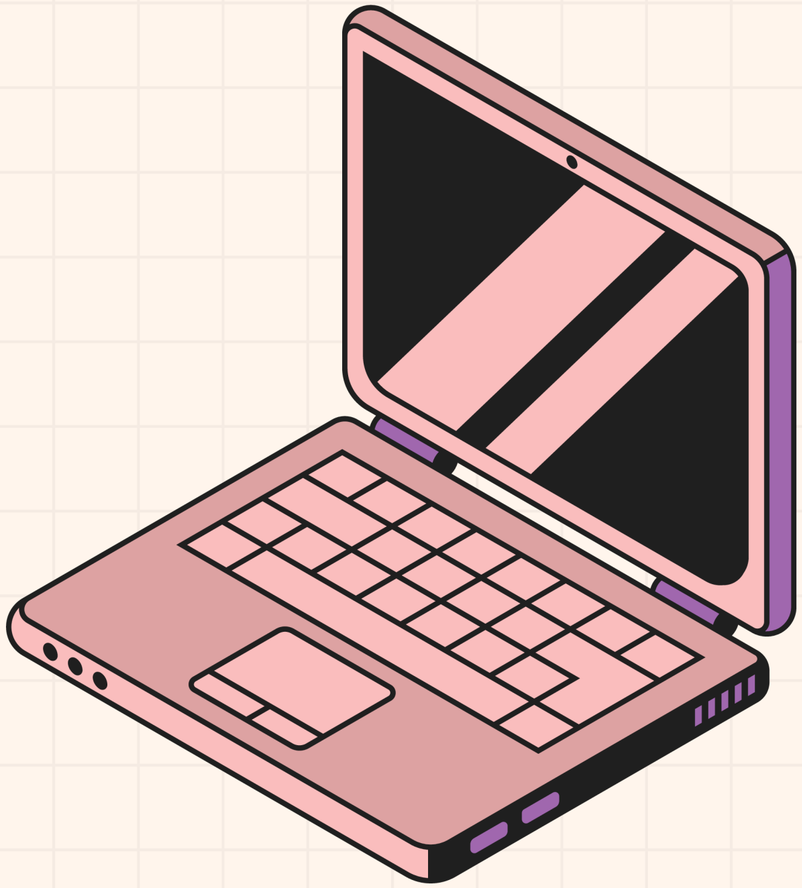
CLEAN CODE

The Principles of Writing Elegant Code.



INTRODUCTION

WHY CLEAN CODE MATTERS

In software development, writing code that simply works is not enough.

Poorly written code may function today, but it often becomes a burden tomorrow — harder to understand, maintain, and extend.

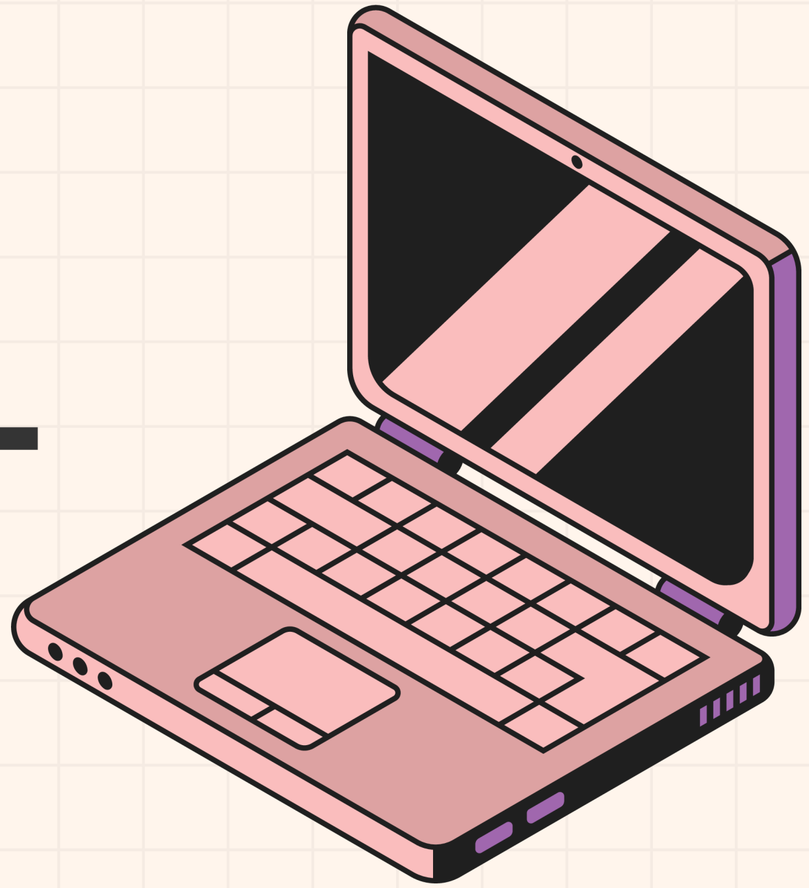
Clean code, on the other hand, is like a well-structured book:

- Easy to read and navigate
- Simple to modify and improve
- Less error-prone, saving time and cost in the long run

The goal of this presentation is to explore key principles from Clean Code by Robert C.

Martin and show how applying them can transform messy, complex code into clear, maintainable, and professional-quality software.

MEANINGFUL NAMES



Meaningful Names

Use Intention-Revealing Names

- Take care with your names and change them when you find better ones.
- Choosing good names takes effort but saves more than it takes.
- A name should tell you the why, what, and how.
- Be explicit about context, avoid relying on implications.

```
# Bad
d = 5  # elapsed time in days
```

```
# Good
elapsed_time_in_days = 5
```

```
# Bad
def calc(d):
    return d * 24
```

```
# Good
def convert_days_to_hours(days):
    return days * 24
```

```
# Bad
if x[0] == 4:
    flagged_cells.append(x)
```

```
# Good
if cell.status == FLAGGED:
    flagged_cells.append(cell)
```

MEANINGFUL NAMES

Avoid Disinformation & Make Meaningful Distinctions

- Avoid Disinformation
  - Don’t use names with misleading meanings (e.g., hp for hypotenuse)
  - Don’t include type names unless true (e.g., accountList when it’s not a real list)
  - Avoid names that differ in tiny, confusing ways:
    - XYZControllerForEfficientHandlingOfStrings
    - XYZControllerForEfficientStorageOfStrings
  - Never use l, O, or similar characters that look like 1 or 0.
- Make Meaningful Distinctions
  - Don’t just change names to please the compiler: `klass vs class`
  - Avoid number-series naming (a1, a2…) → meaningless.
  - Don’t add noise words (Info, Data, Object, Variable…):
    - Customer vs CustomerData vs CustomerInfo → no real distinction

```
# Bad  confusing
O = 1
l = 0
if O == l:
    pass

# Better
order_count = 1
limit = 0
```

MEANINGFUL NAMES

Use Pronounceable & Searchable Names

- Use Pronounceable Names
  - Names should be easy to say and discuss.
  - Use clear, descriptive terms instead of cryptic abbreviations.
  - Don’t use abbreviations that force people to “spell” names letter by letter: (e.g. pszqint vs record_id)
- Use Searchable Names
  - Prefer meaningful constants & variables instead of magic numbers.
  - Longer, descriptive names are better for searchability.
- Single-letter variables (like i, j) are fine for short, local loops.
- The longer the scope, the longer and more descriptive the name should be.

```
genymdhms = "2025-08-22 12:30:00"
generation_timestamp = "2025-08-22 12:30:00"
```

```
for i in range(5):
    print(i)
```

MEANINGFUL NAMES

Avoid Encodings

- Encoding adds mental burden and makes names harder to read & pronounce.

Hungarian Notation & Member Prefixes

- Don't put the type of the variable before it's name: `Boolean_state = T`
- Don’t Prefixes like m_ add unnecessary clutter; modern IDEs already highlight member variables.

Avoid Mental Mapping

- Use clear, descriptive names from the problem or solution domain—clarity is always more valuable than showing off cleverness.

Class & Method Names

- Class names should be nouns or noun phrases
- Methods should be verbs or verb phrases
- Avoid clever or joke-like names

MEANINGFUL NAMES

Pick One Word per Concept

- Use one consistent word for one concept
  (e.g., always use get, not a mix of fetch, retrieve, and get).

Don’t Pun

- Don’t reuse the same word for different meanings—this creates confusion (a pun).

Use Solution And Problem Domain Names

- Use solution domain terms (CS concepts, algorithms, design patterns, math terms) when they clearly convey meaning to programmers.
- If no clear technical (solution domain) term exists, use names from the problem domain to describe the concept.

MEANINGFUL NAMES

Add Meaningful Context

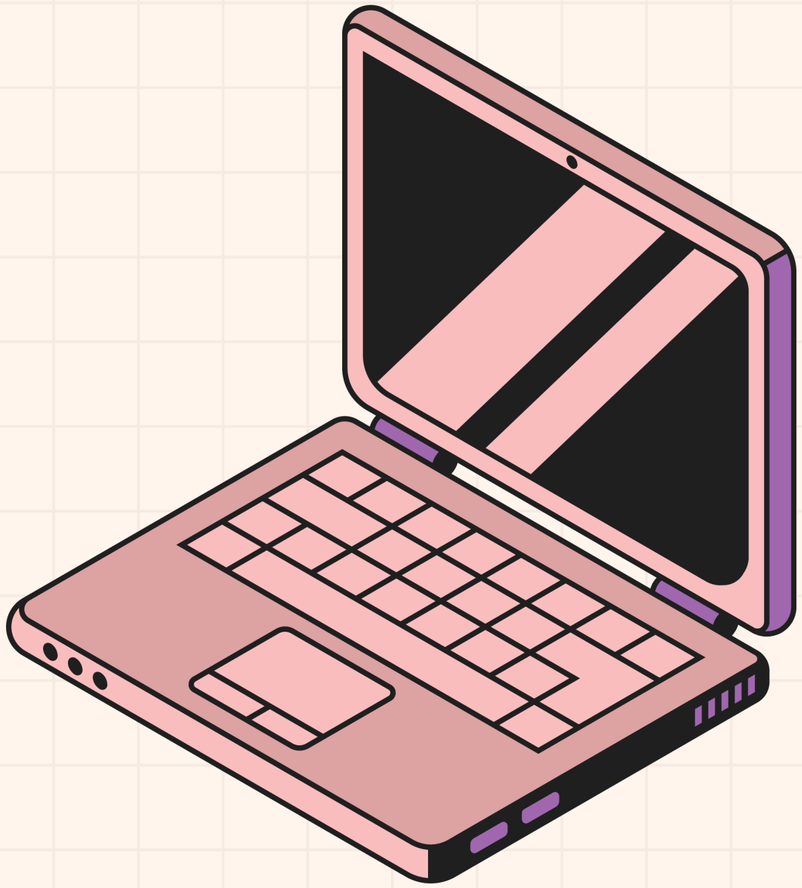
- Names alone are often unclear → put them inside well-named classes/functions.

```
# Unclear code with poor naming
number = "1"
verb = "is"
plural = ""
message = f"There {verb} {number} guess{plural}."
```

```
# Clean code within a meaningful class
class GuessStatisticsMessage:
    def __init__(self):
        self.number_of_guesses = 0

    def get_guess_message(self):
        if self.number_of_guesses == 1:
            return f"There is {self.number_of_guesses} guess."
        else:
            return f"There are {self.number_of_guesses} guesses."

# Example usage
stats = GuessStatisticsMessage()
stats.number_of_guesses = 1
print(stats.get_guess_message())

stats.number_of_guesses = 5
print(stats.get_guess_message())
```

- Good naming is more about communication and culture than technical skill.
- Don’t fear renaming—better names improve readability, and tools make refactoring safe.

FUNCTIONS



FUNCTIONS

Small!

- The first rule is that functions should be small. The second rule is that they should be even smaller.
- Functions should hardly ever be more than 20 lines long. A length of 2-4 lines is often perfect.
- The blocks inside if, else, or while statements should be one line long, which is usually a call to another function.

```
# Long function with multiple indentation levels
def render_page(page_data, is_suite):
    if page_data.is_test_page():
        if is_suite:
            # include suite setup
            ...
        # include page setup
        ...
        # render content
        ...
        if is_suite:
            # include suite teardown
            ...
        # include page teardown
        ...
```

```
# Smaller functions, each doing one thing
def render_page(page_data, is_suite):
    if page_data.is_test_page():
        include_setups_and_teardowns(page_data, is_suite)
    return page_data.get_html()
```

FUNCTIONS

Do One Thing & Abstraction Level

- A function should do one thing, do it well, and do it only.
- All the steps in the function should be at the same level of abstraction.
  - Don't mix high-level concepts
    (like get_page())
    with low-level details
    (like .append("\n"))
    in the same function.

- The Stepdown Rule: Code should read like a top-down story. A function should be followed by the functions it calls, moving down one level of abstraction at a time.

FUNCTIONS

Function Arguments

- The ideal number of arguments for a function is zero (niladic). One (monadic) or two (dyadic) are okay. Three (triadic) should be avoided.
- More than three arguments is a red flag and should almost never be used.
- Avoid Flag Arguments: Passing a boolean to a function is bad practice because it means the function does two different things (one for True, one for False).
    - Instead of adding a flag, write two separate functions.
- Argument Objects: If a function needs more than two or three arguments, consider grouping them into a class.

```
# Flag argument
def book(passenger, is_premium):
    if is_premium:
        # logic for premium booking
    else:
        # logic for regular booking
```

```
# Separate functions for each case
def book_premium(passenger):
    # logic for premium booking

def book_regular(passenger):
    # logic for regular booking
```

```
def draw_circle(x, y, radius): ...

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def draw_circle(center: Point, radius): ...
```

FUNCTIONS

Have No Side Effects

- Your function should not have hidden side effects. It should do what its name promises, and nothing else.

```
# The function name check_password implies it only checks,
# but it has a "side effect" of initializing a session.
def check_password(user, password):
    if user.password == password:
        Session.initialize() # Unexpected side effect!
        return True
    return False
```

- Command Query Separation (CQS):

A function should either be a "command" that does something (changes state) or a "query" that answers something (returns a value), but not both.

```
# Is "set" a verb (command) or an adjective (query)? Confusing.
if set_attribute("username", "bob"):
    ...```

# Separate the command from the query
if attribute_exists("username"):
    set_attribute("username", "bob")
    ...
```

FUNCTIONS

Error Handling

- Prefer Exceptions to Returning Error Codes: Returning error codes leads to deeply nested if statements and forces the caller to handle the error immediately.
- Exceptions are better because they separate the error-handling logic from the main "happy path" code, making both cleaner.
- Error Handling Is One Thing: A function that handles errors should do nothing else. A try block should be the very first thing in a function, with nothing after the catch/finally blocks.

```
# Returning error codes
result = delete_page(page)
if result == "error_1":
    if result == "error_2"
        if result == "error_3"
            # do next thing
        else:
            # handle error
    else:
        # handle error
else:
    # handle error
```

```
# Using exceptions
try:
    delete_page(page)
    # do next thing
except Exception as e:
    # handle error
```

```
def delete(self, page):
    try:
        self.delete_page(page)
    except Exception as e:
        self.log_error(e)
```

```
def delete_page(self, page):
    page.delete()
```

```
def log_error(self, e):
    self.logger.log(str(e))
```

FUNCTIONS

Don't Repeat Yourself (DRY)

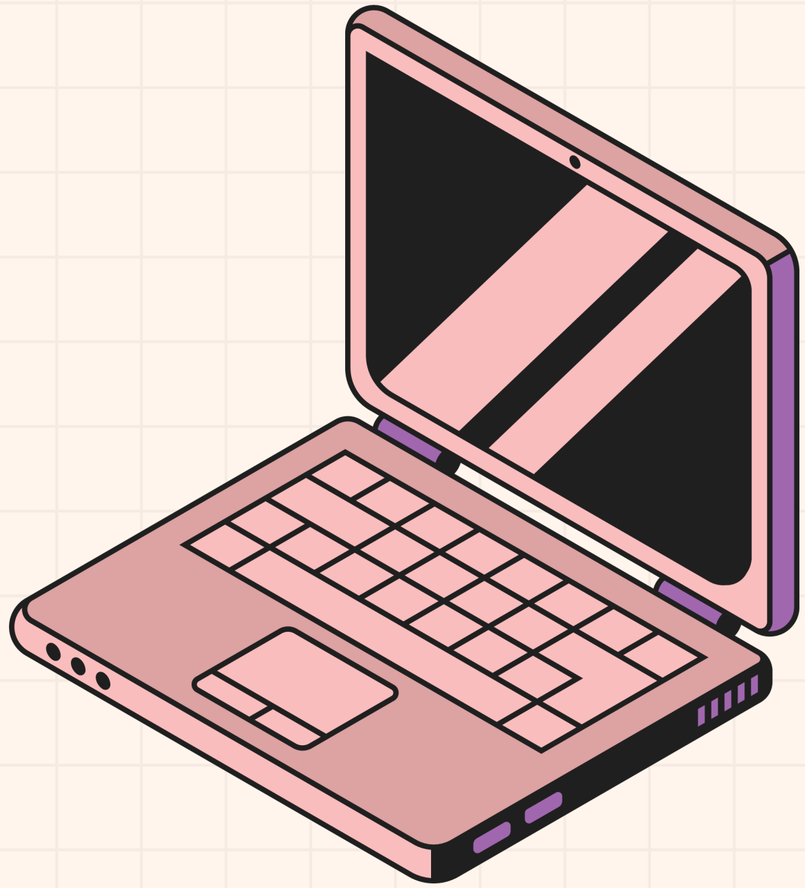
- Duplication is a major source of problems in software. Avoid it at all costs.
- If you see the same code structure or logic repeated, find a way to unify it into a single function or class. This makes the code easier to maintain and reduces the chance of errors.

“THE REAL GOAL OF WRITING FUNCTIONS IS NOT JUST CLEAN CODE, BUT TO TELL THE STORY OF THE SYSTEM THROUGH A CLEAR AND EXPRESSIVE LANGUAGE.”

COMMENTS

THE GOOD

THE BAD



COMMENTS

Good Comments

The Golden Rules

- Comments Do Not Make Up for Bad Code
  - The first priority should always be to write clean, expressive code.
  - Don’t write comments to explain messy or confusing logic. Clean the code instead.
- Explain Yourself in Code
  - Your code should be the primary source of documentation.
  - Strive to make the code express its intent so clearly that it doesn't need comments.
  - Use meaningful variable and function names.

```
// Bad: Relies on a comment
// Check if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) && (employee.age > 65))

// Good: Self-documenting code
if (employee.isEligibleForFullBenefits())
```

COMMENTS

Good Comments: When to Write Them

Comments that Add Value

- Legal Comments
  - Some comments are required for legal reasons, such as copyright notices.

```
// Copyright (C) 2003,2004,2005 by Object Mentor, Inc. All rights reserved.
// Released under the terms of the GNU General Public License version 2 or later.
```

- Informative Comments
  - Provide useful information that the code itself cannot express easily.

```
// Returns an instance of the Responder being tested.
protected abstract Responder responderInstance();
```

- Explanation of Intent
  - Explain why a certain design or implementation decision was made.

```
names = ["Ali", "Omar", "Ali", "Sara"]
# Using a set to remove duplicates quickly
unique_names = set(names)
```

COMMENTS

Good Comments: When to Write Them

Comments that Add Value

- Clarification
  - Use comments to clarify code that deals with complex or obscure external APIs or libraries that cannot be simplified.

```
text = "hello world"
# 'title()' capitalizes the first letter of each word
formatted = text.title()
```

- Warning of Consequences
  - Alert other developers to potential side effects or critical issues.

```
import os

# Warning: This will delete the file permanently and cannot be undone!
os.remove("data.txt")
```

COMMENTS

Bad Comments: Redundancy and Noise

Comments that Add Clutter

- Mumbling
  - Vague, unhelpful comments written just for the sake of it. Always strive for clarity.

```
# Do Something
result = max(numbers)
```

- Redundant Comments
  - Comments that state the obvious and repeat what the code already does.

```
// Bad: Redundant comment
i++; // Increment i

// This comment is completely useless. The code is already clear.
```

- Noise Comments
  - Comments that add no value and just repeat the code in a slightly different way.

```
// The day of the month.
private int dayOfMonth;
```

COMMENTS

Bad Comments: Redundancy and Noise

Comments that Add Clutter

- Mandated Comments
  - Comments written only because a rule or process requires them (e.g., forcing Javadocs for every single function, even obvious ones).

```java
public class Employee {
    private String name;
    /**
     * This method sets the employee name
     * @param name the employee name
     */
    public void setName(String name) {
        this.name = name;
    }
    /**
     * This method gets the employee name
     * @return the employee name
     */
    public String getName() {
        return name;
    }
}
```

COMMENTS

Bad Comments: Poor Practices

When a Comment Is the Wrong Tool

- Don’t Use a Comment When You Can Use a Function or Variable
  - Instead of explaining a complex block of code with a comment, extract it into a function with a descriptive name. This makes the code self-documenting.

```
# Bad Comment
# Calculate the area of a circle
area = 3.14159 * radius * radius
```

```
# Good Comment
def circle_area(radius):
    return 3.14159 * radius * radius
```

- Clutter and Poor Formatting
  - Avoid comments that add visual noise without providing value.
  - This includes:
    - Position Markers: Banners like // Actions //////////////// clutter the code.
    - Too Much Information: Keep comments concise and relevant.
    - HTML Comments: Avoid embedding HTML in source code comments.

COMMENTS

Bad Comments: Poor Practices

When a Comment Is the Wrong Tool

- Wrong Placement
  - Ensure the comment clearly relates to the code it describes and is placed in the correct location.
  - Avoid explaining parts of the system that are not relevant to the current file.

COMMENTS

Bad Comments: Outdated & Obsolete

The Dangers of Unmaintained Comments

- Misleading Comments
  - An incorrect comment is far more dangerous than no comment at all. Always update comments when the code changes.

```python
# Bad Comment: Misleading
def filter_even_numbers(numbers):
    # This function returns only odd numbers from the list
    result = []
    for n in numbers:
        if n % 2 == 0:  # odd check
            result.append(n)
    return result

print(filter_even_numbers([1, 2, 3, 4, 5]))
```

COMMENTS

Bad Comments: Outdated & Obsolete

The Dangers of Unmaintained Comments

- Commented-Out Code
  - It clutters the codebase and confuses developers. Use version control to keep track of history instead.
  - Rule: Just delete it

```python
# Bad Commnet: Commented-Out Code (Old Code)
def calculate_total(prices):
    total = sum(prices)
    # total = 0
    # for price in prices:
    #     total += price  # Old implementation
    return total
print(calculate_total([10, 20, 30]))
```

COMMENTS

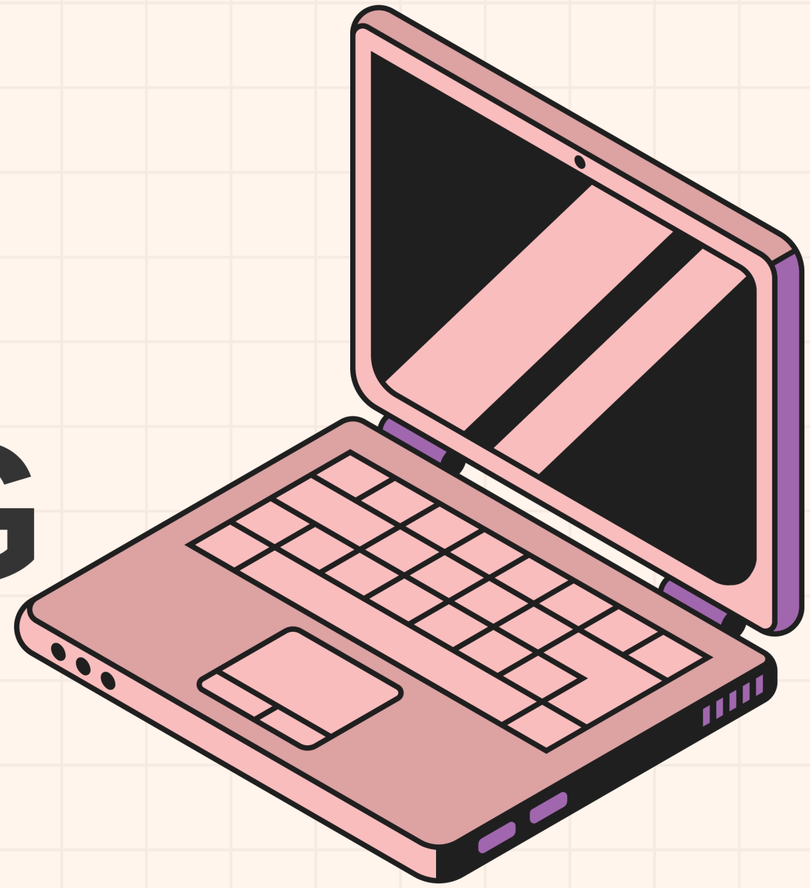
Bad Comments: Outdated & Obsolete

The Dangers of Unmaintained Comments

- History and Attributions in Comments
  - Avoid logging changes or authors in comments. Version control (like Git) is the right tool for this.

```java
public class Calculator {

    // Added by Ahmed on 2022-05-01
    // Fixed bug by Mona on 2022-06-10
    // Modified by Ali on 2023-01-15

    // This method adds two numbers
    public int add(int a, int b) {
        return a + b;
    }
}
```

FORMATTING



FORMATTING

Vertical Formatting

- Keep source files small : Small files (typically under 200 lines, max 500) are much easier to understand and navigate than large ones.
- Use blank lines to separate concepts.

Java

```java
// Before: Cluttered and hard to scan
package com.example;
import java.util.List;
public class MyClass {
private int count;
public void myMethod() {
//...
}
}

// After: Clean and easy on the eyes
package com.example;

import java.util.List;

public class MyClass {
private int count;

public void myMethod() {
//...
}
```

FORMATTING

Vertical Formatting

- Keep related code vertically close (Vertical Distance).
- Explanation: Concepts that are closely related should be near each other in the source file.
- Variables: Declare variables as close as possible to where they are used.
- Functions: If function A calls function B, then A (the caller) should be placed above B (the callee). This creates a natural top-down reading flow.

```
// Good: The calling function is on top
public void processOrder() {
    validateOrder();
    saveOrder();
}

private void validateOrder() { /* ... */ }
private void saveOrder() { /* ... */ }
```

FORMATTING

Horizontal Formatting

- Keep lines short : Programmers overwhelmingly prefer short lines. The old 80-character limit is a good guideline.
- Use horizontal whitespace to improve clarity : use spaces to associate things that are strongly related and to separate things that are less related. This is especially useful around operators to show precedence

Java

```java
// Before: Hard to see the distinct parts of the expression
int result=a*b+c/d-e;

// After: Whitespace clarifies operator precedence
int result = (a * b) + (c / d) - e;
```

FORMATTING

Horizontal Formatting

- Indentation is non-negotiable : Indentation makes the hierarchical structure of the code visible. Without it, code is virtually unreadable. Always follow indentation rules, even for single-line if or while statements.

Java

```java
// Good: The structure is immediately obvious
public void myMethod() {
    if (condition) {
        doSomething();
    }
}
```

FORMATTING

Common Mistakes to Avoid

- Ignoring Team Formatting Rules: The biggest mistake is having multiple coding styles in one project. The team must agree on a single set of formatting rules and apply them consistently
- Misleading Horizontal Alignment : Trying to line up variable types, names, or assignment values in a long list. It draws attention away from the true intent of the code and is hard to maintain.

Java

```java
private String  name;
private int     age;
private boolean isActive;
```

FORMATTING

Common Mistakes to Avoid

- Breaking Indentation for Short Scopes: it is Putting an if statement and its body on a single line. It hides the code's structure and makes the scope of the statement difficult to see, which can lead to bugs.

Java

```java
if (x > 5) doSomething(); // Avoid this
```

FORMATTING

Common Mistakes to Avoid

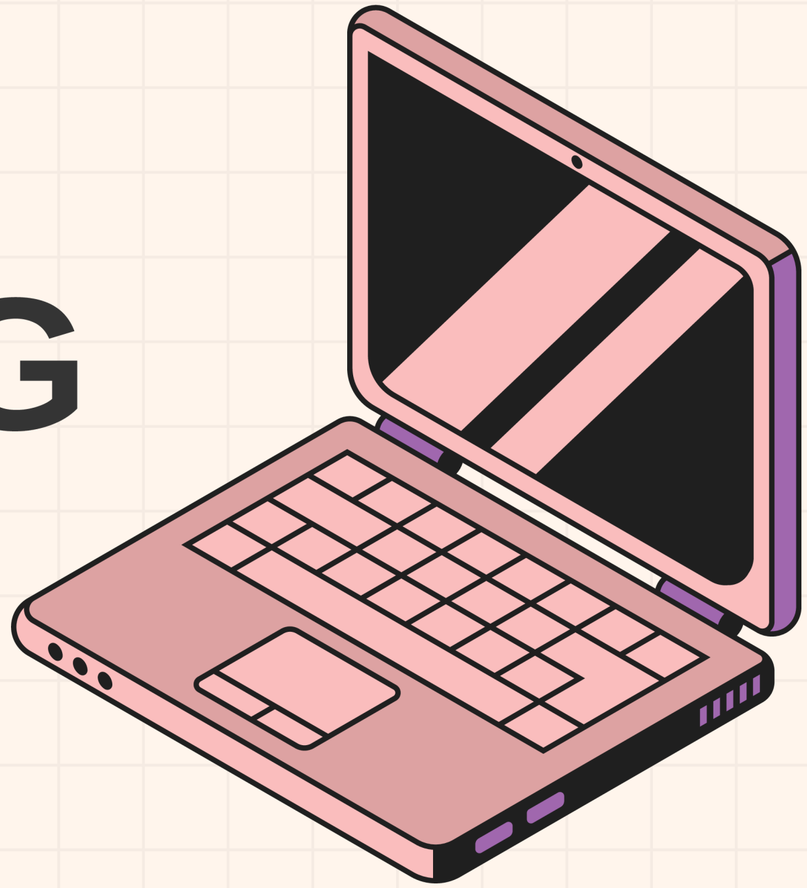
Obscure "Dummy Scopes":

- What it is: A while or for loop with an empty body, like while (condition);.
- Why it's bad: The semicolon at the end of the line is nearly invisible and can cause serious, hard-to-find bugs.
- The Fix: If you can't avoid it, put the semicolon on its own indented line to make it obvious.

Java

```
// Good and clear
while (dis.read(buf, 0, readBufferSize) != -1)
    ;
```

REFACTORING



REFACTORING

First Make it work

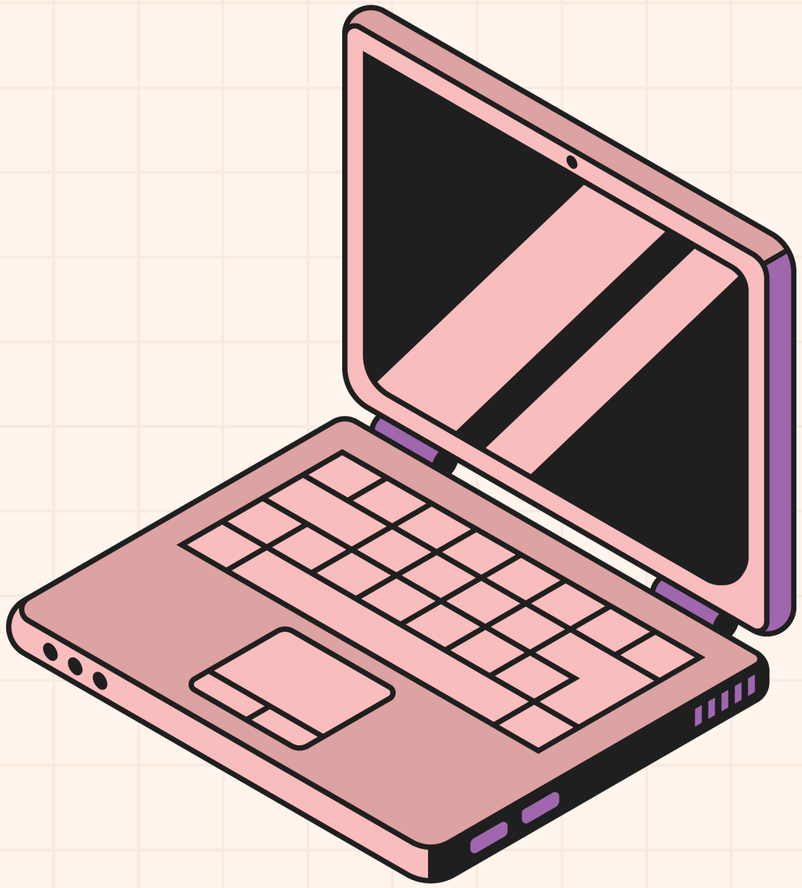
- First use unit test and ensure testing all or at least most of the code
- Fix all errors and mistakes making the code works passing all of the tests

REFACTORING

Then make it right

- Then you can start cleaning the code making it more clear
- apply Boy Scout Rule : leave the code better than you found it
- Test the code with every change to make sure it still works

SMELL & HEURISTIC



CODE SMELLS

Comments

- inappropriate information : comments should be for technical notes about code
- obslete, redundant and poorly written comments.
- commented out code.

CODE SMELLS

Environment

- build and tests that require more than one step

Functions

- too many arguments
- output & flag arguments
- dead function : a never called function

CODE SMELLS

General

- multiple languages in one source file
- obvious behavior not implemented :
- not following "The Principle of least surprise"
- incorrect behavior at boundaries
- overriden safties
- duplication : DRY principle(don't repeat yourself)
- Dead code : is code that isn’t executed
- Vertical Separation : Variables and function should be defined close to where they are used. Local variablesshould be declared just above their first usage and should have a small vertical scope. Private functions should be defined just below their first usage.
- Inconsistency
- Clutter

CODE SMELLS & HEURISTIC

Java

- Avoid Long Import Lists by Using Wildcards
- don't inherit constants : use a static import instead
- Constants vs Enums

CODE SMELLS & HEURISTIC

Names

- choose describtive name
- Names at appropriate level of abstraction
- Use Standard Nomenclature Where Possible
- unambiguous Names
- use long Names for long scopes
- avoid encoding
- Names Should Describe Side-Effects

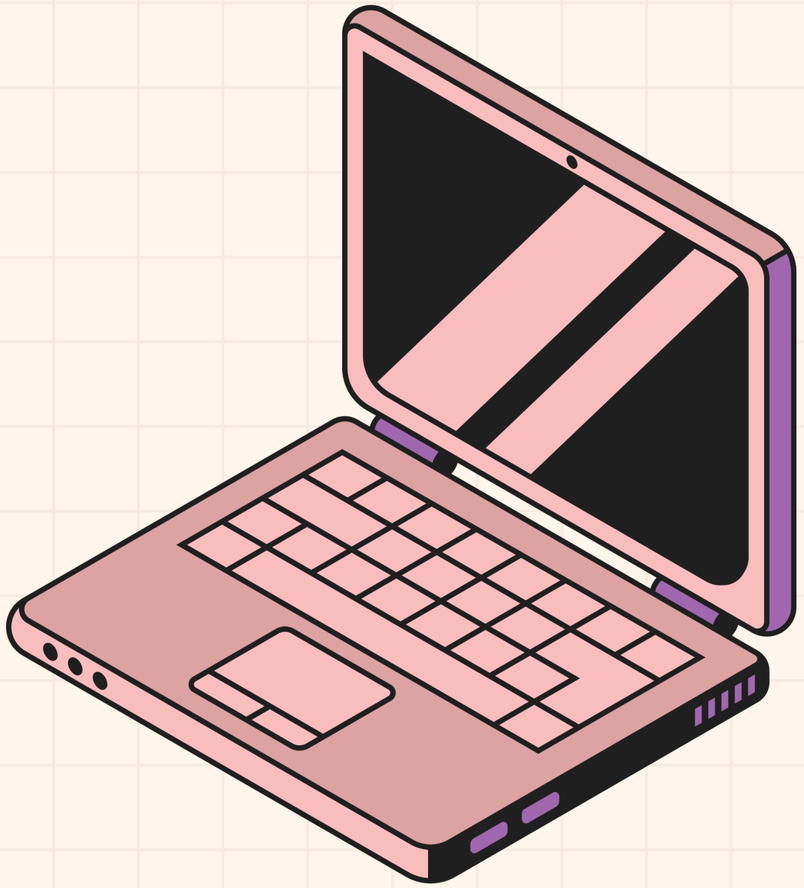
CODE SMELLS & HEURISTIC

Tests

- insufficient tests
- use a coverage tool : Coverage tools reports gaps in your testing strategy , finding modules, classes, and functions that are insufficiently tested
- Don’t Skip Trivial Tests
- Test Boundary Conditions
- Exhaustively Test Near Bugs
- Tests Should Be Fast

TAKEAWAYS

ACTIONABLE HABITS FOR WRITING CLEAN CODE



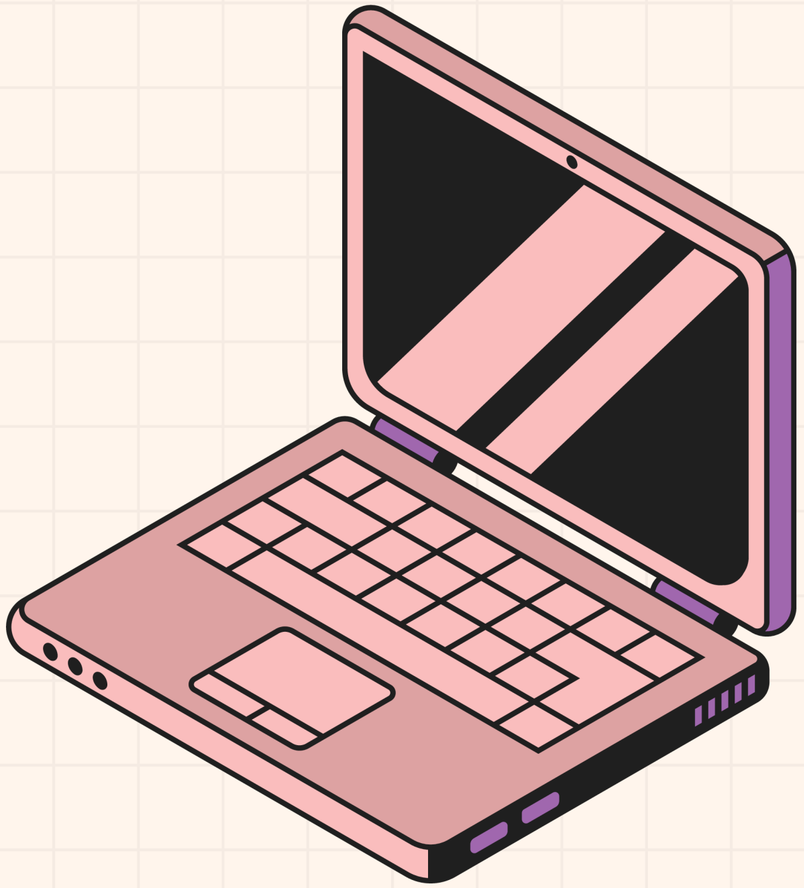
TAKEAWAYS
actionable habits

- ***Name with care***
- ***Keep functions small*** *: One task, one responsibility.*
- ***Consistent formatting*** *: Choose a style guide (PEP8, Airbnb JS, etc.) and stick to it religiously.*
- ***Avoid duplication****: "Don't Repeat Yourself"*
- ***Limit cleverness****: If you're proud of how tricky the code is, it probably won't be readable next week.*
- ***Write meaningful comments****: Explain why something exists, not what the code already says.*

TAKEAWAYS
actionable habits

- ***Refactor often***
- ***Use version control wisely*** *: Small, focused commits with clear messages are part of clean code too.*
- ***Design for tests****: If something is hard to test, it's often poorly structured. Tests force clarity.*
- ***Mind dependencies****: Import only what you need; keep boundaries clear between modules.*
- ***Automate checks****: Linters (ESLint, Flake8), formatters (Prettier, Black), type checkers (TypeScript, mypy).*
- ***Leave code better than you found it****: Even one renamed variable can reduce future confusion.*

FURTHER READING

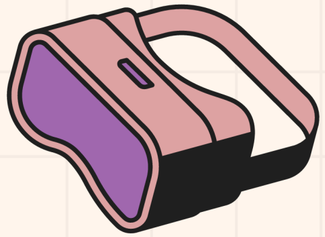


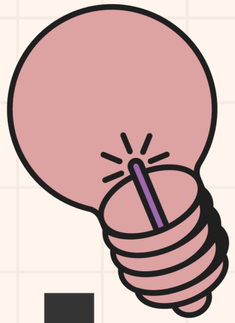
FURTHER READING

Resources

- *Clean Code: A Handbook of Agile Software Craftsmanship by Robert C. Martin.*

- *Refactoring: Improving the Design of Existing Code by Martin Fowler with contributions of Kent Beck*





THANK YOU