SOME BEST PRACTICES & REFACTORING TECHNIQUES

SE3070 - CASE STUDIES IN SOFTWARE ENGINEERING

SEMESTER 2, 2025

SLIIT

LEARNING OUTCOMES

- After completing this topic, you will be able to,
  - Explain the relationship between coding standards, best practices, and code quality.
  - Apply best practices to improve maintainability and readability.
  - Identify code smells in a given codebase and propose appropriate refactoring techniques.
  - Use refactoring to improve cohesion, reduce coupling, and align with SOLID principles.

C O N T E N T S

- Coding Standards and Best Practices
- Why Best Practices & Coding Standards Aren’t Enough
- From Code Smells to Refactoring
- Refactoring Fundamentals
- Some examples
- When Not to Refactor
- Summary

WHY CODE QUALITY MATTERS

- Poor code quality leads to technical debt.
- Design quality ≠ Implementation quality
  - Even great designs can be ruined by bad code.
- Good code is easier to understand, modify, and extend without breaking existing features.

CODING STANDARDS

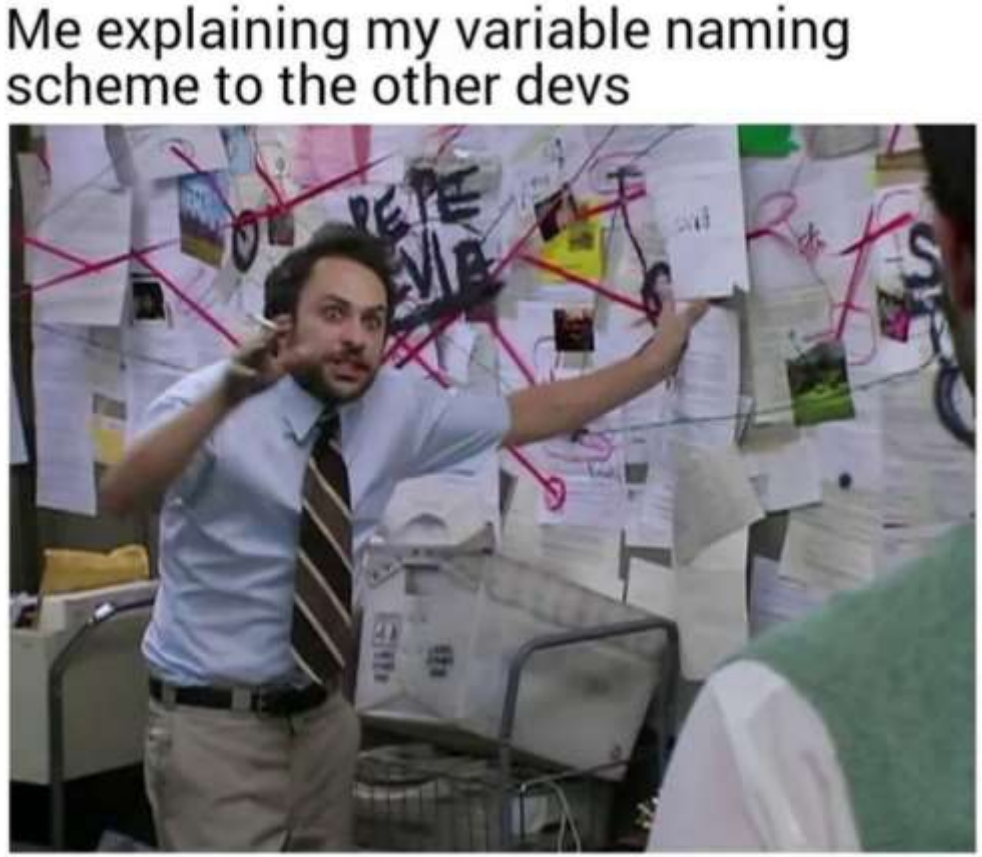
Source: https://programmerhumor.io/programming-memes/i-regret-not-commenting-enough-in-my-early-years-of-cs-its-just-a-habit-of-mine-now/

CODING STANDARDS

- Consistent naming conventions for classes, methods, variables, constants etc.
- Package/module structuring that matches system architecture.
- Consistent formatting
  - indentation, braces, whitespace, line wrapping etc.
- Documentation & comments
  - Explain “why”, not just “what”

CODING STANDARDS

- Google C++ Style Guide
- Google Java Style Guide
- Google Python Style Guide



BEST PRACTICES FOR IMPLEMENTATION

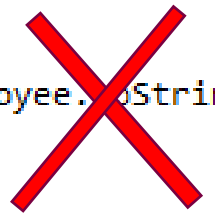
- Logging
  - Use appropriate log levels, include context.
- Exception Handling
  - Don’t swallow exceptions, use meaningful messages.
- Use of Constants & Configuration
  - Replace magic numbers
  - Store in properties/env variables.
- Avoid Deprecated APIs
  - Always prefer supported alternatives
- Organize Code Effectively

BEST PRACTICES FOR IMPLEMENTATION

- Logging: Use appropriate log levels, include context.
- Why?
  - Structured, levelled, timestamped records for troubleshooting
  - configurable per environment
  - supports aggregation/monitoring (E.g. ELK Stack)

```
// Bad practice
System.out.println("User login failed for id=" + userId);

// Good practice
logger.warn("User login failed for id={}", userId);
```

BEST PRACTICES FOR IMPLEMENTATION



```
System.out.println(employee.toString() + "\n");
```

```
/** Initialize logger */
public static final Logger log = Logger.getLogger(AbstractService.class.getName());
```

```
log.info(employee.toString() + "\n");
```

```
} catch (NumberFormatException e) {
    log.log(Level.SEVERE, e.getMessage());
} catch (XPathExpressionException e) {
    log.log(Level.SEVERE, e.getMessage());
} catch (SAXException e) {
    log.log(Level.SEVERE, e.getMessage());
} catch (IOException e) {
    log.log(Level.SEVERE, e.getMessage());
} catch (ParserConfigurationException e) {
    log.log(Level.SEVERE, e.getMessage());
}
```

BEST PRACTICES FOR IMPLEMENTATION

- Exception Handling: Don’t just throw exceptions, handle them properly.

Not Good

```
try {
    process(order);
} catch (Exception e) {
    // ignore
}
```

```
try {
    process(order);
} catch (PaymentException e) {
    log.error("Payment failed for orderId={}", order.getId(), e);
    notifyUser(order.getCustomer(), "Payment could not be processed.");
    rollbackTransaction(order);
    // handled gracefully without crashing the system
}
```

Good

BEST PRACTICES FOR IMPLEMENTATION

- Use of Constants & Configuration

```
double total = amount + amount * 0.12; // What is 0.12?
String url = "http://dev.api.local:8080"; // hard-coded env
```

- This is bad because,
  - Magic Numbers – arbitrary numbers with no meaning ("0.12")
  - scatters configuration across code
    - makes updates error-prone
    - prevents environment-specific overrides

BEST PRACTICES FOR IMPLEMENTATION

```
static final double VAT_RATE = 0.12;
static final String PAYMENT_BASEURL_KEY = "services.payment.baseUrl";
static final String CONFIG_FILEPATH = "Src\\Config.properties";
```

Constants

```
double total = amount + amount * VAT_RATE;
String paymentBaseUrl = loadConfigValue(PAYMENT_BASEURL_KEY);
```

Better Example in code

```
public static String loadConfigValue(String key) {
    Properties props = new Properties();
    FileInputStream fis = null;
    try {
        fis = new FileInputStream(CONFIG_FILEPATH);
        props.load(fis);
        return props.getProperty(key);
    } catch (IOException e) {
        System.err.println("Could not load configuration: " +
                           e.getMessage());
        return null; // or take a fallback action
    } finally {
        if (fis != null) {
            try { fis.close(); } catch (IOException e) {
              /* log or ignore */ }
        }
    }
```

Method for getting the Properties via the Config file

```
services.payment.baseUrl=https://api.payments.com
services.tax.country=LK
```

Config.properties file (contains key=value pairs)

BEST PRACTICES FOR IMPLEMENTATION

- Avoid Deprecated APIs
  - Always prefer supported alternatives

```
Date d = new Date();
int year = d.getYear(); // deprecated
```

```
Note: Main.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
125
```

getYear()

Deprecated.
As of JDK version 1.1, replaced by `Calendar.get(Calendar.YEAR) - 1900`.

```
LocalDate now = LocalDate.now(ZoneId.of("Asia/Colombo"));
int year = now.getYear();
```

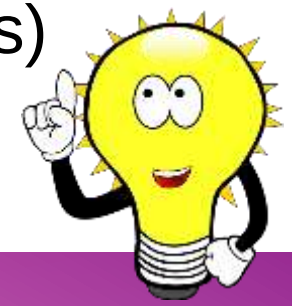
```
2025

=== Code Execution Successful ===
```

ORGANIZING CODE EFFECTIVELY

- Create Utility/ Helper classes for reusable logic
  - Read this: Java Helper vs. Utility Classes
- Follow Single Responsibility Principle
  - One class, one responsibility
- Group related functionality logically in packages/modules.
  - Aim for **High Cohesion, Low Coupling**
- Avoid bloated Utilities
  - A code smell – God Class
  - Keep utility classes focused.

WHY BEST PRACTICES & CODING STANDARDS AREN'T ENOUGH

- Requirements evolve (features, scale, compliance)
  - The original design no longer is adequate
- Problem Domain Understanding improves
  - models, names, and boundaries must be reshaped to match the domain.
- Software entropy
  - many small clean changes still create coupling and duplication over time.
- Style ≠ architecture
  - conventions ensure readability, not good architecture.
- Delivery trade-offs create technical debt (due to code smells)
- **Refactoring** is how you address these problems safely.

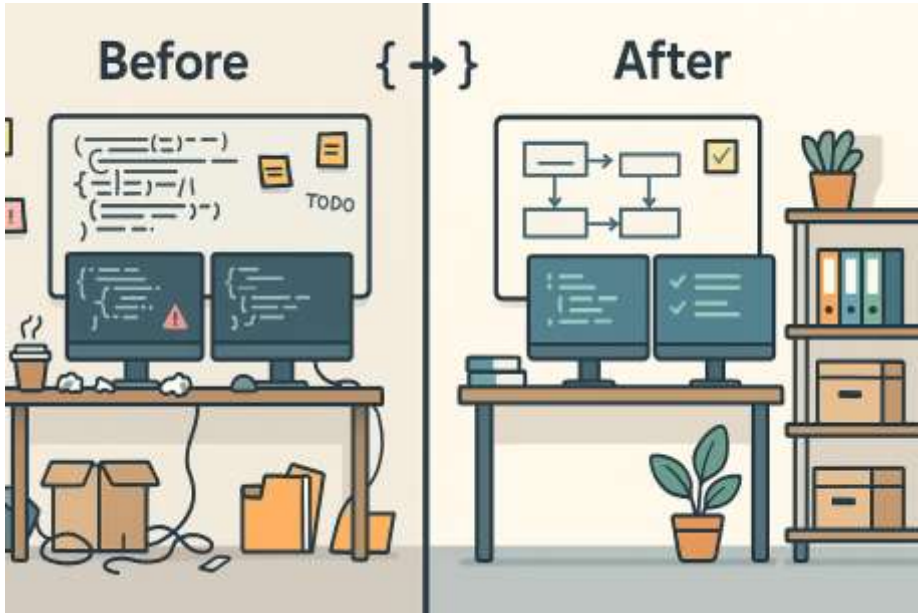


FROM CODE SMELLS TO REFACTORING

- Code smells are indicators of deeper problems
  - not bugs, but warnings.
- Smells make code harder to maintain and extend.
- Refactoring removes smells and restores design integrity.
- Examples:
  - Long Method → Extract Method
  - Large Class → Extract Class
  - Switch Chains → Replace with Polymorphism

REFACTORING FUNDAMENTALS

- Refactoring is *“a **change** made to the **internal structure** of software to make it **easier to understand** and **cheaper to modify without changing its observable behavior**.”* – Martin Fowler.

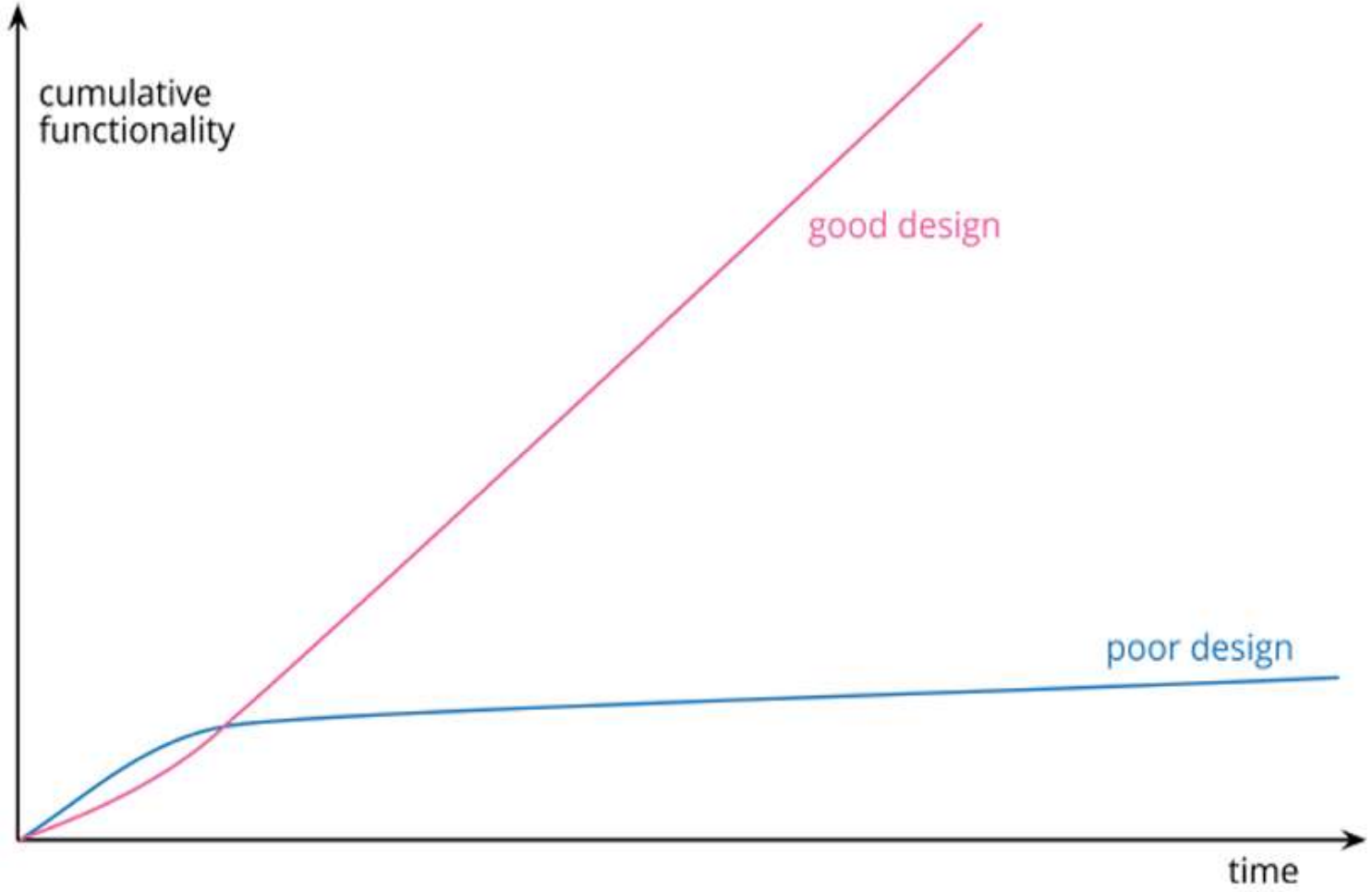


Source:chatGPT

REFACTORING FUNDAMENTALS

- Goals
  - Improve design
  - make code easier to understand
  - help find bugs
  - enable faster development
- Small Steps Principle
  - Incremental changes reduce risk

REFACTORING FUNDAMENTALS



Apparent ease of modification of code with good vs poor design, Source: Martin Fowler, Refactoring: Improving the Design of Existing Code, 2nd Edition

THE REFACTORING PROCESS

- Identify a problem (code smell).
- **Write/verify tests** before changes.
- Apply one or more refactoring techniques.
- Re-run tests to ensure no change in behavior.
- Repeat - refactoring is continuous.
- Common Refactoring Techniques: See Refactoring Techniques

REFACTORING EXAMPLE: EXTRACT METHOD

- Problem:
  - You have a code fragment that can be grouped together.

```
public void printInvoice(Order order) {
    System.out.println("Invoice for " + order.getCustomerName());
    double total = 0;
    for (Item item : order.getItems()) {
        System.out.println(item.getName() + ": " + item.getPrice());
        total += item.getPrice();
    }
    System.out.println("Total: " + total);
}
```

REFACTORING EXAMPLE: EXTRACT METHOD

- Solution:
  - Move this code to a separate new method and call from the old code.

```
public void printInvoice(Order order) {
    printHeader(order);
    printDetails(order);
}
```

```
private void printHeader(Order order) {
    System.out.println("Invoice for " + order.getCustomerName());
}

private void printDetails(Order order) {
    double total = 0;
    for (Item item : order.getItems()) {
        System.out.println(item.getName() + ": " + item.getPrice());
        total += item.getPrice();
    }
    System.out.println("Total: " + total);
}
```

REFACTORING EXAMPLE: INTRODUCE PARAMETER OBJECT

- Problem:
  - Your methods contain a repeating group of parameters.

```
public void createOrder(String customerName, String street, String city
    , String postalCode, List<Item> items) {
    // Implementation
}

public void estimateDelivery (String customerName, String street,
    String city, String postalCode, List<Item> items) {
    // Implementation
}
```

REFACTORING EXAMPLE: INTRODUCE PARAMETER OBJECT

- Solution:
  - Replace these parameters with an object.

```
public void createOrder(Customer customer, List<Item> items) {
    // Implementation
}

class Customer {
    String name;
    Address address;
}

class Address {
    String street;
    String city;
    String postalCode;
}
```

REFACTORING EXAMPLE: REPLACE MAGIC NUMBERS WITH CONSTANTS

- Problem:
  - Your code uses a number that has a certain meaning to it.

```
double circumference = 2 * 3.14159 * radius;
```

- Solution:
  - Replace this number with a constant that has a human-readable name explaining the meaning of the number.

```
public static final double PI = 3.14159;
double circumference = 2 * PI * radius;
```

WHEN NOT TO REFACTOR

- Performance-sensitive code
  - Refactoring that makes code cleaner but adds overhead in critical hot paths may harm performance.
- Stable, rarely-changed modules
  - If a component is stable, well-tested, and rarely modified, refactoring may not yield enough value.
- Impending deadlines
  - Risky to refactor when delivery schedules are tight and adequate regression testing isn’t possible.
- Lack of test coverage
  - Without strong tests, behavior-preserving refactoring cannot be safely guaranteed.
- Avoid refactoring purely for stylistic differences that don’t improve maintainability.

S U M M A R Y

- **Coding Standards** make code uniform, which helps to reduce complexities in collaborations.
- **Best practices** make code correct and operable day-to-day.
- **Refactoring** keeps the design healthy as software evolve.
- Good code quality = standards + best practices + refactoring.
- Refactoring improves design without altering **observable behavior**.
- Refactoring is a **habit**, not a one-time activity.

REFERENCES & FURTHER READING

- Martin Fowler, *"Refactoring: Improving the Design of Existing Code"*, 2nd Edition - Chapters 2, 3.
- Refactoring Techniques - refactoring.guru

A C K N O W L E D G M E N T S

- Some images and slides were taken from previous SE3070 – CSSE Lecture slide decks by Mr. Udara Samaratunge.

THANK YOU!

Vishan Jayasinghearachchi

Department of Software Engineering

vishan.j@sliit.lk