Grow in

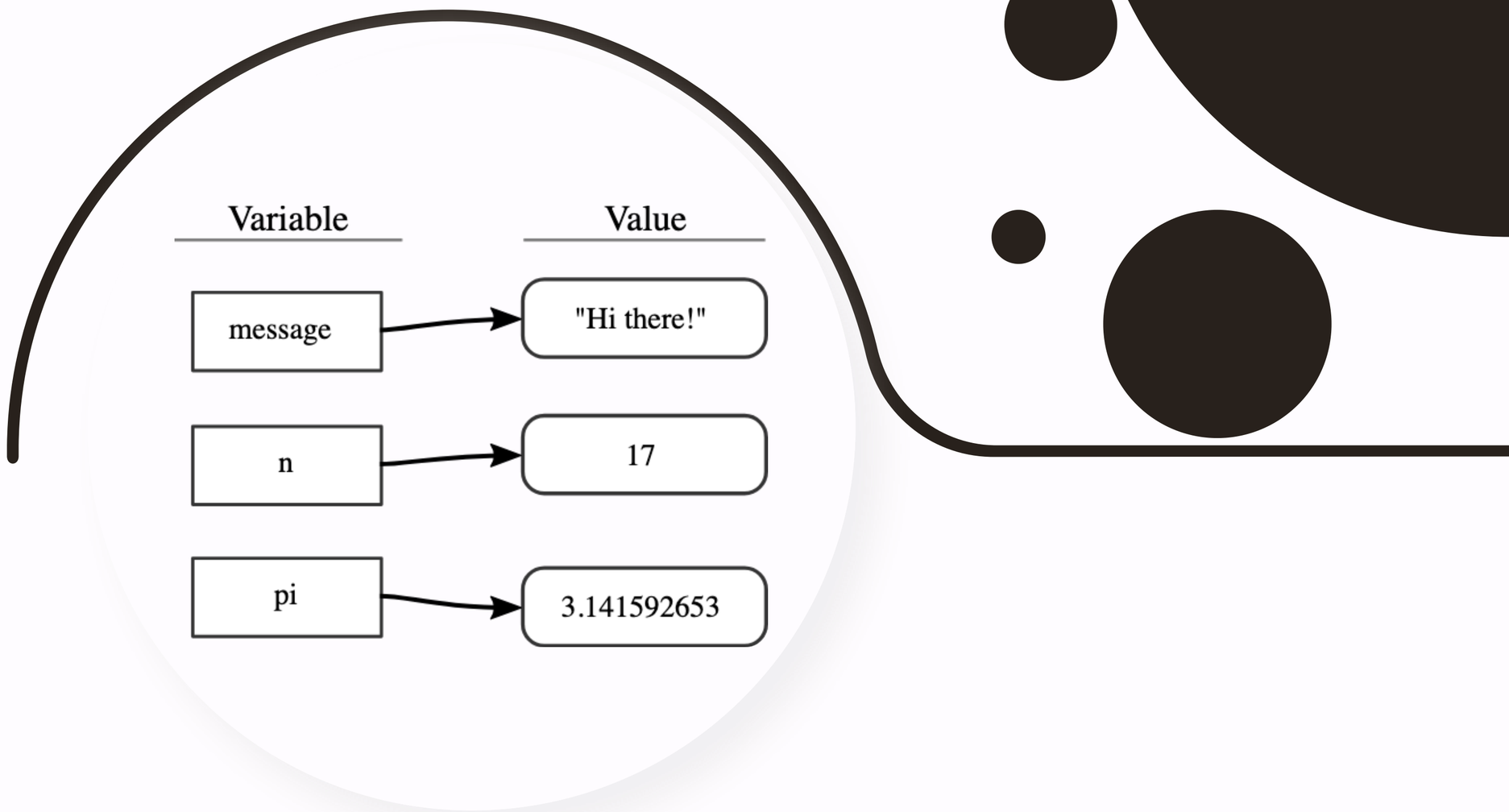
Software Engineering

Clean Code Practices

@tauseeffayyaz

Code Readability & Simplicity	Keep things simple to reduce complexity and improve collaboration.
Function & Class Design	Keep functions small and classes focused on a single job.
Testing & Maintainability	Write meaningful, maintainable tests to validate core logic and prevent regressions.
Code Structure & Architecture	Keep dependencies clean and avoid unnecessary complexity.
Refactoring & Iteration	Keep the codebase clean, DRY, and adaptable.
Robustness & Safety	Handle failure gracefully and reduce the impact of errors.
Documentation & Comments	Use documentation to support understanding and preserve context.
Tooling & Automation	Let tools enforce quality, consistency, and safety automatically, so you can focus on logic and design.
Final Review Practices	Before merging, make sure the code is clean, reviewed, and understandable.

@tauseeffayyaz



Meaningful Names

Use descriptive names for variables, functions, and classes to convey intent. Avoid abbreviations, misleading or generic terms. Clear naming reduces the need for comments.

Improves clarity and understanding

@tauseeffayyaz

```
public void add(Object element) {
  if (!readOnly) {
    int newSize = size + 1;
    if (newSize > elements.length) {
      Object[] newElements =
        new Object[elements.length + 10];
      for (int i = 0; i < size; i++)
        newElements[i] = elements[i];
      elements = newElements;
    }
    elements[size++] = element;
  }
}
```

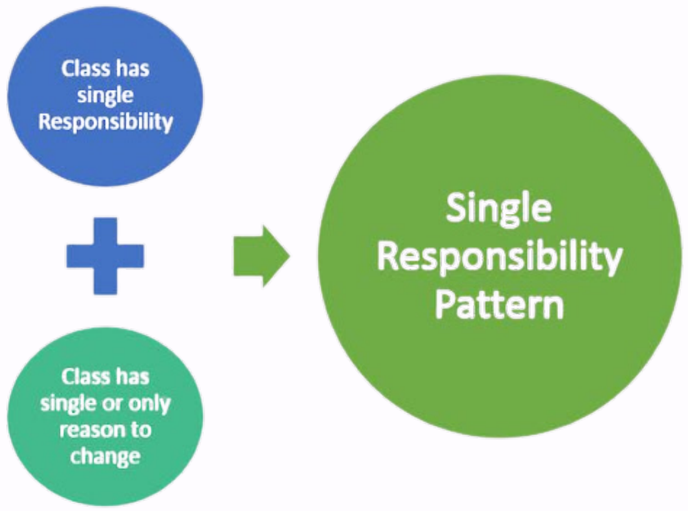
```
public void add(Object element) {
  if (readOnly)
    return;
  if (atCapacity())
    grow();
  addElement(element);
}
```

Short Functions

Split large functions into smaller, single-purpose ones. This enhances readability and simplifies testing.

Easier to read and test

@tauseeffayyaz



Single Responsibility Principle

Each module, class, or function should do one thing well and have only one reason to change.

Encourages modular design

@tauseeffayyaz

```php
function register()
{
    if (!empty($_POST)) {
        $msg     ';
        if ($_POST['user_name']) {
            if ($_POST['user_password_new']) {
                if ($_POST['user_password_new'] === $_POST['user_password_repeat']) {
                    if (strlen($_POST['user_password_new']) > 5) {
                        if (strlen($_POST['user_name']) < 65 && strlen($_POST['user_name']) > 1) {
                            if (preg_match('/^[a-z\d]{2,64}$/i', $_POST['user_name'])) {
                                $user = read_user($_POST['user_name']);
                                if (!isset($user['user_name'])) {
                                    if ($_POST['user_email']) {
                                        if (strlen($_POST['user_email']) < 65) {
                                            if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                                                create_user();
                                                $_SESSION['msg'] = 'You are now registered so please login';
                                                header('Location:    . $_SERVER['PHP_SELF']);
                                                exit();
                                            } else $msg = 'You must provide a valid email address';
                                        } else $msg = 'Email must be less than 64 characters';
                                    } else $msg = 'Email cannot be empty';
                                } else $msg = 'Username already exists';
                            } else $msg = 'Username must be only a-z, A-Z, 0-9';
                        } else $msg = 'Username must be between 2 and 64 characters';
                    } else $msg = 'Password must be at least 6 characters';
                } else $msg    'Passwords do not match';
            } else $msg = 'Empty Password';
        } else $msg    'Empty Username';
        $_SESSION['msg'] = $msg;
    }
    return register_form();
}
```

Avoid Deep Nesting

Flatten complex conditional blocks using early returns or guard clauses. It improves linear readability.

Reduces mental overhead

@tauseeffayyaz

```css
.table-bordered { border: 1px solid #ddd; }

.table-bordered > thead > tr > th, .table-bordered > thead > tr > td, .table-bordered > tbody > tr > th, .
table-bordered > tbody > tr > td, .table-bordered > tfoot > tr > th, .table-bordered > tfoot > tr > td { border:
1px solid #ddd; }

.table-bordered > thead > tr > th, .table-bordered > thead > tr > td { border-bottom-width: 2px; }

.table-striped > tbody > tr:nth-of-type(odd) { background-color: #f9f9f9; }

.table-hover > tbody > tr:hover { background-color: #f5f5f5; }
```

```css
.table-bordered {
  border: 1px solid #ddd;
}

.table-bordered > thead > tr > th,
.table-bordered > thead > tr > td,
.table-bordered > tbody > tr > th,
.table-bordered > tbody > tr > td,
.table-bordered > tfoot > tr > th,
.table-bordered > tfoot > tr > td {
  border: 1px solid #ddd;
}

.table-bordered > thead > tr > th,
.table-bordered > thead > tr > td {
  border-bottom-width: 2px;
}

.table-striped > tbody > tr:nth-of-type(odd) {
  background-color: #f9f9f9;
}

.table-hover > tbody > tr:hover {
  background-color: #f5f5f5;
}
```

Consistent Formatting

Use a consistent code style for spacing, indentation, and brackets. Automate formatting with tools.

Uniform appearance aids scanning

@tauseeffayyaz

(a) Code under development or for debugging

```php
<?php
if (false) {
  print_r($variable);
}
if (0) {
  // Huge piece of code with bug
  // or unfinished new feature
}
?>
```

(b) Unreachable code due to return statement

```php
<?php
function foo($bar) {
  $bar *= 2;
  return $bar;
  $bar += 1;
  return $bar;
}
?>
```

(c) Variable assigned but never used

```php
<?php
function foo() {
  $a = "Hello world";
  //more code
  $a = "Hello universe";
  //more code
  $a .= " and the world";
}
?>
```

(d) Partially dead code: dead on some program paths

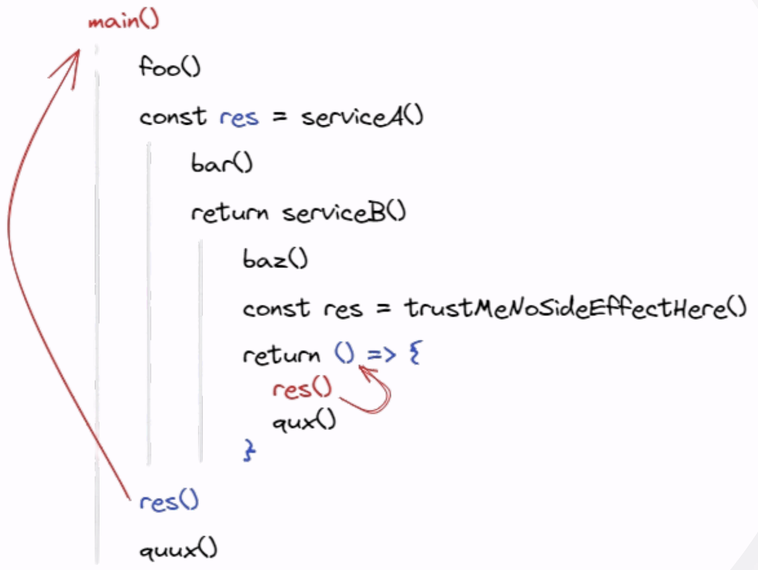
```php
<?php
$a = 1;
if (foo()) {
  $a = 10;
}
else {
  $a *= 2;
}
$a += 1;
?>
```

Remove Dead Code

Eliminate unused variables, commented code, and unnecessary imports. Keeps the codebase clean and lean.

Prevents confusion and clutter

@tauseeffayyaz

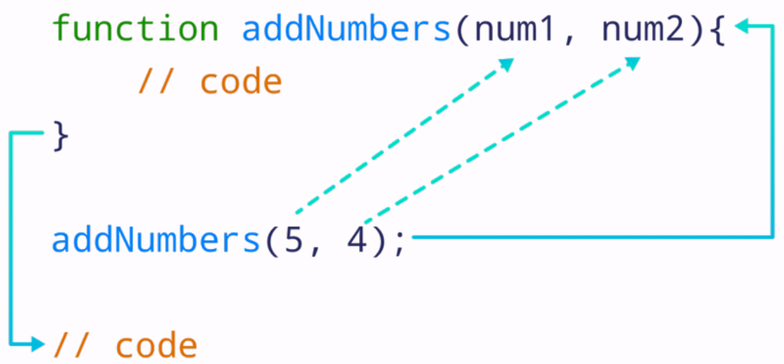


Minimal Side Effects

Functions should not unexpectedly change external state. Keep them pure unless intended.

Promotes predictable behavior

@tauseeffayyaz



Function Arguments (<=3)

Limit parameters to avoid complexity. Use objects to group related data and enhance readability.

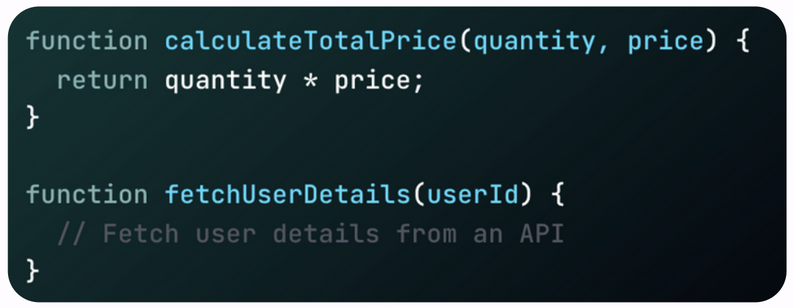
Simpler and cleaner signatures

@tauseeffayyaz

Pure Functions

Functions should produce the same output for the same input without modifying external state.

Reliable and test-friendly



Descriptive Function Names

Use names that describe what the function does (e.g., getUser, calculateTax). Avoid vague verbs.

Self-explanatory code

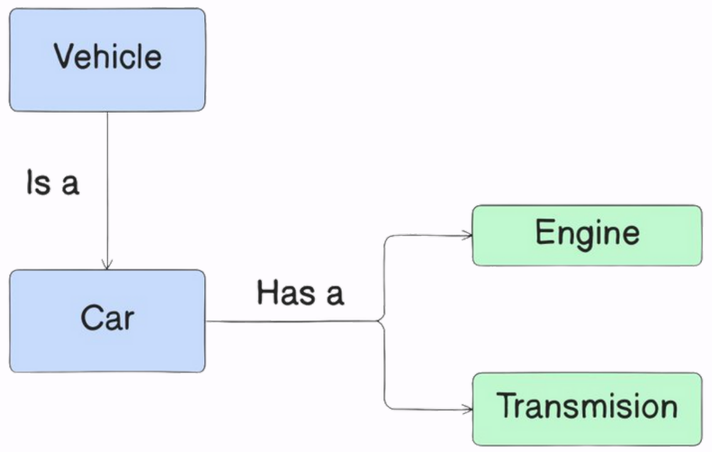
```java
public class Department
{
    private String departmentName;
    private Date startingDate;
    public Department(String departmentName)
    {
        this.departmentName = departmentName;
        this.startingDate = new Date();
    }

    public String getDepartmentName()
    {
        return this.departmentName;
    }

    public Date getStartingDate()
    {
        return this.startingDate;
    }
}
```

Keep Classes Small

Each class should do one thing and avoid bloating with unrelated logic or data.

Promotes reuse and testability

@tauseeffayyaz

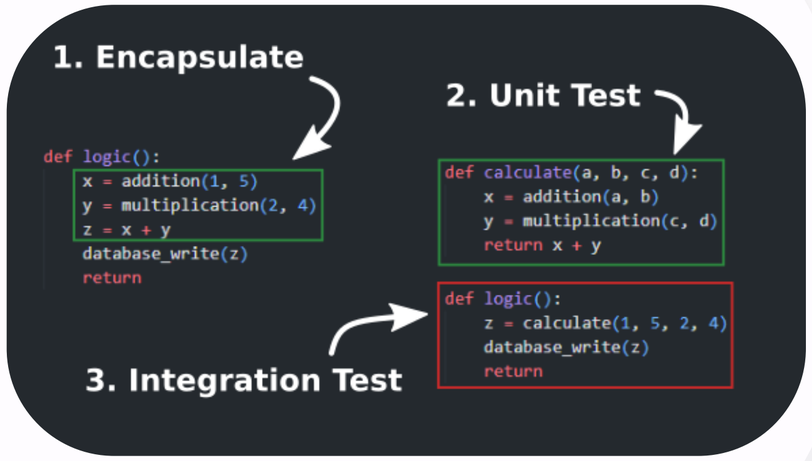


Composition Over Inheritance

Reuse behavior by combining smaller objects rather than using deep inheritance trees.

Enhances flexibility and control

@tauseeffayyaz



Write Unit Tests For Logic

Test core logic in isolation to catch bugs early and enable confident refactoring.

Validates functionality

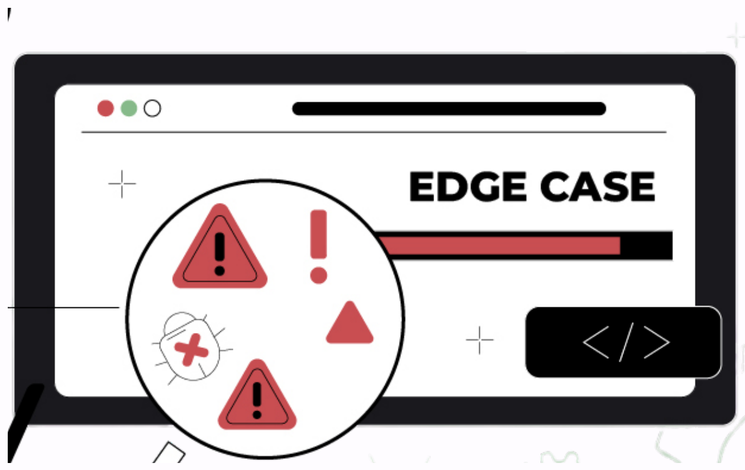
@tauseeffayyaz

Avoid Over-Mocking

Mock only what you must. Prefer real implementations in integration and end-to-end tests.

Ensures realistic testing

@tauseeffayyaz

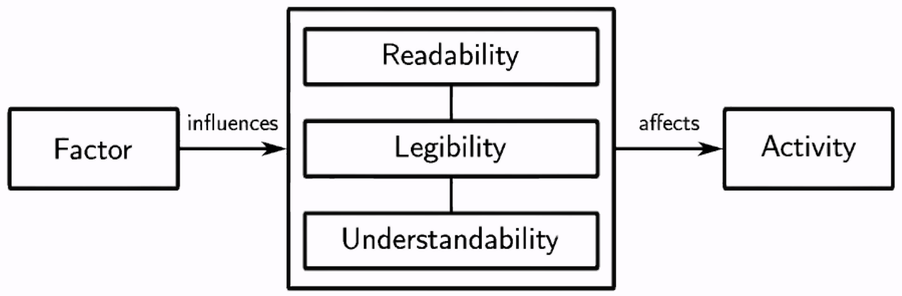


Test Edge Cases

Go beyond the happy path. Handle invalid inputs, boundaries, and rare conditions.

Increases code reliability

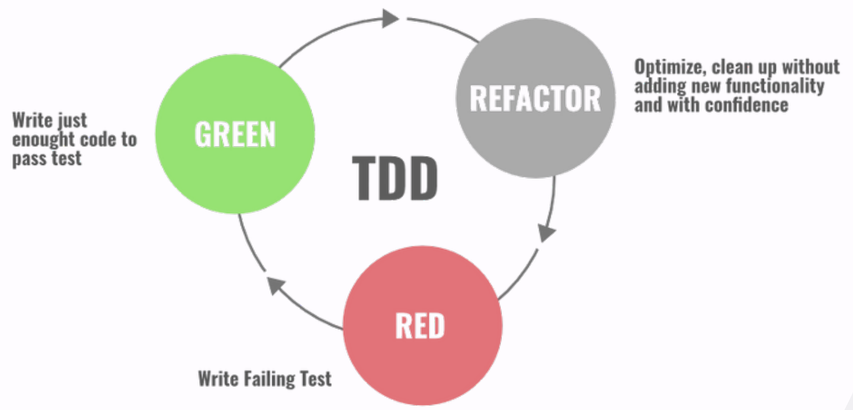
@tauseeffayyaz



Keep Tests Readable

Tests should tell a clear story. Use descriptive names and structure for readability.

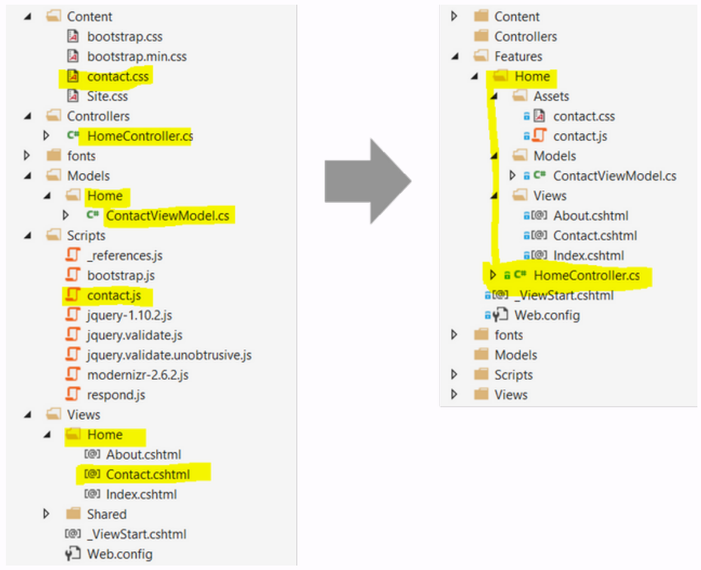
Easier to maintain



Refactor with Tests

Refactor code incrementally and rely on tests to verify behavior before and after.

Safe and reliable changes

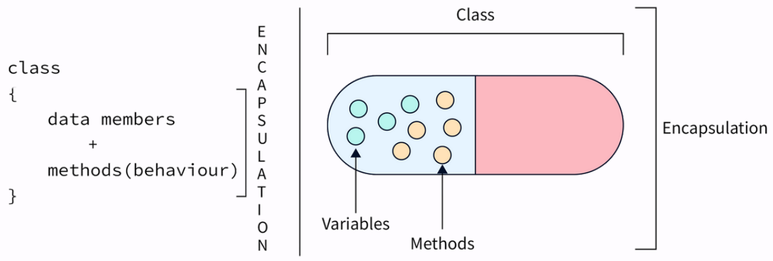


Group by Feature, Not Type

Structure folders and files by domain features instead of technical types (e.g., utils/services).

Improves project navigation

@tauseeffayyaz

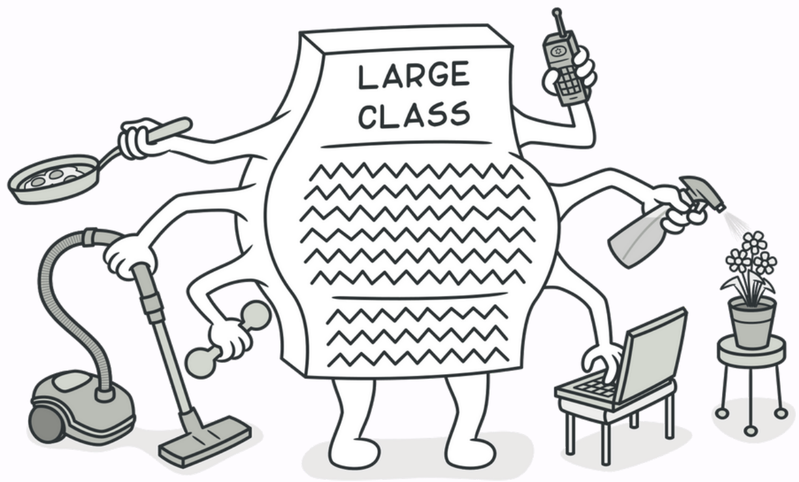


Encapsulate Logic

Hide internal details and avoid leaking implementation through public interfaces.

Reduces coupling

@tauseeffayyaz

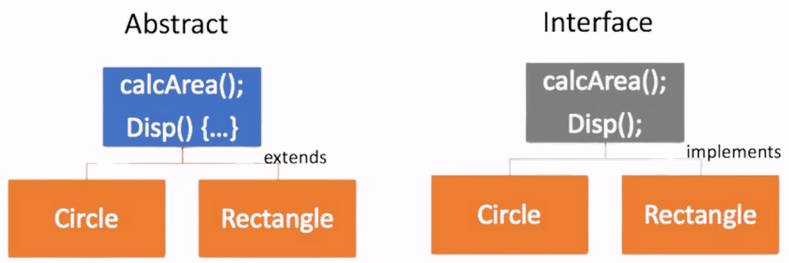


Avoid Large Objects

Break down large, all-knowing classes into smaller domain-specific ones.

Improves maintainability

@tauseeffayyaz



Use Interface Abstractions

Expose only what's needed. Abstract dependencies for flexibility and testability.

Enables decoupling

@tauseeffayyaz

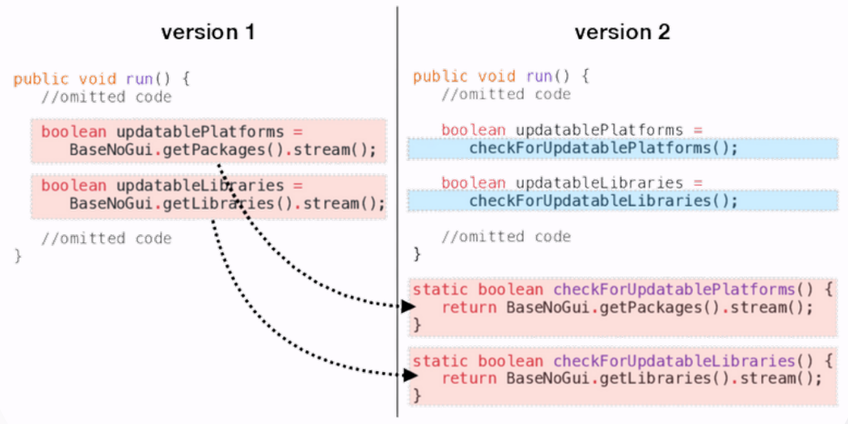
```java
public class ScopeExample {

    static double x = 0.5;

    public static void main (String[] argv)
    {
        System.out.println (x);
        squareIt ();
        System.out.println (x);
    }

    static void squareIt ()
    {
        x = x * x;
    }

}
```

The variable x can be accessed by any method in the class

Minimize Global State

Avoid shared mutable state. Use scoped, controlled data to prevent unexpected behavior.

Reduces bugs



Refactor Ruthlessly

Make code better continuously without changing behavior. Simplify, rename, and split responsibilities.

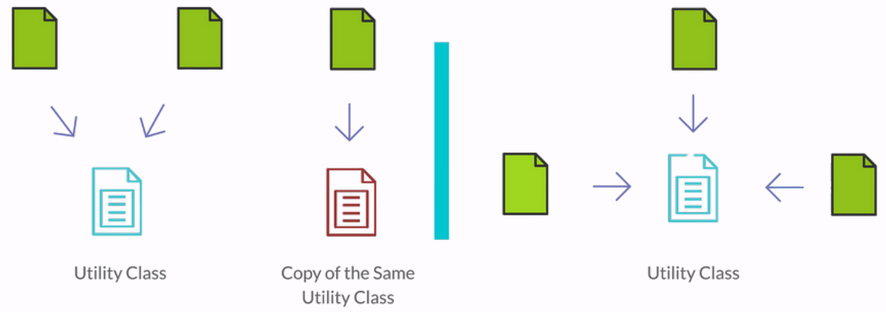
Keeps code healthy

Follow Boy Scout Rule

Always leave code cleaner than you found it, even in small ways.

Promotes collective care

@tauseeffayyaz

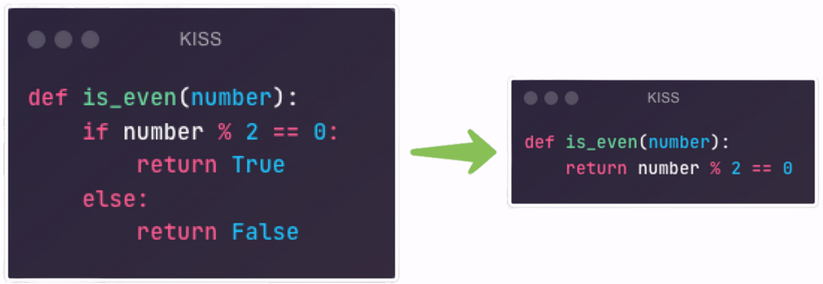


Don't Repeat Yourself (DRY)

Extract repeated logic into functions or shared modules. Duplication breeds maintenance headaches.

Avoids redundancy

@tauseeffayyaz

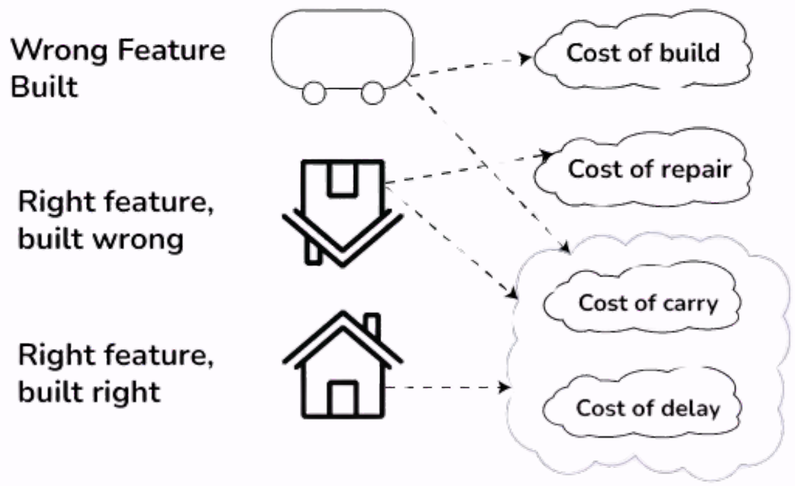


Keep It Simple (KISS)

Start with the simplest working solution. Add complexity only when necessary.

Prevents over-engineering

@tauseeffayyaz



You Aren't Gonna Need It (YAGNI)

Don't write code for future hypothetical needs. Build when required.

Saves time and effort

@tauseeffayyaz

```
class BadConditionals{
    async execute(email: string){
        if(!email){
            throw new Error("Empty e-mail");
        }else if (email == 'teste@teste.com'){
            throw new Error("Bad e-mail");
        }else{
            return 'Valid e-mail'
        }
    }
}
```

Fail Fast

Validate inputs early and throw errors where appropriate. Don't let bad data spread.

Detects issues early

```python
numerator = 10
denominator = 0

try:
    result = numerator / denominator
    print(result)
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")
```

Graceful Error Handling

Catch exceptions with meaningful messages. Don't crash unnecessarily.

Improves UX

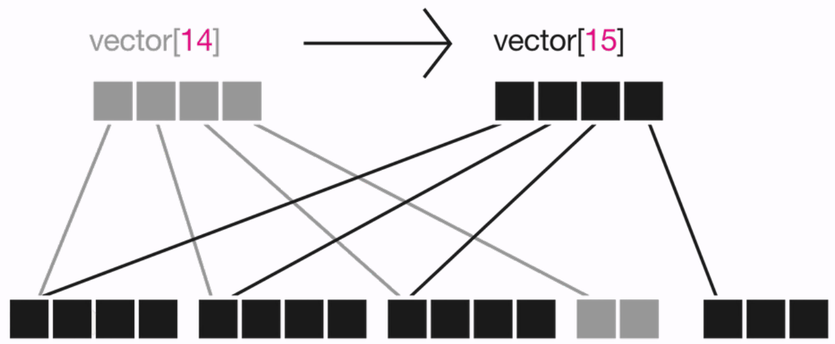
@tauseeffayyaz

Avoid Magic Numbers/Strings

Use named constants or enums for better meaning, consistency, and refactoring.

Enhances readability

@tauseeffayyaz



Immutable Data Structures

Favor immutability to prevent accidental changes and side effects.

Improves Safety

```python
# oh my, what on earth does this function do?
def my_function(foo, bar, baz):
    x = []
    if bar:
        z = foo.split()
        for i in z:
            if i != baz:
                x.append(i)
        return x
```

Self-Documenting Code

Use clear names and structure so others can understand without comments.

Reduces need for extra docs

@tauseeffayyaz

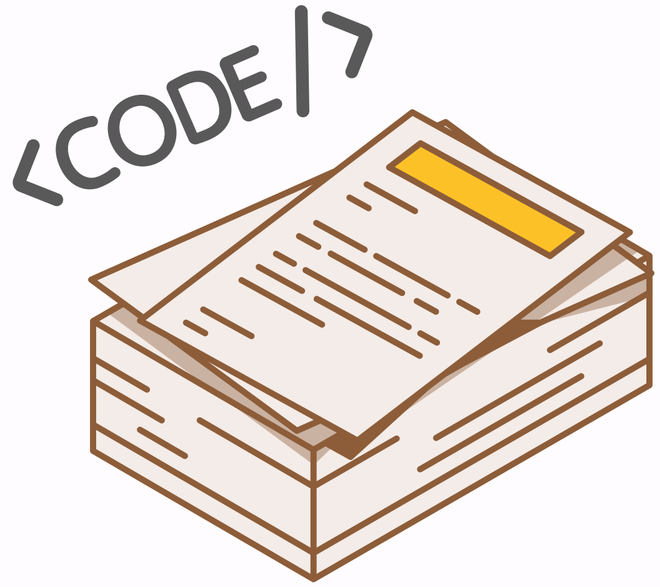
```
const age = 21;             // Specify the age of the user
const name = "Bob";         // Set the name of the user
const yearsLeft = 84 - 21; // years of life left
const bananas = 'yum';      // I like bananas
```

Comment Why, Not What

Only explain intent, reasoning, or non-obvious decisions, not what is already evident.

Preserves logic context

@tauseeffayyaz

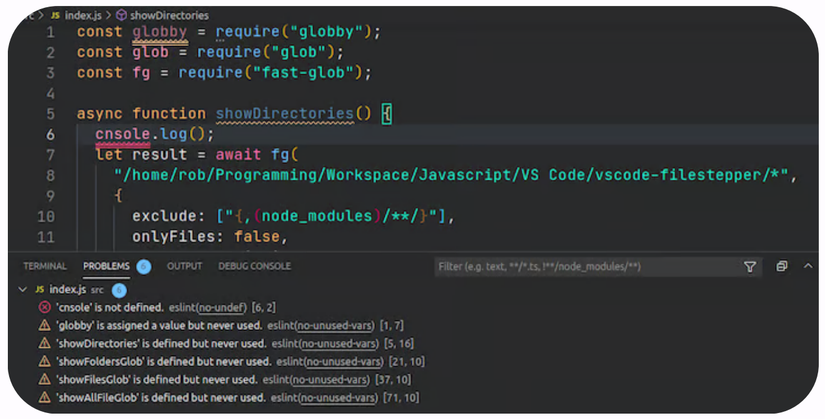


Keep Docs Close to Code

Use inline docstrings, annotations, or tool-supported formats to describe usage and purpose.

Accessible explanations

@tauseeffayyaz

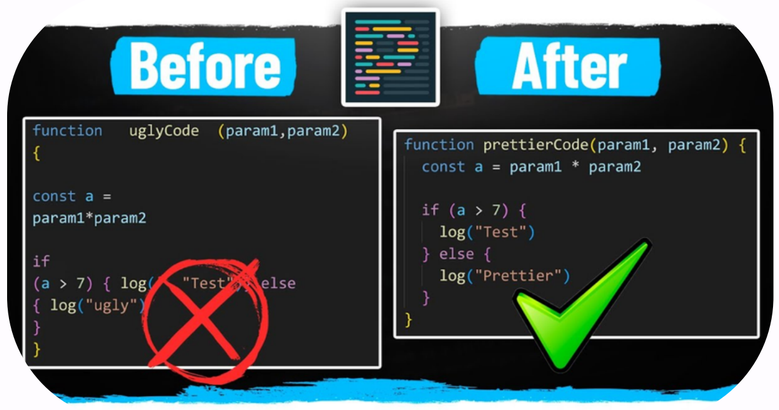


Use Linters

Catch formatting issues, code smells, and errors with automated linters.

Enforces code standards

@tauseeffayyaz

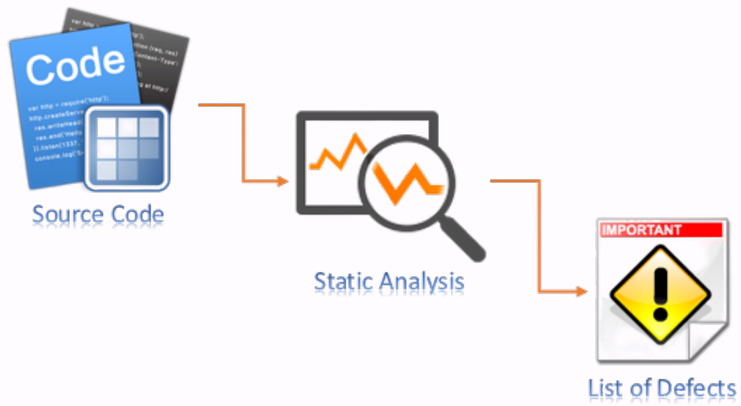


Use Formatters (Prettier/Black)

Automatically format code to keep a uniform, readable style across the team.

Saves manual formatting

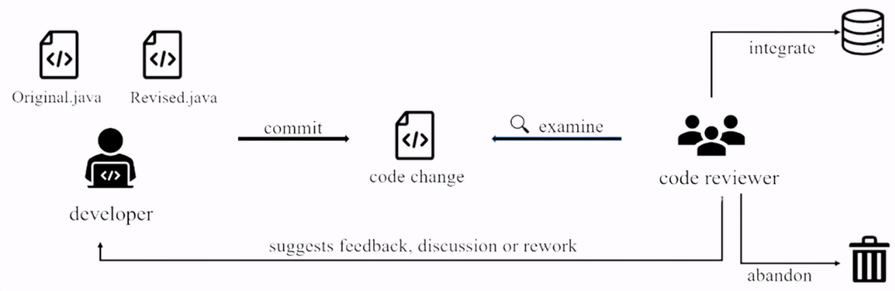
@tauseeffayyaz



Use Static Analysis Tools

Analyze code for bugs, complexity, and bad practices before runtime.

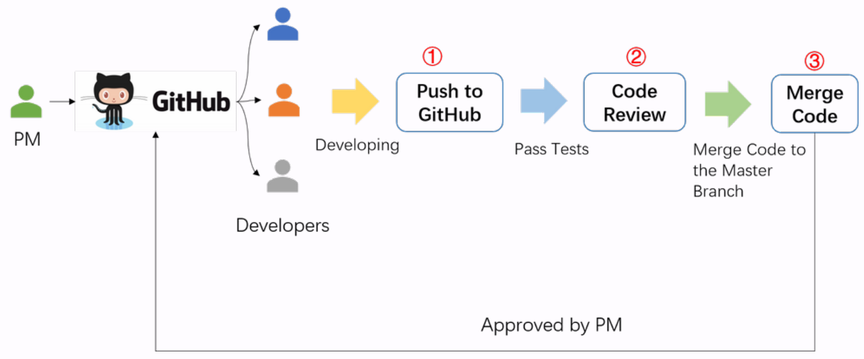
Prevents deeper issues



Automate Code Reviews

Use CI tools to enforce code quality rules on each pull request.

Scales review process

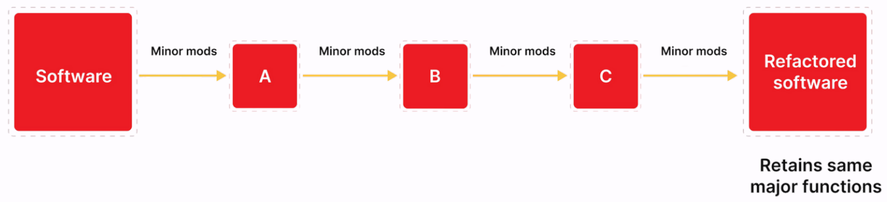


Code Review Before Merge

Always have peers review code for quality, logic, and standards before merging.

Catches issues early

@tauseeffayyaz

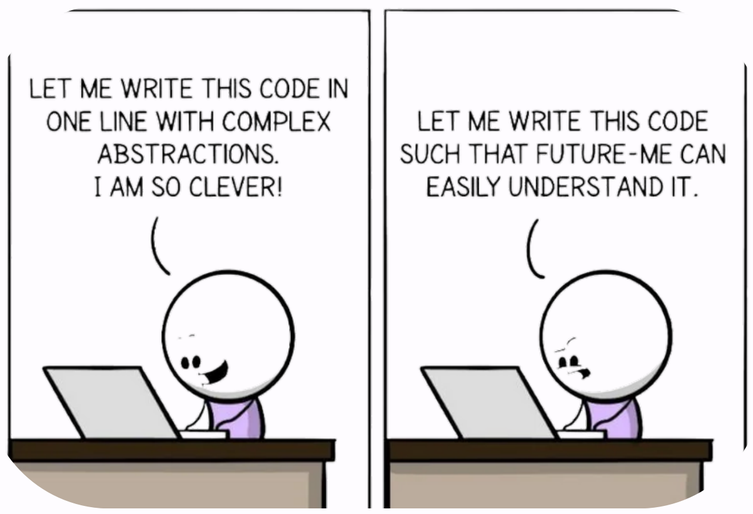


Refactor Before You Add

Improve nearby code before adding new logic to avoid compounding tech debt.

Keeps code clean

@tauseeffayyaz



Avoid Clever Code

Write code that's obvious and easy to follow, even if it's longer.

Clarity over complexity

@tauseeffayyaz

About Tauseef Fayyaz

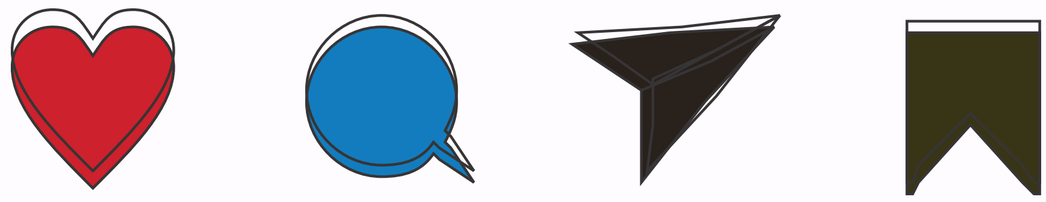
Full-Stack Engineer, Ranked #1 in Computer Engineering (Pakistan), Career Mentor for Aspiring Engineers, Tech Content Creator

---

You can request a dedicated mentorship session with me for guidance, support, or just honest career advice.

⟶ **https://topmate.io/tauseeffayyaz/page/ofsztX71UK**

Your support keeps me motivated.



@tauseeffayyaz