Dealing with the Second Hardest Thing in Computer Science

Thoughts on naming things for software development

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

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

> ⚠ **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.

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 heavy commenting:

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

Contrast it with this:

```
def fahrenheit_to_celsius(temp_fahrenheit: float) -> float:
    pass
```

No need for a comment here!

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

Generic names should follow conventions

Generic names work *only if* you follow established conventions.

Examples:

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

```
for j in range(len(arr)):
    for i in range(len(arr[j])):
```

- `tmp` shouldn't be used to store objects that are not temporary
- `retVal` shouldn't be used for objects not returned from a function

> **Tip**
>
> **Don't violate reader assumptions about what generic names represent.**

Alternatives to generic names

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

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

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

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

```
# generic name
if right < left:
    tmp = right
    right = left
    left = tmp
```

```
# more descriptive
if right < left:
    old_right = right
    right = left
    left = old_right
```

Tip

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

Names should be consistent

Consistent names **reduce cognitive burden** by allowing readers to safely reuse knowledge across contexts.

These names are inconsistent—*size* means different things in different contexts:

```python
# context-1: `size` stands for number of memory bytes
size = len(x.encode('utf-8'))  # bytes

# context-2: `size` stands for number of elements
size = len(a)  # length
```

```python
# context-1:
size = len(x.encode('utf-8'))

# context-2:
length = len(a)
```

Tip

Enable safe assumptions about names across different contexts.

Unnecessary details in names should be removed...

```
# okay
convert_to_string()
file_object
str_name  # type prefix notation
```

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

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!

```
# okay
child_height
password
id
address
```

```
# better
child_height_cm
plaintext_password
hex_id
ip_address
```

Tip

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

But, if doing so imposes ambiguity, then you can of course tolerate some redundancy.

```
# bad
MediaPlayer.play()
```

```
# better
MediaPlayer.play_audio()
MediaPlayer.play_video()
```

Tip

Shorten names with the help of context.

Names should be precise but not *too* long

How precise (and thus long) the name should be is a **contextual 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

# ...
```

Tip

Don't go too far with making names precise.

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
```

Tip

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.

```
# bad
n = x
nn = x ** 2
nnn = x ** 3
```

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

Tip

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.

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

```
# good
def array_map(arr, fun):
    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** *a*s and *f*s (**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.

> **Tip**
>
> **Choose names that can be searched and, if needed, replaced.**

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:

```
# bad - scattered when tab-completing
chi_square_test()
fisher_exact()
t_test_paired()
wilcoxon_test()
mann_whitney()
anova_one_way()
```

```
# good - birds of a feather flock together
test_chi_square()
test_fisher_exact()
test_t_paired()
test_wilcoxon()
test_mann_whitney()
test_anova_one_way()
```

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

> **Tip**
>
> **Use consistent prefixes or namespacing to group related functionality together.**

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, Python convention is to use PascalCase for class names and snake_case for variables.

```python
# non-conventional
class playerentity:
    def __init__(self):
        self.HairColor = ""
```

```python
# conventional
class PlayerEntity:
    def __init__(self):
        self.hair_color = ""
```

Tip

Don’t break conventions unless other guidelines require overriding them for consistency.

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 parent_supervision:
        watch_horror_movie = True
```

```
# better
if is_child:
    if has_parent_supervision:
        can_watch_horror_movie = True
```

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

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

```
# better
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.

> **Tip**
>
> **Boolean variable names should convey what true or false values represent.**

Break English rules to improve clarity

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

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

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

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

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

Choose names from problem domain

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

Examples of problem domain vs implementation domain naming:

```
# implementation domain - how it works
binary_search_users()
hash_table_lookup()
linked_list_add()
sql_query_products()
json_parse_config()
thread_pool_execute()
```

```
# problem domain - what it does
find_user()
get_customer()
add_item()
fetch_products()
load_config()
process_request()
```

The problem domain names remain meaningful even if you switch from binary search to linear search, from hash tables to arrays, or from SQL to NoSQL databases.

> **Tip**
>
> **Name based on the business purpose, not the technical mechanism.**

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:

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

aws_s3_url  # AWS bucket
gcp_url     # GCP bucket
```

```
# good
bonuses    # data structure independent

bucket_url # cloud service independent
```

Note that good names don't need to change even if the implementation details change.

> **Tip**
>
> **Names should be independent of implementation details.**

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.

```
1  # bad
2  def compare(value_before, value_after):
3      pass
```

```
1  # good
2  def compare(value1, value2):
3      pass
```

Note that the good parameter names clarify the general purpose of the function, which is to compute difference between *any* two values, not just before and after values.

> **Tip**
>
> **Choose names that reflect the higher-level concept.**

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
test1
my_test
retrieve_commands
serialize_success
```

```
# good
test_array
test_multilinear_model
test_all_the_saved_commands_should_be_retrieved
test_should_serialize_the_formula_cache_if_required
```

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()`.

> **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()
```

This is a much better (and pronounceable) alternative:

```
generate_timestamp()
```

Additionally, avoid naming separate entities with homonyms.

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

Use consistent lexicon in a project

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

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

```
1 CreditCardAccount().retrieve_expenditure()
2 DebitCardAccount().fetch_expenditure()
```

Both of these methods should either be named `.retrieve_expenditure()` or `.fetch_expenditure()`.

> **Tip**
>
> **Consistency of naming conventions should be respected at both narrow and broad scopes.**

Naming molds: Templates for consistency

Feitelson's research reveals that developers use **naming molds**—consistent templates for constructing identifiers.

Examples of Different Naming Molds

Concept	Mold A	Mold B	Mold C
Maximum user connections	`max_user_connections`	`user_connection_limit`	`peak_user_count`
Total file size in bytes	`total_file_size_bytes`	`file_byte_count`	`accumulated_file_bytes`
Minimum password length	`min_password_length`	`password_length_floor`	`shortest_password_chars`
Current active sessions	`current_active_sessions`	`active_session_count`	`live_session_tally`

The Consistency Advantage

Pick one mold and use it consistently. Mixed molds in the same codebase force readers to decode different patterns, increasing cognitive load.

Feitelson's Three-Step Model

1. **Select concepts** to include in the name
2. **Choose words** to represent each concept
3. **Construct the name** using a consistent mold

This deliberate process produces names that are more precise and maintainable

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 Python naming convention guidelines

- Use UPPER_CASE for constants (`PI = 3.14159`, `MAX_RETRIES = 5`)
- Use snake_case for variables and functions (`user_age = 25`, `def calculate_total():`)
- Use PascalCase for class names (`class BankAccount:`, `class DataProcessor:`)
- Prefix private attributes with single underscore (`self._balance = 0`)
- Use trailing underscore to avoid keyword conflicts (`class_ = "math"`, `type_ = "int"`)
- Use descriptive module names in lowercase (`utils.py`, `data_handler.py`)

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.

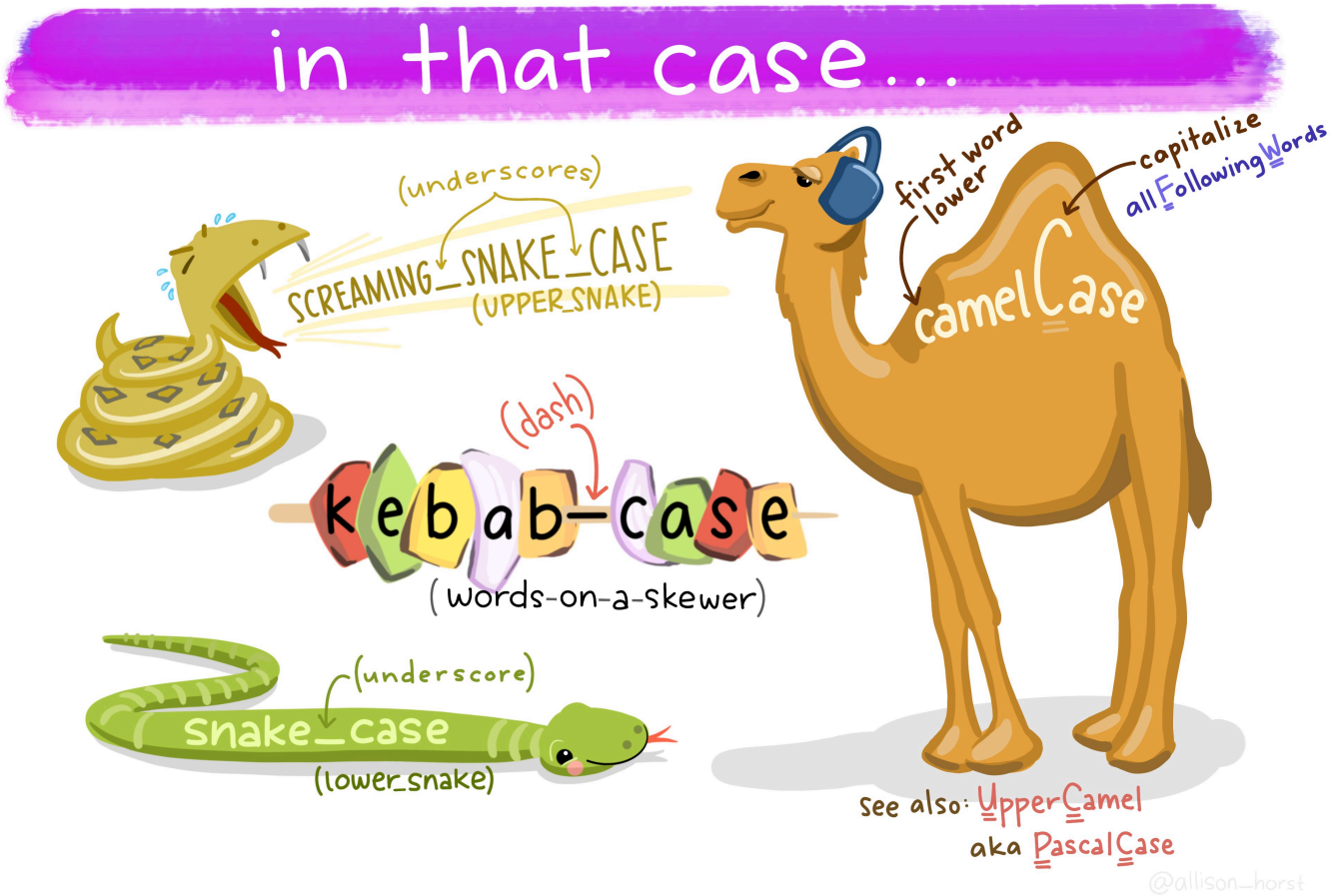


Illustration (CC-BY) by Allison Horst

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., `l` and `I`, `O` and `0`, etc.). With certain fonts, `firstl` and `firstI` 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`, `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`.

- **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.

Naming is hard

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

```
def mystery_function(model, sort="asc"):
    # code to extract estimates from model
    # code to sort table
    pass
```

Naming is easy

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

```
def extract_estimates(model):
    # code to extract estimates from model
    pass

def sort_estimates(table, sort="asc"):
    # code to sort table
    pass
```

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.

Boolean parameter name

Doing more than one thing.

```
def convert_to_pdf(file, is_markdown=False):
    if is_markdown:
        # code to convert Markdown to PDF
        pass

    if not is_markdown:
        # code to convert HTML to PDF
        pass
```

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
```

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

- Psalms 52:9

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

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

Generative AI tools can be valuable allies

AI 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

```python
# Your initial attempt
def process_data(x):
    pass

def calc(a, b):
    pass

temp_var = get_info()
```

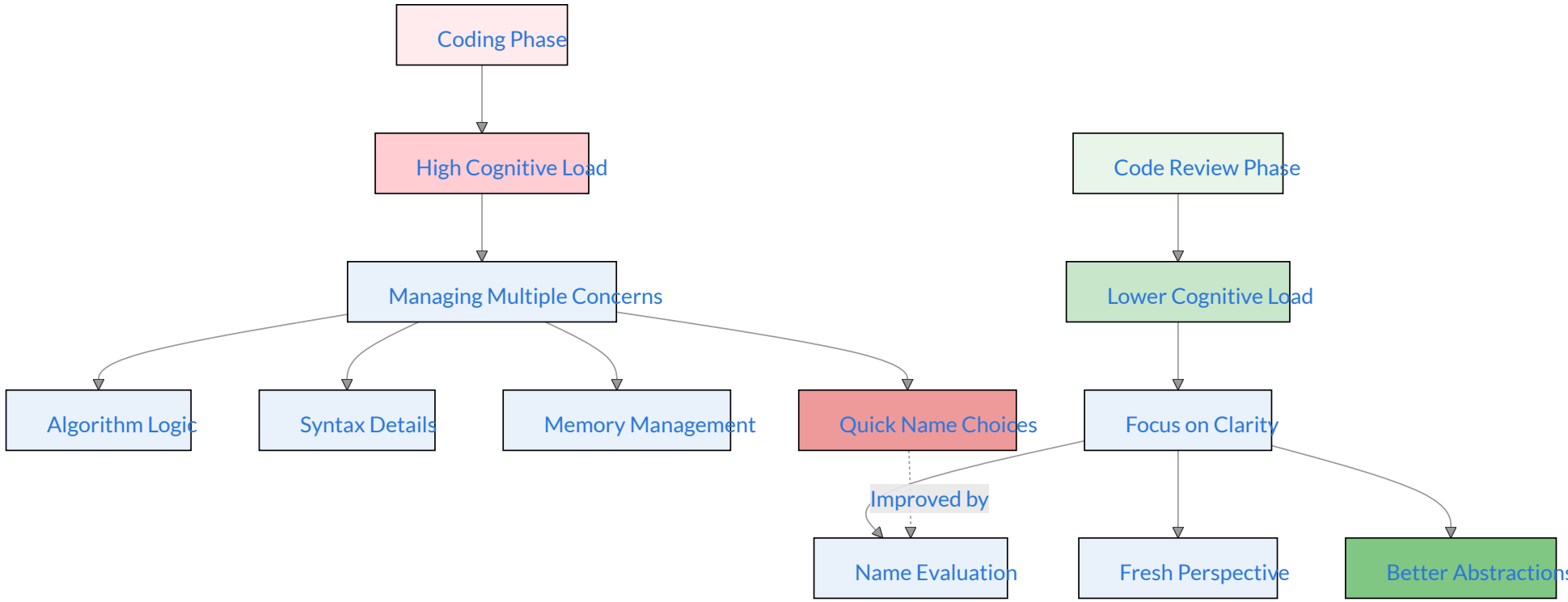
```python
# AI suggestions
def validate_user_input(user_data):
    pass

def calculate_compound_interest(principal, rate):
    pass

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 Review Advantage

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

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**, 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!

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**)
- 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...

- Twitter
- LikedIn
- GitHub
- Website
- E-mail

Thank You

And Happy Naming! 😊

Session information

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

```
─ Session info ─────────────────────────────────────────────────────────────
 setting  value
 version  R version 4.5.1 (2025-06-13)
 os       Ubuntu 24.04.3 LTS
 system   x86_64, linux-gnu
 ui       X11
 language (EN)
 collate  C.UTF-8
 ctype    C.UTF-8
 tz       UTC
 date     2025-08-29
 pandoc   3.7.0.2 @ /opt/hostedtoolcache/pandoc/3.7.0.2/x64/ (via rmarkdown)
 quarto   1.8.21 @ /usr/local/bin/quarto

─ Packages ─────────────────────────────────────────────────────────────────
 package     * version date (UTC) lib source
 base        * 4.5.1   2025-06-13 [3] local
 cli           3.6.5   2025-04-23 [1] RSPM
 compiler      4.5.1   2025-06-13 [3] local
 datasets    * 4.5.1   2025-06-13 [3] local
```