How to Refactor Your Code to Be More Clear and Understandable

Uday Turakhia

This guide assumes an intermediate level of coding knowledge, meaning you are familiar with basic control structures, functions, and variable naming conventions. The focus is on improving the readability and maintainability of your code through refactoring techniques.

You will learn how to:

- Avoid unnecessary nesting to simplify code logic.
- Use meaningful names for variables, functions, and constants to make the code more self-explanatory.
- Replace comments with cleaner, more expressive code.

By the end of this guide, you'll be able to write code that is easier to understand and maintain, without relying on excessive comments.

1 Don't Nest Your Code

1.1 Step 1: Understand Code Nesting

Nesting occurs when one block of code, such as an `if` statement or loop, is placed inside another. For example, consider the following code:

```
def process_data(data):
    if data:
        for item in data:
            if item.is_valid():
                # Do something
```

In this case, the code has three levels of nesting: an `if` block, a `for` loop, and another `if` block. This can make the function harder to follow as the depth increases.

1.2 Step 2: Recognize the Problems with Nesting

When code is heavily nested, it becomes difficult to read, debug, and maintain. Each level of indentation forces the reader to track multiple conditions simultaneously, increasing the cognitive load. Let's look at a more complex example:

```
def complex_function(data):
    if data:
        for item in data:
            if item.is_valid():
                for sub_item in item.get_sub_items():
                    if sub_item.is_active():
                        # Do something
```

This code becomes harder to follow as the indentation levels increase, and adding more conditions will only make it worse.

Too many nested blocks make the logic of the function hard to follow and can lead to errors. Avoid nesting more than three levels deep.

1.3 Step 3: Refactor with *Extraction*

To reduce nesting, one effective strategy is *Extraction*. This involves moving nested code into separate functions, which helps make the logic clearer and more modular. Here's how we can refactor the example:

Before:

```
def process_data(data):
    if data:
        for item in data:
            if item.is_valid():
                # Do something
```

After (with Extraction):

```
def process_data(data):
    if data:
        for item in data:
            process_item(item)

def process_item(item):
    if item.is_valid():
        # Do something
```

In this refactored version, we moved the logic inside the loop into a new function, `process_item`. This keeps the code more readable by reducing the nesting levels.

Warning: When extracting code into new functions, ensure the functions are given meaningful names that clearly describe their responsibility. More on this later

Warning: When extracting code, be cautious of creating side effects. If the extracted code depends on or modifies shared variables, improper handling may lead to bugs that are hard to trace. Ensure proper passing of parameters and return values.

1.4 Step 4: Refactor with *Inversion*

Another technique for reducing nesting is *Inversion*, which involves flipping conditions and returning early from functions when a condition is not met. This

helps to keep the core logic at the top level and avoid unnecessary indentation.

Before:

```
def process_data(data):
    if data:
        if not data.is_valid():
            return
        # Continue processing
```

After (with Inversion):

```
def process_data(data):
    if not data or not data.is_valid():
        return
    # Continue processing
```

By inverting the condition, we handle the error case early and eliminate an unnecessary `else` block, making the function easier to read and maintain.

Warning: While early returns simplify the main logic, overusing them in a function can lead to multiple exit points, making debugging and testing more difficult. Use early returns judiciously.

Warning: Inverting logic can sometimes lead to confusion, especially when dealing with complex conditions. Make sure the logic remains clear and understandable after inversion.

2 Avoiding Bad Naming Practices

2.1 Step 1: Avoid Single-Letter Variables

Using single-letter variable names is a bad practice, as they don't convey the purpose of the variable. This practice may have originated from mathematics, where concise symbols are often preferred, but programming requires clear and meaningful names.

```
# Bad example:
x = 10
y = 20
result = x + y

# Good example:
width = 10
height = 20
area = width * height
```

Single-letter variable names should be avoided because they do not communicate any information about the variable's purpose. Always prefer descriptive names.

2.2 Step 2: Avoid Abbreviations

Abbreviating variable names can be misleading, as it relies on the reader's ability to infer the correct meaning from context. Abbreviations may save a few

keystrokes but often lead to confusion.

Before:

```
# Bad example with abbreviations:
int dpmt = getDepartment();
```

After:

```
# Good example with full names:
int department = getDepartment();
```

Abbreviations make it difficult to understand code, especially for someone new to the project. Use full names to avoid unnecessary confusion.

2.3 Step 3: Avoid Including Types in Variable Names

In modern programming, the type of a variable is either inferred or explicitly defined, so there's no need to include the type in the variable name. This practice, often referred to as Hungarian notation, is outdated.

```
# Bad example with type in the name:
int nCount = 5;

# Good example without type in the name:
int count = 5;
```

Including the type in a variable name (like adding an "n" prefix for numbers) is redundant in modern programming languages that have strong typing systems. Let the type system do the work.

2.4 Step 4: Specify Units in Variable Names

It's a good practice to include units in your variable names to make it clear what unit the variable is representing, especially for time or measurements.

```
# Good example with units:
int delaySeconds = 10;

# Good example with explicit types in some languages (e.g., C#):
TimeSpan delay = TimeSpan.FromSeconds(10);
```

Without units in the variable name, it can be unclear what value is being represented. Be explicit when dealing with measurements, like seconds, meters, or kilograms.

2.5 Step 5: Avoid Naming Classes "Base" or "Abstract"

Naming classes with prefixes like "Base" or "Abstract" does not provide meaningful information to the user. If you're struggling to name a parent class, it may indicate that the child class should be renamed instead.

```
# Bad example with "Base" in the name:
class BaseTruck:
    # Base class for trucks

class Truck(BaseTruck):
    # Child class

# Good example without "Base":
class Truck:
    # Parent class representing trucks

class TrailerTruck(Truck):
    # Specific type of truck
```

Using "Base" or "Abstract" in class names adds unnecessary detail. If the class is a general type, name it accordingly and provide more specific names to child classes.

2.6 Step 6: Avoid "Utils" or "Helper" in Module Names

When naming modules, avoid generic names like "Utils" or "Helper". Instead, organize the functionality into more descriptive modules or classes that represent specific purposes.

```
# Bad example with "Utils" class:
class MovieUtils:
    def sort_movies(self):
        pass

# Good example with meaningful class:
class MovieSorter:
    def sort(self):
        pass
```

The term "Utils" is overused and leads to code organization issues. Break down utility functions into appropriate classes with clear responsibilities.

3 Why You Should Avoid Writing Comments

3.1 Step 1: Replace Magic Numbers with Constants

Using numbers directly in code, often referred to as "magic numbers", is a bad practice because it is unclear what the number represents. Instead, replace magic numbers with constants that have meaningful names.

Before:

```
# Bad example with a magic number:
if status == 5:
    print("Message sent")
```

After:

```
# Good example with a constant:
MESSAGE_SENT = 5
if status == MESSAGE_SENT:
    print("Message sent")
```

Magic numbers make it hard to understand what the code is doing. Use constants to replace numbers that have a specific meaning, making the code self-documenting.

3.2 Step 2: Simplify Complex Conditions

When you have a complex condition in your code, it might be tempting to add a comment explaining what it does. Instead, break the condition down by introducing variables with meaningful names. This makes the code more readable without the need for a comment.

Before:

```
# Bad example with a complex condition and a comment:
if (user.is_admin() and not user.is_banned() and user.has_access())
    :
    # Check if the user is allowed to access the resource
    access_resource()
```

After:

```
# Good example with descriptive variables:
is_authorized = user.is_admin() and not user.is_banned()
if is_authorized and user.has_access():
    access_resource()
```

Warning: Complex conditions are hard to understand at a glance. Break them down into smaller variables with meaningful names to improve readability.

3.3 Step 3: Use Types to Eliminate the Need for Comments

In statically typed languages, types can convey important information about how the code works. This eliminates the need for comments that explain ownership, optionality, or other concepts that can be expressed through types.

Before:

```
# Bad example with a comment about ownership:
def get_resource():
    # Caller is responsible for freeing the resource
    return resource
```

After:

```
// Good example using types (C++ example):
std::unique_ptr<Resource> get_resource() {
    return std::make_unique<Resource>();
}
```

Using comments to explain concepts like ownership or optionality is error-prone. Instead, use the type system to communicate these ideas more clearly.

3.4 Step 4: Consider Functions for Complex Expressions

If you find yourself writing a comment to explain a particularly complex expression, it may be better to refactor that expression into a separate function. This way, the function name acts as documentation, and the comment becomes unnecessary.

Before:

```
# Bad example with a comment:
# Check if user is eligible for discount
if user.is_loyal() and user.has_active_subscription():
    apply_discount()
```

After:

```
# Good example with a function:
def is_eligible_for_discount(user):
    return user.is_loyal() and user.has_active_subscription()

if is_eligible_for_discount(user):
    apply_discount()
```

Warning: If an expression requires a comment, it might be a sign that it should be refactored into its own function. Use function names to make the code more readable.

3.5 Step 5: The Problems with Comments

While comments may seem helpful, they come with several issues:

- **Comments can become outdated**: When code changes, people often forget to update the corresponding comments, leading to discrepancies between the code and its explanation.
- **Comments don't prevent bugs**: Unlike code, comments do not undergo testing or validation, so they can easily introduce misunderstandings or errors.
- **Comments are subjective**: What seems clear to one person might be confusing to another. Instead of relying on comments, aim to make the code itself as clear as possible.

Comments can become a liability if they are not maintained or updated correctly. Rely on self-explanatory code whenever possible.

3.6 Step 6: Use Documentation for High-Level Descriptions

While comments are often unnecessary, documentation is still important, especially for describing high-level architecture, public APIs, and usage patterns. Tools like `Doxygen`, `pydoc`, and `JavaDoc` can generate documentation from code, ensuring that it stays up to date with the codebase.

Use documentation tools to describe the overall design and usage of your code.

Warning: Avoid putting too many implementation details into comments within the code itself.