Dealing with the Second Hardest Thing in Computer Science

Indrajeet Patil

Source code for these slides can be found on GitHub.

“There are only two hard things in Computer Science: cache invalidation and naming things.”

- Phil Karlton

The following advice on naming applies to all kinds of programming entities (variables, functions, packages, classes, etc.) and is **language-agnostic**.

Principle: Names are a form of abstraction

"[T]he best names are those that focus attention on what is most important about the underlying entity, while omitting details that are less important."

- John Ousterhout

Importance: Names are at the core of software design

If you can't find a name that provides the right abstraction for the underlying entity, it is possible that the design isn't clear.

Properties: Good names are precise and consistent

If a name is good, it is difficult to miss out on critical information about the entity or to misunderstand what it represents.

“The beginning of wisdom is to call things by their proper name.”

- Confucius

Good names are a form of documentation

How good a name is can be assessed by how detailed the accompanying comment needs to be.

E.g., the function and parameter are named poorly here, and so comments need to do all the heavy lifting:

```
1 // function to convert temperature from Fahrenheit to Celsius scale
2 // temp is the temperature in Fahrenheit
3 double unitConverter(double temp)
```

Generic names should follow conventions

Using generic names can improve code readability, but *only if* language or domain customs are followed.

Examples:

- In a nested loop, using `j` for outer and `i` for inner loop index is confusing!

```
for (let j = 0; j < arr.length; j++) {
  for (let i = 0; i < arr[j].length; i++) {
```

Alternatives to generic names

If a loop is longer than a few lines, use more meaningful loop variable names than `i`, `j`, and `k` because you will quickly lose track of what they mean.

```
# abstruse
exam_score[i][j]
```

```
# crystal clear
exam_score[school][student]
```

Names should be consistent

Consistent names **reduce cognitive burden** because if the reader encounters a name in one context, they can safely reuse that knowledge in another context.

For example, these names are inconsistent since the reader can't safely assume that the name *size* means the same thing throughout the program.

```
1  // context-1: `size` stands for number of memory bytes
2  {
3    size = sizeof(x);
4  }
5
6  // context-2: `size` stands for number of elements
7  {
8    size = strlen(a);
9  }
```

```
1  // context-1:
2  {
3    size = sizeof(x);
4  }
5
6  // context-2:
7  {
8    length = strlen(a);
9  }
```

Source code for these slides can be found on GitHub.

Unnecessary details in names should be removed...

```
# okay
convert_to_string()
fileObject
strName # Hungarian notation
```

```
# better
to_string()
file
name
```

Names should utilize the context

When naming, avoid redundant words by exploiting the context.

E.g. if you are defining a class, its methods and variables will be read in that context.

```
# okay
Router.run_router()
FileHandler.close_file()
BeerShelf.beer_count
```

```
# better
Router.run()
FileHandler.close()
BeerShelf.count
```

Names should be precise but not *too* long

How precise (and thus long) the name should be is a **subjective decision**, but keep in mind that long names can obscure the visual structure of a program.

You can typically find a middle ground between too short and too long names.

```
# not ideal - too imprecise
d

# okay - can use more precision
days

# good - middle ground
days_since_last_accident

# not ideal - unnecessarily precise
days_since_last_accident_floor_4_lab_23

...
```

Names should be difficult to misinterpret

Try your best to misinterpret candidate names and see if you succeed.

E.g., here is a GUI text editor class method to get position of a character:

```
1  std::tuple<int, int> getCharPosition(int x, int y)
```

Names should be distinguishable

Names that are too similar make great candidates for mistaken identity.

E.g. `nn` and `nnn` are easy to be confused and such confusion can lead to painful bugs.

```
// bad
let n   = x;
let nn  = x ** 2;
let nnn = x ** 3;
```

```
// good
let n       = x;
let nSquare = x ** 2;
let nCube   = x ** 3;
```

Names should be searchable

While naming, always ask yourself how easy it would be to find and update the name.

E.g., this function uses `a` and `f` parameters to represent an array and a function.

```r
# bad
arrayMap <- function(a, f) {
  ...
}
```

```r
# good
arrayMap <- function(arr, fun) {
  ...
}
```

Names should honour the conventions

The names should respect the conventions adopted in a given project, organization, programming language, domain of knowledge, etc.

For example, C++ convention is to use PascalCase for class names and lowerCamel case for variables.

```cpp
// non-conventional
class playerEntity
{
  public:
    std::string HairColor;
};
```

```cpp
// conventional
class PlayerEntity
{
  public:
    std::string hairColor;
};
```

Name Booleans with extra care

Names for Boolean variables or functions should make clear what true and false mean. This can be done using prefixes (**is**, **has**, **can**, etc.).

```
# not great
if (child) {
  if (parentSupervision) {
    watchHorrorMovie <- TRUE
  }
}
```

```
# better
if (isChild) {
  if (hasParentSupervision) {
    canWatchHorrorMovie <- TRUE
  }
}
```

Avoid implementation details in names

Names with implementation details (e.g., data structure) have high maintenance cost. When implementation changes, identifiers need to change as well.

E.g., consider variables that store data in different data structures or cloud services:

```python
# bad
bonuses_pd  # pandas DataFrame
bonuses_pl  # polars DataFrame

aws_s3_url  # AWS bucket
gcp_url     # GCP bucket
```

```python
# good
bonuses    # data structure independent

bucket_url # cloud service independent
```

Find correct abstraction level for names

Don’t select names at a lower level of abstraction just because that’s where the corresponding objects were defined.

E.g., if you are writing a function to compute difference between before and after values, the parameter names should reflect the higher-level concept.

```
# bad
function(value_before, value_after) { ... }
```

```
# good
function(value1, value2) { ... }
```

Test function names should be detailed

If unit testing in a given programming language requires writing test functions, choose names that describe the details of the test.

The test function names should effectively act as a comment.

```
# bad
def test1
def my_test
def retrieve_commands
def serialize_success
```

```
# good
def test_array
def test_multilinear_model
def all_the_saved_commands_should_be_retrieved
def should_serialize_the_formula_cache_if_required
```

Source code for these slides can be found on GitHub.

Names should be kept up-to-date

To resist software entropy, not only should you name entities properly, but you should also update them. Otherwise, names will become something worse than meaningless or confusing: **misleading**.

For example, let's say your class has the `$getMeans()` method.

- In its initial implementation, it used to return *precomputed* mean values.
- In its current implementation, it *computes* the mean values on the fly.

Therefore, it is misleading to continue to call it a getter method, and it should be renamed to (e.g.) `$computeMeans()`.

> **Tip**
>
> **Keep an eye out for API changes that make names misleading.**

Names should be pronounceable

This is probably the weakest of the requirements, but one can't deny the ease of communication when names are pronounceable.

If you are writing a function to generate a time-stamp, discussing the following function verbally would be challenging.

```
# generate year month date hour minute second
genymdhms()
```

Use consistent lexicon in a project

Once you settle down on a mapping from an abstraction to a name, use it consistently *throughout the code base*.

E.g., two similar methods here have different names across `R6` classes:

```
CreditCardAccount$new()$retrieve_expenditure()
DebitCardAccount$new()$fetch_expenditure()
```

Choose informative naming conventions

Having different name formats for different entities **acts like syntax highlighting**. That is, a name not only represents an entity but also provides hints about its nature.

Example of coding standards adopted in OSP organization

- Use all ALL_CAPS for constant variables (`public const double PI = 3.14;`)
- Prefix private/protected member variable with _ (`private int _currentDebt`)
- Use Pascal Casing for class names (`public class GlobalAccounting`)
- Use Pascal Casing for public and protected method name (`public void GetRevenues()`)
- Use Camel Casing for private method name (`private int balanceBooks()`)
- ...

Tip

Following *a* convention consistently is more important than *which* convention you adopt.

ICYMI: Available casing conventions

There are various casing conventions used for software development.

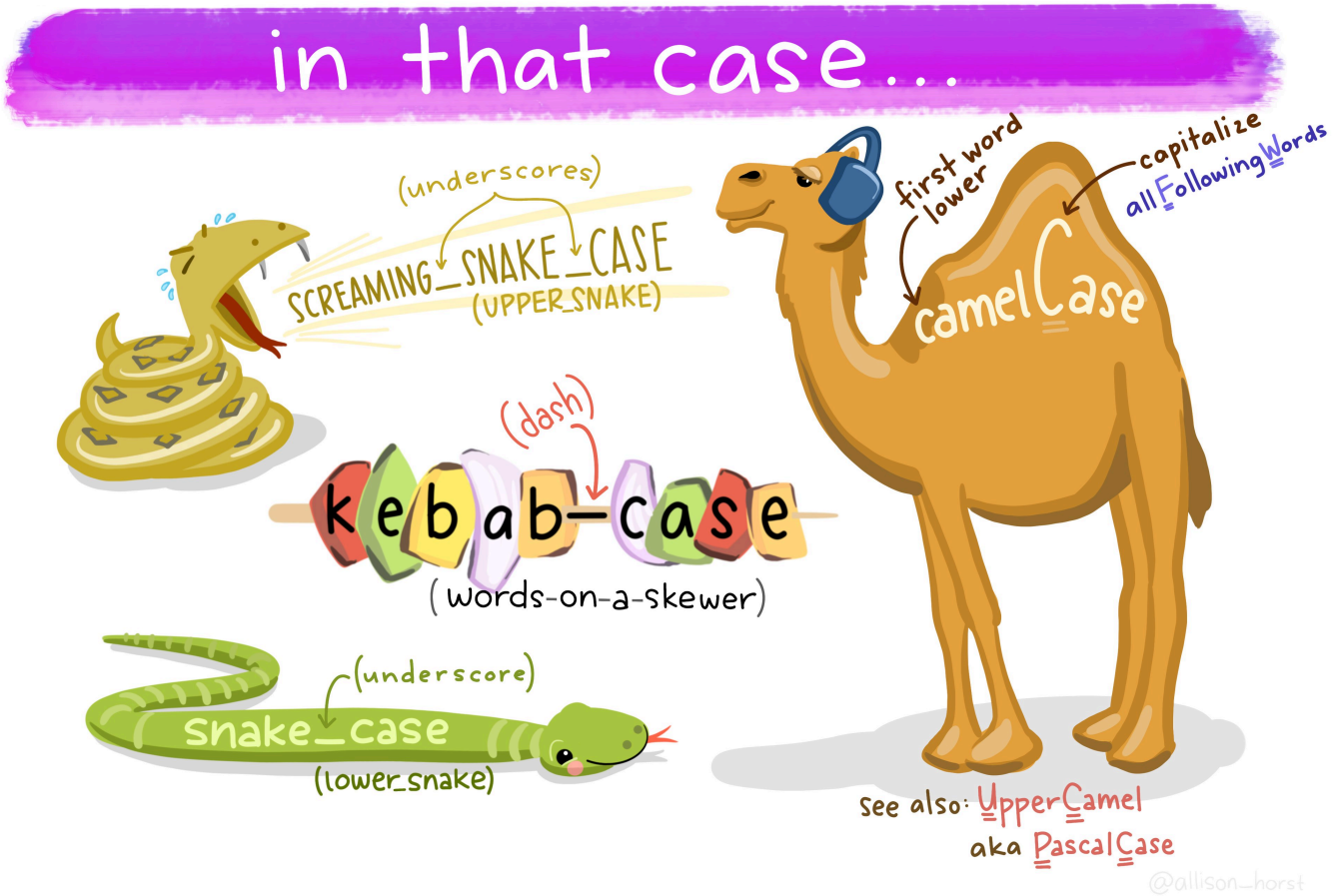

Illustration (CC-BY) by Allison Horst

Source code for these slides can be found on GitHub.

A sundry of don’ts

You won’t have to remember any of these rules if you follow the following principle:

“Names must be readable for the *reader*, not *writer*, of code.”

- **Don't use pop-culture references in names.** Not everyone can be expected to be familiar with them. E.g. `female_birdsong_recording` is a better variable name than `thats_what_she_said`.

- **Don't use slang.** You can't assume current or future developers to be familiar with them. E.g. `exit()` is better than `hit_the_road()`.

- **Avoid unintended meanings.** Do your due diligence to check dictionaries (especially Urban dictionary!) if the word has unintended meaning. E.g. `cumulative_sum()` is a better function name than `cumsum()`.

- **Avoid imprecise opposites**, since they can be confusing. E.g. parameter combination `begin`/`last` is worse than either `begin`/`end` or `first`/`last`.

- **Don't use hard-to-distinguish character pairs in names** (e.g., `l` and `I`, `O` and `0`, etc.). With certain fonts, it can be hard to distinguish `firstl` from `firstI`.

- **Don't use inconsistent abbreviations.** E.g. instead of using `numColumns` (*number* of columns) in one function and `noRows` (*number* of rows) in another, choose one abbreviation as a prefix and use it consistently.

- **Don't misspell to save a few characters.** Remembering spelling is difficult, and remembering *correct misspelling* even more so. E.g. don't use `hilite` instead of `highlight`. The benefit is not worth the cost here.

- **Don't use commonly misspelled words in English.** Using such names for variables can, at minimum, slow you down, or, at worst, increase the possibility of making an error. E.g. is it `accumulate`, `accummulate`, `acumulate`, or `acummulate`?!

- **Don't use numeric suffixes in names to specify levels.** E.g. variable names `level1`, `level2`, `level3` are not as informative as `beginner`, `intermediate`, `advanced`.

- **Don't use misleading abbreviations.** E.g., in R, `na.rm` parameter removes (`rm`) missing values (`NA`). Using it to mean "remove (`rm`) non-authorized (`NA`) entries" for a function parameter will be misleading.

- **Don't allow multiple English standards.** E.g. using both American and British English standards would have you constantly guessing if the variable is named (e.g.) `centre` or `center`. Adopt one standard and stick to it.

- **Don't use similar names for entities with different meanings.** E.g. `patientRecs` and `patientReps` are easily confused because they are so similar. There should be at least two-letter difference: `patientRecords` and `patientReports`.

Case studies

Looking at names in the wild that violate presented guidelines.

This is **not** to be taken as criticisms, but as learning opportunities to drive home the importance of these guidelines.

Violation: Breaking (domain) conventions

R is a programming language for statistical computing, and function names can be expected to respect the domain conventions.

Statistical distributions can be characterized by centrality measures, and R has functions with names that wouldn't surprise you, **except one**:

```
x <- c(1, 2, 3, 4)
mean(x)   # expected output
median(x) # expected output
mode(x)   # unexpected output!
```

Violation: Generic name

The parameter `N` in `std::array` definition is too generic.

```
template<
    class T,
    std::size_t N
> struct array;
```

Violation: Inconsistency in naming

ggplot2 is a plotting framework in R, and supports both **British** and **American** English spelling standards. But does it do so *consistently*?

Function names

```
# works
guide_colorbar(...)

# this works as well!
guide_colourbar(...)
```

Function parameters

```
# works
aes(color = ...)

# this works as well!
aes(colour = ...)
```

Violation: Room for misunderstanding

In Python, `filter()` can be used to apply a function to an iterable.

```
1  list(filter(lambda x: x > 0, [-1, 1]))
```

etc.

It is easy to find such violations.

But, whenever you encounter one, make it a personal exercise to come up with a better name.

Naming and good design

Deep dive into benefits of thoughtful naming for an entity at the heart of all software: ***function***

Following Unix philosophy

Unix philosophy specifies the golden rule for writing good a function:
"Do One Thing And Do It Well."

Finding a descriptive name for a function can inform us if we are following this rule.

Consider a function to extract a table of regression estimates for a statistical model. For convenience, it also allows sorting the table by estimate.

Naming is hard

Trying to find a name highlights that the function is doing more than one thing.

```
`???` <- function(model, sort = "asc") {
  # code to extract estimates from model
  ...
  # code to sort table
  ...
}
```

Naming is easy

These individual functions are easier to read, understand, and test.

```
extract_estimates <- function(model) {
  # code to extract estimates from model
  ...
}

sort_estimates <- function(table, sort = "asc") {
  # code to sort table
  ...
}
```

Function parameter names

When it comes to writing a good function, finding a good name for a parameter can also reveal design problems.

E.g. a boolean or flag parameter name means function is doing more than one thing.

Consider a function that converts Markdown or HTML documents to PDF.

Boolean parameter name

Doing more than one thing.

```r
convert_to_pdf <- function(file, is_markdown = FALSE) {
  if (is_markdown) {
    # code to convert Markdown to PDF
    ...
  }

  if (!is_markdown) {
    # code to convert HTML to PDF
    ...
  }
}
```

Non-boolean parameter name

Doing one thing.

```r
convert_md_to_pdf <- function(file) {
  # code to convert Markdown to PDF
  ...
}

convert_html_to_pdf <- function(file) {
  # code to convert HTML to PDF
  ...
}
```

“In your name I will hope, for your name is good.”

- Psalms 52:9

Benefits of good names

"What's in a name?" Well, everything!

- Intent-revealing names make the **code easier to read**.
- Trying to find good names forces you to detach from the problem-solving mindset and to **focus on the bigger picture** that motivates this change. This is critical for thoughtful software design.
- Searching for precise names requires clarity, and seeking such clarity **improves your own understanding** of the code.
- Naming precisely and consistently **reduces ambiguities and misunderstandings**, which in turn reduces the possibility of bugs.
- Good names **reduce the need for documentation**.
- Consistent naming **reduces cognitive overload** for the developers and makes the code more maintainable.

Challenges

Initially, you may struggle to find good names and settle down for the first serviceable name that pops into your head.

Resist the urge!

Worth the struggle

Adopt an investment mindset and remember that the little extra time invested in finding good names early on will pay dividends in the long run by reducing the accumulation of complexity in the system.

The more you do it, the easier it will get!

And, after a while, you won't even need to think long and hard to come up with a good name. You will instinctively think of one.

“Using understandable names is a foundational step to producing quality software.”

- Al Sweigart

Further Reading

For a more detailed discussion about how to name things, see the following references.

References

- McConnell, S. (2004). *Code Complete*. Microsoft Press. (**pp. 259-290**)
- Boswell, D., & Foucher, T. (2011). *The Art of Readable Code*. O'Reilly Media, Inc. (**pp. 7-31**)
- Martin, R. C. (2009). *Clean Code*. Pearson Education. (**pp. 17-52**)
- Ousterhout, J. K. (2018). *A Philosophy of Software Design*. Palo Alto: Yaknyam Press. (**pp. 121-129**)
- Goodliffe, P. (2007). *Code Craft*. No Starch Press. (**pp. 39-56**)
- Padolsey, J. (2020). *Clean Code in JavaScript*. Packt Publishing. (**pp. 93-111**)
- Thomas, D., & Hunt, A. (2019). *The Pragmatic Programmer*. Addison-Wesley Professional. (**pp. 238-242**)
- Ottinger's Rules for Variable and Class Naming
- For a good example of organizational naming guidelines, see Google C++ Style Guide.

For more

If you are interested in good programming and software development practices, check out my other slide decks.

Find me at...

 Twitter

 LikedIn

 GitHub

 Website

 E-mail

Source code for these slides can be found on GitHub.

Thank You

And Happy Naming! 😊

Session information

```
1  sessioninfo::session_info(include_base = TRUE)
```

Source code for these slides can be found on GitHub.