ICSM, 2000.

16 Simon Butler, Michel Wermelinger, Yijun Yu and Helen Sharp - *Relating Identifier Naming Flaws and Code Quality: An Empirical Study* (2009) In Proceedings of WCRE 2009.

17 Dawn Lawrie, Christopher Morrell, Henry Feild and David Binkley - *What's in a Name? A Study of Identifiers* (2006) In Proceedings of ICPC 2006.

18 Felice Salviulo and Giuseppe Scanniello - *Dealing with identifiers and comments in source code comprehension and maintenance: results from an ethnographically-informed study with students and professionals* (2014) In Proceedings of EASE 2014