Dealing with the Second Hardest Thing in Computer Science

Thoughts on naming things for software development

Indrajeet Patil



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

- Phil Karlton

Why naming matters

The hidden cost of poor naming

Bad names create a cascade of problems that compound over time.

Immediate consequences:

- Code reviews take longer as reviewers struggle to understand intent
- Debugging becomes detective work instead of systematic analysis
- New team members need extensive onboarding to decode the codebase

Long-term impact:

- Technical debt accumulates as developers avoid touching poorly-named code
- Bug introduction rates increase due to misunderstanding
- Refactoring becomes risky when you can't trust what code actually does

1 The multiplication effect

Poor naming spreads confusion throughout the entire system.

Good names pay dividends

Well-chosen names transform code from puzzles into stories.

Development velocity:

- Code reviews focus on logic rather than deciphering meaning
- Debugging targets the right components faster
- New features build confidently on existing foundations

Maintenance benefits:

- Refactoring becomes safe and predictable
- Bug fixes address root causes rather than symptoms
- Documentation writes itself when names are self-explanatory



The investment mindset

Time spent on naming is not overhead—it's an investment that pays compound interest.

Naming Strategies

"The beginning of wisdom is to call things by their proper name." - Confucius

Although all code examples are in Python, the following advice on naming 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, the design may be unclear.

i 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.

Good names are a form of documentation

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

Poor names require more comments: Good names are self-documenting:

```
# function to convert temperature
# from Fahrenheit to Celsius scale
# temp is the temperature in Fahrenheit
# def unit_converter(temp: float):
# pass
```

```
1 def fahrenheit_to_celsius(temp_fahrenheit: float):
2  pass
```



Good names rarely require readers to read the documentation to understand what they represent.

Alternatives to generic names

For longer loops, use meaningful names instead of i, j, k:

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

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

All variables are temporary in some sense. Calling one tmp is inviting carelessness.

```
1  # generic name
2  if right < left:
3    tmp = right
4    right = left
5    left = tmp</pre>
1  # more descriptive
2  if right < left:
3    old_right = right
4    right = left
5    left = old_right
```



Even when you think you need generic names, you are better off using more descriptive names.

Maintain consistency across codebase

Consistent naming **reduces cognitive burden** by allowing readers to safely reuse knowledge across contexts.

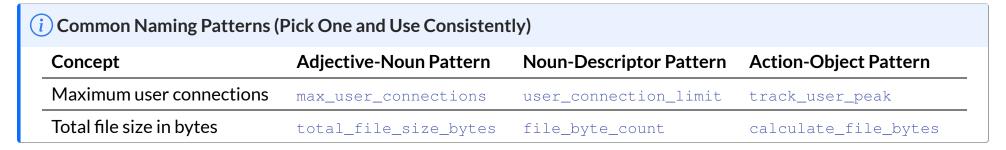
Avoid conflicting meanings for the same word:

```
1  # inconsistent - size means different things
2  size = len(x.encode('utf-8'))  # bytes
3  size = len(a)  # elements
4
5  # inconsistent - different words, same concept
6  CreditCardAccount().retrieve_expenditure()
7  DebitCardAccount().fetch_expenditure()
```

```
# consistent - clear distinctions
byte_size = len(x.encode('utf-8'))
length = len(a)

# consistent - same word, same concept
CreditCardAccount().retrieve_expenditure()
DebitCardAccount().retrieve_expenditure()
```

Use naming molds (templates) consistently:





Enable safe assumptions about names across different contexts. Mixed patterns force readers to decode different templates, increasing cognitive load.

Unnecessary details in names should be removed...

```
1 # okay
                                                            # better
2 convert_to_string()
                                                            to string()
3 file object
4 str_name # type prefix notation
```

(i) Avoid redundancy

- In type names, avoid using class, data, object, and type (e.g. bad: classShape, good: Shape)
- In function names, avoid using be, do, perform, etc. (e.g. bad: doAddition(), good: add())

but important details should be kept!

```
1 # okay
                                                             # better
2 child_height
3 password
                                                            hex id
                                                          5 ip_address
5 address
```

```
child_height_cm
plaintext_password
```



If some information is critical to know, it should be part of the name.

Balance precision with conciseness

Find the right level of detail for your context—precise enough to be clear, concise enough to be readable.

Use context to eliminate redundancy:

```
1 # redundant in context
2 Router.run_router()
3 BeerShelf.beer_count
4 child_height # missing units

1 # leverages context
2 Router.run()
3 BeerShelf.count
4 child_height # critical detail
```

Find the precision sweet spot:

```
1 # too imprecise → okay → good → unnecessarily precise
2 d → days → days_since_last_accident → days_since_last_accident_floor_4_lab_23
```

Match abstraction level to purpose:

```
1  # too specific for general function
2  def compare(value_before, value_after):
3   pass
1  # right abstraction level
2  def compare(value1, value2):
3  pass
```



Include critical information, exclude redundant details, and choose names that reflect the right level of abstraction for their purpose.

Names should be difficult to misinterpret

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

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

```
1 def get_char_position(x: int, y: int):
2  pass
```

How I interpret: "x and y refer to pixel positions for a character."

In reality: "x and y refer to line of text and character position in that line."

You can avoid such misinterpretation with better names:

```
1 def get_char_position(line_index: int, char_index: int):
2 pass
```



Precise and unambiguous names leave little room for misconstrual.

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.

```
1 # bad

2 n = x

3 nn = x ** 2

4 nnn = x ** 3
```

```
1  # good
2  n = x
3  n_square = x ** 2
4  n_cube = x ** 3
```



Any pair of names should be difficult to be mistaken for each other.

Names should be searchable

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

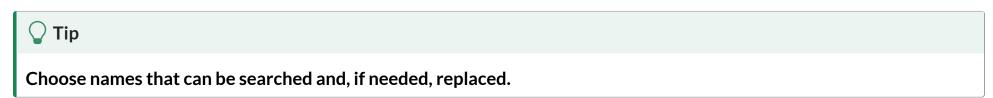
E.g., this function uses a and f parameters to represent an array and a function.

```
1 # bad
2 def array_map(a, f):
3 pass

1 # good
2 def array_map(arr, fun):
3 pass
```

If needed, it wouldn't be easy either to search for and/or to rename these parameters in the codebase because searching for a or f would flag **all** as and fs (**a**pi, **f**ile, etc.).

Instead, if more descriptive identifiers are used, both search and replace operations will be straightforward. In general, searchability of a name indexes how generic it is.



Names should be tab-friendly

Choose names that allow related items to group together when using tab completion or auto-complete features in your IDE.

E.g., if you're creating functions for different statistical tests, prefix them consistently so they group together:

```
1 # bad - scattered when tab-completing
2 chi_square_test()
3 fisher_exact()
4 t_test_paired()
5 wilcoxon_test()
6 mann_whitney()
7 anova_one_way()

1 # good - birds of a feather flock together
2 test_chi_square()
3 test_fisher_exact()
4 test_t_paired()
5 test_wilcoxon()
6 test_mann_whitney()
7 test_anova_one_way()
```

This principle also applies to variables, classes, and modules within the same domain.



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.).

```
1  # not great
2  if child:
3   if parent_supervision:
4   watch_horror_movie = True

1  # better
2  if is_child:
3  if has_parent_supervision:
4  can_watch_horror_movie = True
4  can_watch_horror_movie = True
```

In general, use positive terms for Booleans since they are easier to process.

```
1 # double negation - difficult
2 is_firewall_disabled = False

1 # better
2 is_firewall_enabled = True
```

But if the variable is only ever used in its false version (e.g. is_volcano_inactive), the negative version can be easier to work with.



Break English rules to improve clarity

Some nouns have identical singular/plural forms, making "incorrect" plurals clearer.

```
1 # unclear - one fish or many fish?
2 [f for f in fish if f.is_healthy()]

1 # clearer - obviously plural
2 [fish for fish in fishes if fish.is_healthy()]
```

Similarly: peoples, sheeps, deers, feedbacks, etc. can be clearer than grammatically correct forms.



Helping readers understand the code more easily is often worth breaking grammatical rules.

Focus on problem domain

Names should reflect **what** the code does (problem domain) rather than **how** it does it (implementation domain).

Choose business purpose over technical mechanism:

```
1 # implementation domain - how it works
2 binary_search_users()
3 hash_table_lookup()
4 sql_query_products()
5
6 # data structure in name
7 bonuses_pd # pandas DataFrame
8 aws_s3_url # AWS bucket
```

```
1 # problem domain - what it does
2 find_user()
3 get_customer()
4 fetch_products()
5
6 # implementation independent
7 bonuses # any data structure
8 bucket_url # any cloud service
```

Benefits of problem domain names:

- Remain meaningful when implementations change
- Reduce maintenance cost (no rename needed when switching technologies)
- Focus on business logic rather than technical details



Good names don't need to change when implementation details change.

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.

```
1 # bad
2 test1
3 my_test
4 retrieve_commands
5 serialize_success
```

```
1 # good
2 test_array
3 test_multilinear_model
4 test_all_the_saved_commands_should_be_retrieved
5 test_should_serialize_the_formula_cache_if_required
```

(i) Note

Don't hesitate to choose lengthy names for test functions.

Unlike regular functions, long names are less problematic for test functions because

- they are not visible or accessible to the users
- they are not called repeatedly throughout the codebase

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 .get_means() 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.) . compute_means().



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.

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

This is a much better (and pronounceable) alternative:

```
1 generate_timestamp()
```

Additionally, avoid naming separate entities with homonyms.

Discussing entities named waste and waist is inevitably going to lead to confusion.

Follow and respect naming conventions

Conventions create predictable patterns that reduce cognitive load and enable safe assumptions.

Language and domain conventions matter:

- i Informative conventions act like syntax highlighting
- UPPER_CASE for constants (MAX_RETRIES = 5)
- snake_case for variables/functions (user_age, calculate_total())
- PascalCase for classes (BankAccount, DataProcessor)
- Prefix private attributes (self._balance)



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.



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 knows 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., 1 and 1, 0 and 0, etc.). With certain fonts, first1 and first1 look identical.

- **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, accumulate, accumulate, or acumulate?!
- 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.
- Don't use uncommon English words. Stick to common parlance that most developers understand. E.g. start_process() is better than commence_process(), get_list() is better than procure_list(), find_user() is better than ascertain_user().

Naming and good design

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

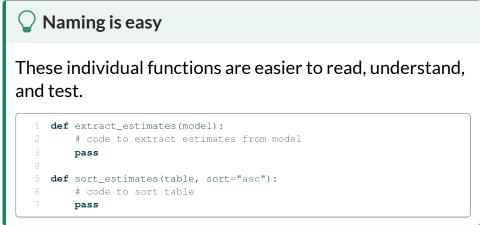
Following Unix philosophy

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.





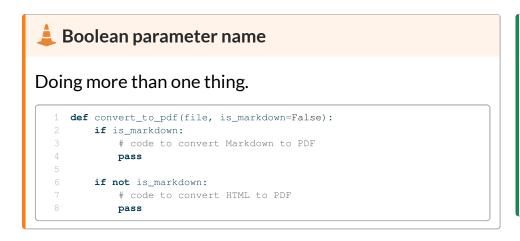
Functions with and or or in their names are dead giveaways that they don't follow the Unix philosophy.

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.



```
Non-boolean parameter name

Doing one thing.

def convert_md_to_pdf(file):
    # code to convert Markdown to PDF
    pass

def convert_html_to_pdf(file):
    # code to convert HTML to PDF
    pass
```

Utilizing tools

Naming limitations of QA tools

Static analysis tools can only do so much when it comes to naming.

What they CAN do:

- Enforce naming conventions
- Check for reserved keywords
- Detect naming pattern violations
- Flag overly short or long names
- Ensure consistent formatting

What they CANNOT do:

- Understand the intent behind your code
- Suggest meaningful names based on context
- Assess whether names represent what entities do
- Determine problem domain consistency
- Evaluate clarity for future developers

(i) The fundamental limitation

Tools can enforce *syntax* **but not** *semantics.* Good naming requires human understanding of both the problem and the solution.

Generative Al tools can be valuable allies

Al tools have context of your entire codebase and can provide meaningful names.

How AI tools can help:

- They see the whole function/class and understand relationships
- They recognize patterns across different programming domains
- They can spot inconsistent naming across your codebase
- They propose multiple naming options with rationales

```
1 # Your initial attempt
2 def process_data(x):
3    pass
4
5 def calc(a, b):
6   pass
7
8 temp_var = get_info()
```

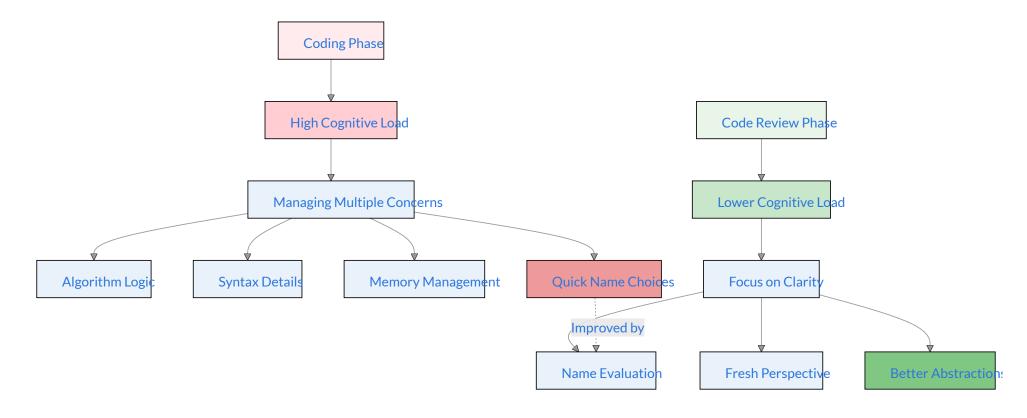
```
1  # AI suggestions
2  def validate_user_input(user_data):
3    pass
4
5  def calculate_compound_interest(principal, rate):
6   pass
7
8  user_profile = get_user_info()
```

Symbiotic Naming

Try to come up with good names yourself. Then, ask AI tools to validate, assess, or suggest improvements.

Code Review: A fresh perspective

When coding, we operate at peak cognitive load and this mental overload makes it the worst time to choose thoughtful names.





The Code Review Advantage

It provides the mental space needed to evaluate whether names truly capture the intent and abstraction level of code.

Benefits of good names

"In your name I will hope, for your name is good." - Psalms 52:9

"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, reducing 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!

"Using understandable names is a foundational step to producing quality software." - Al Sweigart

Thank You

And Happy Naming! 😊

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)
- Hermans, F. (2021). The Programmer's Brain. Manning Publications. (pp. 127-146)
- 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...

- **Y** Twitter
- in LikedIn
- **GitHub**
- **W**ebsite
- **E-mail**