upGrad ENTERPRISE

Refactoring & Design Patterns in C#

By
Narasimha Rao T
Corporate Trainer and Mentor
Professional Development Trainer
tnrao.trainer@gmail.com

1. Refactoring Concepts

Definition:

Refactoring is the process of **restructuring existing code without changing its external behavior**. The goal is to improve its internal structure, maintainability, and readability.

Key Points:

- Improves code quality.
- Makes future changes easier.
- Keeps functionality the same.
- Often done in small, incremental steps.

Examples in C#:

```
// Before Refactoring
if(userType == "Admin") { /* admin logic */ }
else if(userType == "Customer") { /* customer logic */ }

// After Refactoring (Strategy Pattern)
userRole.ExecuteRoleSpecificLogic();
```

2. Why Do We Use Refactoring?

- **Readability:** Easier for developers to understand.
- **Maintainability:** Less effort to make changes.
- **Reusability:** Reduces duplicate logic.
- **Performance:** Sometimes improves efficiency.
- **Bug Reduction:** Cleaner code is less error-prone.
- **Prepares for Scaling:** Better suited for adding features.

Benefits of Refactoring

- Cleaner, modular design.
- Reduced complexity.
- Easier debugging.
- Encourages best practices.
- Increases developer productivity.

3. Code Smells

Definition:

Indicators in the code that something may be wrong in design or structure — not bugs, but signs of poor implementation.

Common Code Smells in C#:

1. **Long Method:** A method doing too much.
2. **Large Class:** A class with too many responsibilities.
3. **Duplicated Code:** Same logic repeated in multiple places.
4. **Long Parameter List:** Too many method parameters.
5. **Feature Envy:** A method overly dependent on another class's data.
6. **Shotgun Surgery:** A small change requires edits in many places.
7. **God Object:** A class that knows or does too much.

4. How to Refactor Using Design Patterns

4.1 Factory Pattern

- **Purpose:** Create objects without exposing creation logic to the client.
- **When to Use:** When object creation is complex or should be centralized.

```
public interface IShape { void Draw(); }
public class Circle : IShape { public void Draw() => Console.WriteLine("Circle"); }
public class ShapeFactory {
    public static IShape GetShape(string type) =>
        type switch {
            "Circle" => new Circle(),
            _ => throw new ArgumentException("Invalid type")
        };
}
```

4.2 Strategy Pattern

- **Purpose:** Encapsulate interchangeable behaviors and select them at runtime.
- **When to Use:** When you have multiple algorithms for a task.

```
public interface ISortStrategy { void Sort(List<int> list); }
public class QuickSort : ISortStrategy { public void Sort(List<int> list) => Console.WriteLine("QuickSort"); }
public class SortContext {
    private ISortStrategy _strategy;
    public SortContext(ISortStrategy strategy) { _strategy = strategy; }
    public void ExecuteStrategy(List<int> list) => _strategy.Sort(list);
}
```

4.3 Builder Pattern

- **Purpose:** Construct complex objects step-by-step.
- **When to Use:** When creating objects with many optional parameters.

```
public class Report {
    public string Title { get; set; }
    public string Content { get; set; }
}
public class ReportBuilder {
    private Report _report = new Report();
    public ReportBuilder SetTitle(string title) { _report.Title = title; return this; }
    public ReportBuilder SetContent(string content) { _report.Content = content; return this; }
    public Report Build() => _report;
}
```

4.4 Singleton Pattern

- **Purpose:** Ensure only one instance of a class exists.
- **When to Use:** For shared resources like configuration or logging.

```
public class Logger {
    private static Logger _instance;
    private static readonly object _lock = new();
    private Logger() { }
    public static Logger Instance {
        get {
            lock (_lock) {
                return _instance ??= new Logger();
            }
        }
    }
}
```

5. Real-Time Case Studies

Case Study 1 – Payment Gateway Integration

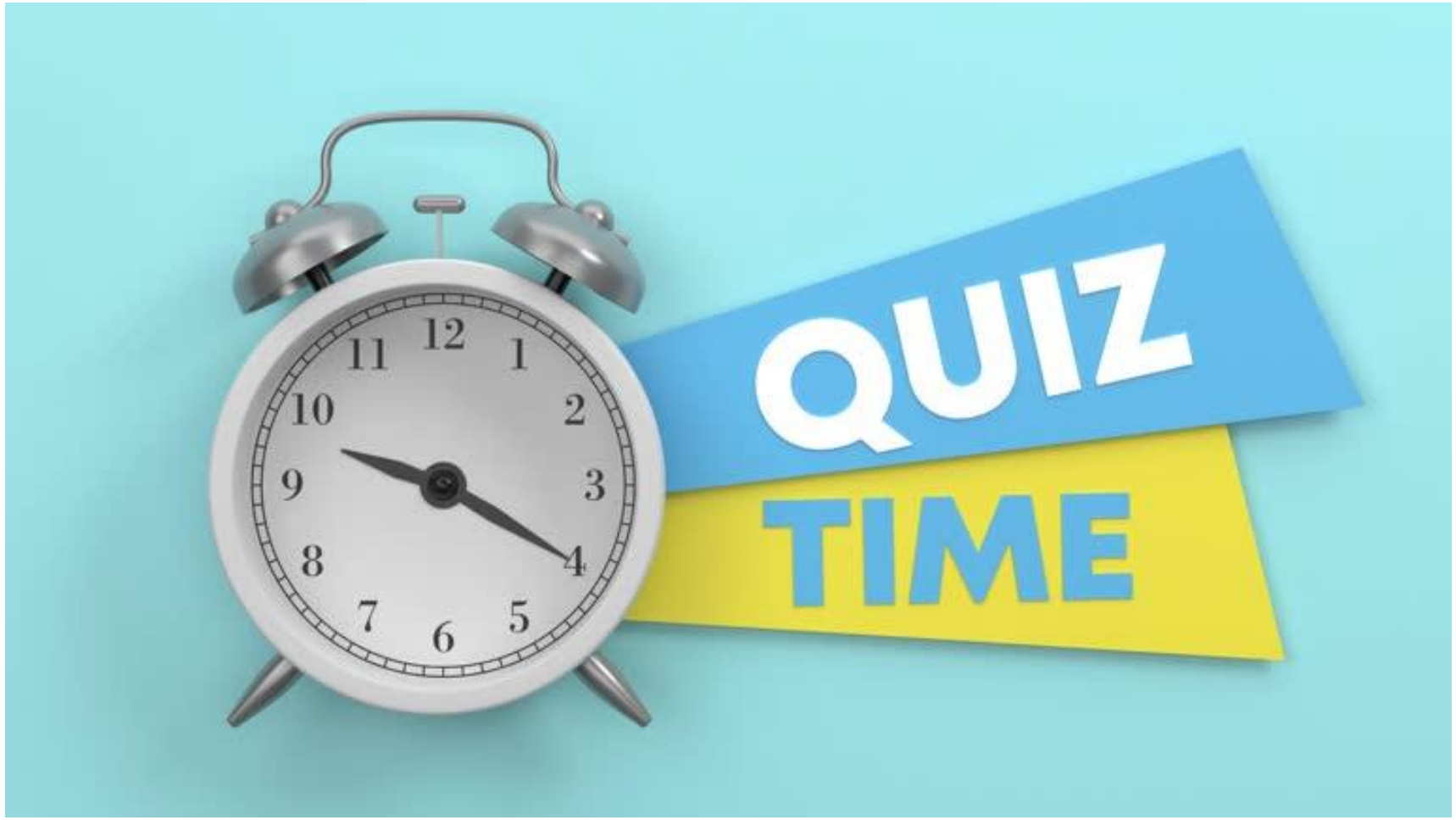
- Problem: Payment processing logic spread across multiple methods.
- Refactoring: Used **Strategy Pattern** to separate payment algorithms (Credit Card, PayPal, UPI).

Case Study 2 – Report Generation Tool

- Problem: Complex constructor with too many parameters.
- Refactoring: Used **Builder Pattern** to create reports with optional fields.

Case Study 3 – Configuration Management

- Problem: Multiple configuration objects loaded repeatedly.
- Refactoring: Applied **Singleton Pattern** to have one configuration object shared across the app.


QUIZ
TIME

Quiz Questions

1. What is refactoring? How is it different from rewriting code?
2. What are Code Smells? Can you name five?
3. When would you use the Strategy Pattern during refactoring?
4. Explain a real-world scenario for the Builder Pattern.
5. How does the Singleton Pattern help in resource management?
6. What are the risks of refactoring?
7. Can refactoring introduce bugs? How do you prevent it?
8. How do you identify where to refactor?
9. Difference between Factory and Builder patterns in C#?
10. Explain the relationship between Code Smells and Design Patterns.

upGrad ENTERPRISE

Q & A

Narasimha Rao T
tnrao.trainer@gmail.com