Design Wise, Test Smart

Building Resilient Test Frameworks with OOP, SOLID, and Design Patterns

in/bsridharpatnaik

Contents

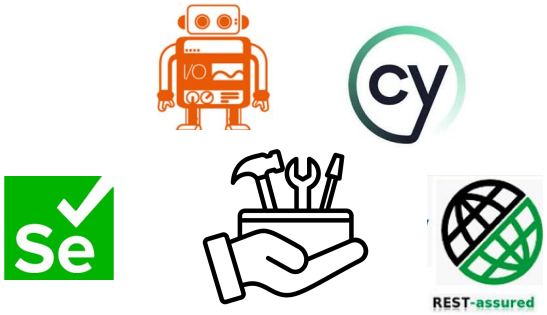
Introduction

Everyone talks about knowing the latest frameworks and list the tools they've mastered and boast about their automation tech stack.

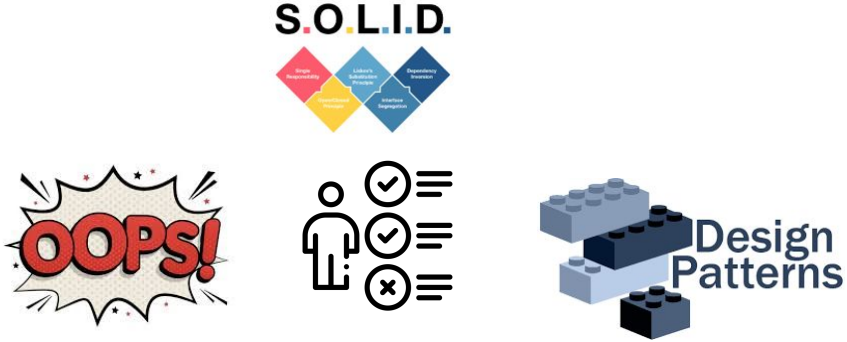
But What Are We Missing?

- In this race to adopt the latest and greatest tools, many forget that frameworks come and go, but fundamentals stay.
- If your foundation is shaky, even the best tools will crumble under the weight of poor structure and bad practices.

Why Fundamentals Still Matter



Tools



Skills

Tools Evolve, But Fundamentals Endure

- Selenium and other tools offer ready-made methods, but the real skill is in how you stich them with strong coding practices.
- Tools and frameworks change, but fundamentals stay constant. Master the basics, and you can build frameworks in any language and adapt to any tool.

Impact of Bad Code

60%

Time in Test Automation is Spent on Maintaining Test Scripts

Source: Tricentis Research

70%

Automation Framework Failures are Due to Not Following Principles

Source: Industry Insights (Referenced in Selenium communities)

62%

Organizations Struggle with Test Automation Scalability

Source: SmartBear State of Software Testing Report 2021

50%

Productivity is lost due to Poor Test Automation Code Quality

Source: McKinsey & Company Research

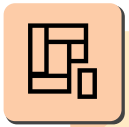
Why Do These Bad Statistics Exist?

RECAP

- 60% Time in Test Automation is Spent on Maintaining Test Scripts
- 62% Organizations Struggle with Test Automation Scalability
- 70% Automation Framework Failures are Due to Not Following Principles
- 50% Productivity is lost due to Poor Test Automation Code Quality

Lack of Adherence to Principles

Quick Fixes and Shortcuts



Lack of Modularity and Reusability

Poor Test Architecture

Inconsistent or No Code Reviews

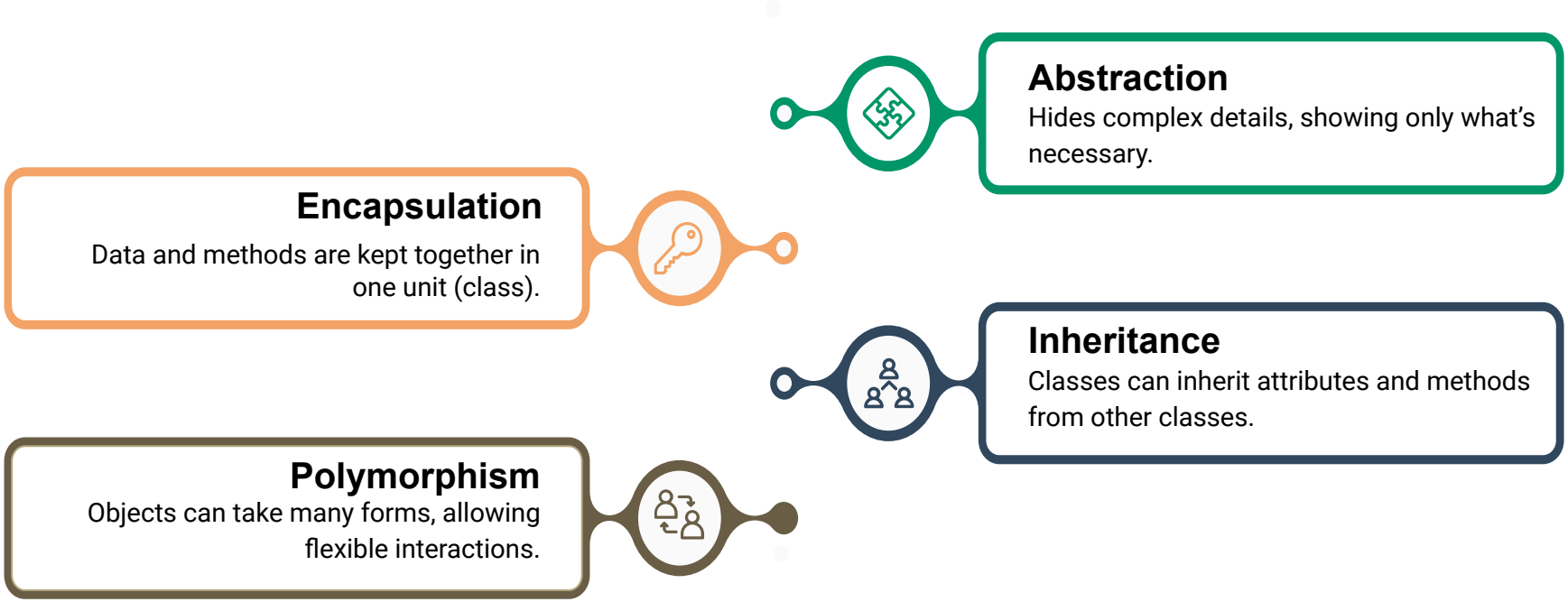
Hard-Coded Values and Dependencies

The Solution: Writing Maintainable and Scalable Code

Object-Oriented Programming Structure

Object-Oriented Programming (OOP)

OOP is a programming paradigm based on the concept of "objects," which can contain data (fields or attributes) and code (methods or behaviors)



Encapsulation

Grouping related data and functions into a single unit, restricting direct access to some components.

The LoginPage class encapsulates the login page actions, exposing only necessary methods to interact with the login page

```
public class LoginPage {

    private WebDriver driver;

    public LoginPage(WebDriver driver) {
        this.driver = driver;
    }

    public void open() {
        driver.get("http://example.com/login");
    }

    public void enterUsername(String username) {
        driver.findElement(By.id("username")).sendKeys(username);
    }

    public void enterPassword(String password) {
        driver.findElement(By.id("password")).sendKeys(password);
    }

    public void submit() {
        driver.findElement(By.id("loginButton")).click();
    }
}
```

Abstraction

Hiding implementation details and exposing only the necessary functionality.

The login method abstracts away the details of entering credentials and submitting the form. The actual input methods are hidden as private.

```java
public class LoginPage {

    private WebDriver driver;

    public LoginPage(WebDriver driver) {
        this.driver = driver;
    }

    public void login(String username, String password) {
        enterUsername(username);
        enterPassword(password);
        submit();
    }

    private void enterUsername(String username) {
        driver.findElement(By.id("username")).sendKeys(username);
    }

    private void enterPassword(String password) {
        driver.findElement(By.id("password")).sendKeys(password);
    }

    private void submit() {
        driver.findElement(By.id("loginButton")).click();
    }
}
```

Inheritance

Extending a class to reuse its functionality.

LoginPage inherits common functionality from BasePage, such as isElementDisplayed. This prevents code duplication across page classes.

```java
public class BasePage {

    protected WebDriver driver;

    public BasePage(WebDriver driver) {
        this.driver = driver;
    }

    public boolean isElementDisplayed(By locator) {
        return driver.findElement(locator).isDisplayed();
    }
}

public class LoginPage extends BasePage {

    public LoginPage(WebDriver driver) {
        super(driver);
    }

    public void login(String username, String password) {
        driver.findElement(By.id("username")).sendKeys(username);
        driver.findElement(By.id("password")).sendKeys(password);
        driver.findElement(By.id("loginButton")).click();
    }

    public boolean isLoginSuccessful() {
        return isElementDisplayed(By.id("welcomeMessage"));
    }
}
```

Polymorphism

Allowing an object to take multiple forms.

The driver variable is polymorphic and can take different WebDriver implementations (e.g., ChromeDriver, FirefoxDriver). Switching browsers only requires changing one line.

```java
public class TestRunner {

    private WebDriver driver;

    public TestRunner(WebDriver driver) {
        this.driver = driver;
    }

    public void runLoginTest() {
        LoginPage loginPage = new LoginPage(driver);
        loginPage.login("testUser", "testPassword");
        if (loginPage.isLoginSuccessful()) {
            System.out.println("Login Test Passed");
        } else {
            System.out.println("Login Test Failed");
        }
        driver.quit();
    }

    public static void main(String[] args) {
        WebDriver driver = new ChromeDriver(); // Can switch t
        TestRunner runner = new TestRunner(driver);
        runner.runLoginTest();
    }
}
```

Design Patterns

Design Patterns

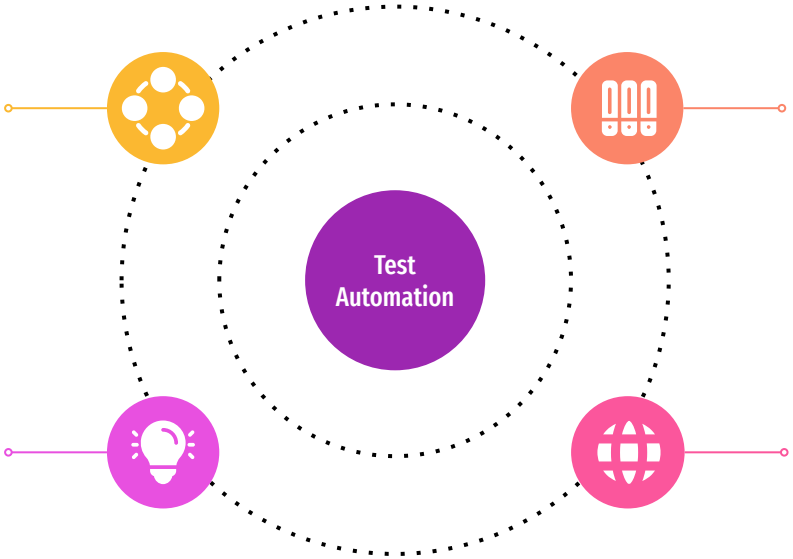
Design patterns are well-established solutions to common software design problems. They are templates that can be applied to solve specific problems in a particular context. Example:

Data Setup

Creating complex test data configurations is repetitive and error-prone. We need a flexible mechanism to generate data based on test requirements.

Dynamic Test Scenarios

Handling multiple test scenarios (e.g., valid login, invalid login) in a single test case results in complex, hard-to-maintain code. We need a way to manage varying scenarios cleanly.

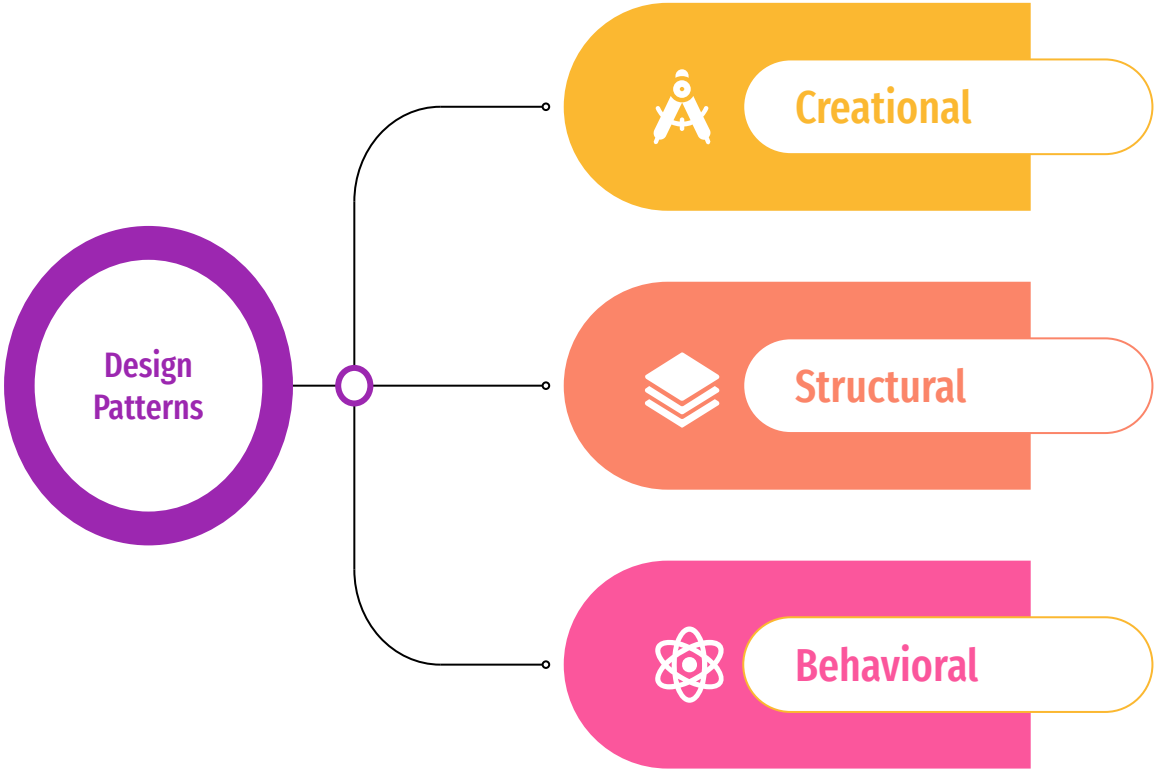


Thread-Safe Logging

During parallel test execution, log messages from different threads often get mixed up. The logger should ensure sequential logging for better traceability.

Browser Independence

Running tests on different browsers often leads to duplicated code. There should be a centralized way to create browser drivers dynamically.



Patterns for creating objects, like Factory and Singleton.

Patterns that deal with object composition, like Adapter and Decorator.

Patterns that manage object interactions, like Strategy and Observer.

- Test Setup Patterns
- Basic Structural Patterns
- Test Execution Patterns

Test Setup Patterns

Problem Statement

Imagine you're maintaining an automation script that runs on multiple browsers—Chrome, Firefox, and Safari. But here's the challenge: the list of supported browsers is not static. It frequently changes based on requirements. This means, every time a new browser is added or removed, you have to manually update all your test cases to reflect the change. Over time, this can result in:

High Maintenance Effort

Every addition or removal of browser support requires manual updates across hundreds of test files, consuming valuable development time.

Inconsistent Code

Different team members implement browser-specific logic differently, leading to inconsistent approaches and harder-to-maintain test suites.

Error-Prone Process

Manual modifications to browser configurations across multiple test files increase the risk of missed updates and configuration mistakes.

Configuration Overhead

Managing separate browser settings, driver capabilities, and environment-specific configurations becomes scattered and difficult to track across the framework.

Similar Use Cases

Custom Reporting

In some test runs, you may need HTML reports, while in others, you need XML or JSON formats.

Test Environment Setup

Test environments (local, staging, production) often have different configurations. Constantly updating tests with specific configurations leads to a lot of maintenance work

API Client Creation

When your tests need to handle different types of APIs like REST or SOAP, you need to manage which client to use. If the API types change, you'll end up updating each test that handles API interactions.

Test Data Providers

Dynamically pulling test data from CSV, databases, or JSON files requires managing multiple sources. As new data sources are introduced, this leads to similar issues as with browser handling.

Factory Pattern

Pattern Category: **Creational Pattern**

Problem

Object creation or configuration is frequently changing, leading to:

- Code duplication
- High maintenance
- Error-prone processes

Solution

- Centralize Object Creation: Instead of hardcoding logic for creating WebDrivers, API clients, or data providers in each test, centralize the creation in one place.
- Dynamic Selection: Let one entity decide which object to use based on the input (browser, API, or environment). This reduces the need for manual changes across all tests.

The Factory Pattern abstracts object creation, making the code more modular, easier to extend, and less prone to errors by decoupling object creation logic from the rest of the test logic.

Code Example

Bad Code https://tinyurl.com/2twaz8m7

Refactored Code https://tinyurl.com/5esvfm2z

Problem Statement

Imagine you're writing automation scripts that need to configure complex test data, such as creating user profiles with many optional attributes—name, email, address, phone number, etc. Managing the creation of these objects with multiple optional fields is cumbersome and error-prone. Over time, this can result in:

Complex Constructor Calls

Passing numerous parameters in constructors becomes unwieldy, especially when many fields are optional or have default values.

Inconsistent Object Creation

Different developers create test objects in varied ways, leading to inconsistent initialization and potential validation issues.

Hard to Extend

Adding new fields requires modifying existing object creation code across multiple test files, increasing maintenance overhead significantly.

Validation Complexity

Managing object state validation and mandatory field checks becomes scattered across tests, making it difficult to maintain consistent rules.

Similar Use Cases

Test Data Setup

Configuring complex test data with many optional fields (e.g., creating a user profile, configuring product details) becomes difficult to manage if handled manually.

Creating API Request Payloads

When building API requests, there are often many optional parameters. Hardcoding these values in tests leads to duplication and inflexibility.

UI Component Initialization

In UI tests, initializing complex components with multiple optional attributes (e.g., button styles, actions, labels) can be repetitive if done manually in each test.

Database Connection Configuration

Configuring database connections with various optional parameters (e.g., pool size, timeout, retry strategy) leads to complicated constructor calls, especially when not all parameters are required.

Builder Pattern

Pattern Category: **Creational Pattern**

Problem

Manually managing the creation of complex objects (e.g., user profiles, API requests) with many optional parameters leads to:

- Complicated Constructor Calls
- Inconsistent Object Creation
- High Maintenance Effort

Solution

The Builder Pattern simplifies object creation by constructing complex objects step by step. Instead of passing all parameters to a constructor, you build the object by chaining methods, making the code more readable and maintainable.

The Builder Pattern simplifies the construction of complex objects by separating the object creation process from its representation. It allows you to construct objects step by step, making the code more flexible and readable, especially when dealing with optional parameters.

Code Example

Bad Code https://tinyurl.com/4afe42hb

Refactored Code https://tinyurl.com/dh9j2u6r

Problem Statement

Imagine you're working on an automation project where you need different combinations of test data - users with various roles, orders in different states, products with multiple configurations. You find yourself:

Duplicated Test Data

Repeatedly copying test data across multiple test files leads to maintenance nightmares and inconsistent data updates.

Data File Maintenance

Managing large JSON or Excel files with overlapping test data creates redundancy and makes updates time-consuming.

Business Rule Changes

When business requirements change, updating scattered test data across various locations becomes error-prone and inefficient.

Complex Object Creation

Manually creating test objects with numerous fields and dependencies becomes tedious and prone to initialization errors.

Similar Use Cases

Order Generation Testing

Building orders with different products, shipping methods, and payment statuses to validate various checkout and order processing flows.

User Session Testing

Creating test scenarios with different user permissions, authentication tokens, and session states to verify access control mechanisms.

API Payload Testing

Generating complex request payloads with multiple nested objects and arrays to test API validation and processing logic.

Form Validation Testing

Creating diverse datasets to test form validation rules, field dependencies, and error handling across different input combinations.

Test Data Patterns

Problem

When test data is managed without proper patterns:

- No reuse of common test data
- Hard to maintain when business rules change
- Complex object creation scattered across tests
- Inconsistent test data across test suite

Solution

Test Data Management Patterns provide structured ways to:

- Create test objects consistently
- Reuse common test data
- Build complex objects step by step
- Maintain test data in one place

Pattern Category: **Creational Pattern**

Three main patterns work together:

- Object Mother: Creates standard test
- Objects Builder: Constructs complex objects flexibly
- Test Data Factory: Centralizes object creation logic

Code Example

Bad Code https://tinyurl.com/5f83ucpd

Refactored Code https://tinyurl.com/ynkdjfev

Basic Structural Patterns

Problem Statement

Imagine you're maintaining a large suite of UI automation tests for a web application with multiple pages (e.g., Login, Dashboard, Profile). Each test interacts directly with the page elements, leading to a lot of code duplication. Over time, as the application's UI changes, this setup creates problems:

High Maintenance Effort

Each time a UI element changes (like an ID or CSS selector), every test script that references that element needs to be updated.

Code Duplication

Tests for different scenarios on the same page duplicate the code for locating and interacting with the same elements.

Unclear Test Logic

Test scripts contain both business logic and UI interactions, making the tests harder to read and understand.

Error-Prone Updates

Updating UI element locators across multiple tests increases the chance of missed changes and leads to inconsistent test results.

Page Object Model (POM)

Pattern Category: **Structural Pattern**

The Page Object Model (POM) pattern organizes test code by encapsulating each page's interactions in dedicated classes, known as page objects. This separation of test logic from UI elements improves maintainability, enhances reusability, and keeps tests focused on business scenarios.

Similar Use Cases

API Testing Integration with UI Tests

When tests need to validate both UI elements and API responses together, POM helps organize API endpoints, request/response handling, and corresponding UI validations.

Cross-browser Testing Frameworks

For running tests across multiple browsers (Chrome, Firefox, Safari), POM centralizes browser-specific element locators and handling strategies. It simplifies maintenance when dealing with browser-specific behavior or element identification differences.

Data-Driven Test Automation

For scenarios requiring same test flows with different test data sets, POM separates the test data handling from UI interactions. This makes it easier to maintain test data sources and modify test flows independently while reusing the same page objects.

Code Example

Bad Code https://tinyurl.com/yrydd92h

Refactored Code https://tinyurl.com/4hjwrkcx

Problem Statement

Imagine you're working on an automation framework that interacts with two different APIs—one for RESTful services and another for SOAP services. These APIs have completely different interfaces, but your test cases need to interact with both in a unified way. Over time, this can result in:

High Maintenance Effort

Changes in any API interface require updates across multiple test cases, consuming significant time and increasing technical debt.

Inconsistent Code

Different APIs require unique handling approaches and syntax, making test code inconsistent and significantly harder to understand and maintain.

Difficult to Reuse

Incompatible API interfaces prevent test logic reuse across services, forcing duplicated code and reducing testing efficiency significantly.

Complex Test Setup

Various APIs need different configurations, authentication methods, and data states, making environment setup overly complex and hard to manage.

Similar Use Cases

API Testing

When testing different types of APIs (e.g., REST and SOAP), each one has its own way of making requests and handling responses. Manually handling these differences in every test leads to duplication and complexity.

UI Test Automation

Different web browsers may have slight variations in their WebDriver implementations. Using an adapter can standardize how you interact with different browsers.

Database Interaction

If your tests interact with different databases (e.g., MySQL, MongoDB), an adapter can abstract away the differences in how each database is queried.

File Handling

Different file formats (e.g., CSV, JSON) have unique methods for reading and writing data. An adapter can provide a unified interface for handling various file types in your test automation scripts.

Adapter Pattern

Pattern Category: **Structural Pattern**

Problem

Manually managing interactions with incompatible APIs (or services) in every test case leads to:

- Inconsistent Code
- High Maintenance Effort
- Difficult to Reuse

Solution

The Adapter Pattern provides a way to unify interfaces by creating an adapter class that acts as a middle layer between the incompatible interfaces. This makes it easier to interact with different APIs or services using a common interface.

The Adapter Pattern allows incompatible interfaces to work together by providing a common interface to clients. In test automation, this pattern is useful when interacting with different APIs, services, or tools that have incompatible interfaces, enabling tests to be written in a unified way.

Code Example

Bad Code https://tinyurl.com/ty6nrd4u

Refactored Code https://tinyurl.com/m6tzmjpy

Test Execution Patterns

Problem Statement

Imagine you're maintaining an automation script that needs to log results consistently across all tests. However, multiple instances of the logger are being created, leading to inconsistent logging. Over time, this results in:

High Memory Usage

Multiple instances of the same resource consume unnecessary memory and processing power, impacting test execution performance.

Inconsistent Logs

Different logger instances create scattered log entries, making it difficult to track test execution flow and debug issues.

Difficult Maintenance

Managing and synchronizing multiple instances of the same resource creates unnecessary complexity and increases maintenance overhead.

Resource Conflicts

Concurrent access to shared resources like database connections or file handlers leads to race conditions and unpredictable behavior.

Similar Use Cases

Configuration Management

Maintains a single source of test configurations (environment variables, test settings, feature flags) to ensure consistent test execution across different modules.

Database Connection

Manages a single database connection instance across test suites to prevent connection pool exhaustion and optimize resource usage during testing.

WebDriver Management

Controls browser instance creation and reuse across test cases, preventing multiple unnecessary browser sessions and improving test execution speed.

Test Report Generator

Ensures all test results are logged to a single report instance, maintaining consistent formatting and preventing data loss from scattered reports.

Singleton Pattern

Pattern Category: **Creational Pattern**

Problem

Object instantiation is frequently needed in the automation process, leading to:

- Code duplication
- High Memory Usage
- Inconsistent Behavior

Solution

Centralized Object Creation: Ensure that only one instance of an object (e.g., logger, configuration, database connection) is created and reused across tests. This reduces redundancy, lowers memory usage, and ensures consistency.

The Singleton Pattern ensures that a class has only one instance and provides a global point of access to it. This is especially useful for shared resources like loggers, configuration settings, or database connections. By centralizing the creation of such objects, we avoid redundancy and maintain consistency across tests.

Code Example

Bad Code https://tinyurl.com/4bxznk6h

Refactored Code https://tinyurl.com/yywhfy5m

Problem Statement

Imagine you're maintaining an automation script that handles different payment methods. However, the payment methods are frequently updated or expanded. This means, every time a new payment method is added, you must manually update your test cases to reflect the change. Over time, this can result in:

High Maintenance Effort

Adding or modifying payment methods requires changes across multiple test files, making maintenance time-consuming and complex.

Error-Prone Process

Manual updates to payment handling logic across various test scenarios increase the risk of incorrect validations.

Inconsistent Code

Different team members implement payment verification logic differently, leading to scattered and hard-to-maintain test approaches.

Complex Test Flows

Managing different payment workflows, validation rules, and error scenarios becomes increasingly difficult without a standardized approach.

Similar Use Cases

API Request Handling

Different types of API requests (e.g., GET, POST, PUT) require different handling. Constantly updating the logic for every API request leads to duplicated and hard-to-maintain code.

Test Data Validation

Different types of test data (JSON, XML, CSV) need to be validated in different ways. If you hardcode validation logic into your tests, any new data format requires significant code changes.

Retry Mechanism

In test automation, you may need different retry strategies depending on the test environment or test case (e.g., exponential backoff, immediate retry). Constantly modifying the retry logic can be complex.

Browser Handling

Similar to the payment method example, automation scripts that handle multiple browsers (e.g., Chrome, Firefox, Safari) require different handling. Hardcoding browser-specific logic into each test leads to duplicated code and high maintenance.

Strategy Pattern

Pattern Category: **Behavioral Pattern**

Problem

Object instantiation is frequently needed in the automation process, leading to:

- Code duplication
- High Memory Usage
- Inconsistent Behavior

Solution

The solution is to encapsulate different algorithms (or strategies) for handling various cases, such as payment methods, API requests, or data validation, into separate classes. You can dynamically choose the appropriate strategy at runtime without modifying the test logic.

The Strategy Pattern encapsulates a family of algorithms (e.g., payment processing methods) and allows them to be interchangeable. It enables the selection of an algorithm (or strategy) at runtime, without modifying the client code. This pattern is useful when you need to switch between different behaviors (e.g., API requests, test data validation) dynamically.

Code Example

Bad Code https://tinyurl.com/ydx9cm3e

Refactored Code https://tinyurl.com/bdf64w5w

Test Reporting / Observation Patterns

Problem Statement

Imagine you're maintaining an automation script that monitors test execution results in real time. When a test completes, multiple components need to be notified—such as the reporting system, logging system, and alert system. Manually notifying each component becomes complex and difficult to manage. Over time, this can result in:

High Complexity

Coordinating notifications across different systems like reporting, logging, and alerts becomes increasingly complicated and hard to manage.

Tight Coupling

Direct dependencies between test execution and notification systems make it difficult to add or remove monitoring components.

Error-Prone Updates

Adding new notification requirements means modifying existing test code, risking disruption of working notification flows and introducing bugs.

Inconsistent Notifications

Different parts of the test framework handle notifications differently, leading to missed updates and inconsistent monitoring coverage.

Similar Use Cases

Test Result Monitoring

Different systems (reporting tools, dashboards, logging services) need real-time updates when test execution status changes, pass/fail events, or errors occur.

Test Environment Status

Multiple components need notification when environment states change, like service availability, resource utilization, or configuration updates during test runs.

Data Validation Events

Different validators need to monitor data changes during testing, such as database updates, file modifications, or API response transformations.

UI State Changes

Various test components monitor dynamic UI changes, like element visibility, form submissions, or AJAX updates during automation execution.

Observer Pattern

Pattern Category: **Behavioral Pattern**

Problem

Managing notifications for different components when a test completes leads to:

- Code Duplication
- High Maintenance Effort
- Tight Coupling

Solution

The solution is to use a central mechanism where observers (e.g., reporting systems, logging systems) can subscribe to receive updates when a test completes. This decouples the test case from specific notification logic.

The Observer Pattern allows objects (observers) to subscribe to and receive updates from a subject. When the subject's state changes (e.g., a test completes), all subscribed observers (e.g., logging, reporting, alerts) are notified. This decouples the subject (test case) from the specific notification logic and allows for easy extensibility.

Code Example

Bad Code https://tinyurl.com/yhrp9zm7

Refactored Code https://tinyurl.com/3hc3hk48

Problem Statement

Imagine you're maintaining an automation script that logs test execution details, generates reports, and handles error reporting. However, adding these features (logging, reporting, error handling) to every test is tedious and leads to code duplication. Over time, this can result in:

Code Duplication

Implementing common functionalities like logging, screenshots, and retry mechanisms repeatedly across different test cases creates redundant code.

Difficult to Extend

Adding new cross-cutting features requires modifying existing test code, making it time-consuming and error-prone to enhance functionality.

Tight Coupling

Test logic becomes tightly intertwined with utility features like logging and reporting, making it harder to modify one without affecting others.

Maintenance Overhead

Managing multiple layers of functionality in each test case becomes complex, especially when updating common features across tests.

Similar Use Cases

API Request Enhancement

Decorating API test methods to add authentication headers, request timestamps, and correlation IDs without modifying base request logic.

Data Validation Enrichment

Adding additional validation layers to basic test checks, like boundary validation or data format verification, while keeping core validations simple.

Browser Action Augmentation

Enhancing basic Selenium actions with additional behaviors like highlights, waits, or JS scrolling without changing base element interactions.

Test Report Enhancement

Extending basic test results with extra metadata, custom formatting, or additional context while maintaining simple test result generation.

Decorator Pattern

Pattern Category: **Structural Pattern**

Problem

When adding additional functionality (e.g., logging, reporting, validation) to tests, it leads to:

- Code Duplication
- High Maintenance Effort
- Tight Coupling of core logic with extra features.

Solution

The solution is to use the Decorator Pattern, which allows you to dynamically add responsibilities (such as logging or reporting) to objects without modifying their structure. This keeps the core logic separate from additional features.

The Decorator Pattern allows additional functionality (e.g., logging, error handling) to be added to an object dynamically, without modifying its structure. This is particularly useful when you want to extend an object's behavior without altering the core logic.

Code Example

Bad Code https://tinyurl.com/24e7js3r

Refactored Code https://tinyurl.com/ypuy8wja

SOLID Principles

Problem Statement

Imagine you're working on a test automation framework, and one of your classes is responsible for multiple tasks: generating test data, executing tests, and logging results. Over time, this can result in:

High Complexity

A single class handling test data, execution, and reporting becomes unwieldy and difficult to understand or debug effectively.

Difficult to Maintain

Changes to one feature like logging require modifying a complex class, risking unintended effects on test execution.

Tight Coupling

Different functionalities become intertwined, making it impossible to modify test execution without impacting reporting or data generation.

Testing Challenges

Unit testing becomes complicated as each test must account for multiple responsibilities and their interconnected behaviors.

Similar Use Cases

Test Data Setup

If a class is responsible for both generating and validating test data, the logic becomes mixed and harder to manage. These should be separated into distinct classes.

Logging and Reporting

A class that handles test execution should not also be responsible for logging the results. The responsibility of logging should be moved to its own class.

Configuration Management

If a class responsible for test execution also manages environment-specific configurations, it mixes concerns and increases complexity.

UI Test Automation

A class that handles WebDriver interactions should not also be responsible for handling screenshots and reporting errors.

Single Responsibility Principle (SRP)

Problem

When a class has more than one responsibility (e.g., test execution, logging), it becomes:

- Complex to Understand
- Difficult to Maintain
- Error-Prone

Solution

The Single Responsibility Principle (SRP) states that a class should have only one reason to change, meaning it should only have one responsibility. By splitting responsibilities into separate classes, the code becomes cleaner, easier to maintain, and less prone to errors.

The Single Responsibility Principle ensures that a class has only one reason to change, meaning it should be responsible for only one aspect of the functionality. This makes the class easier to maintain, test, and extend.

Code Example

Bad Code https://tinyurl.com/ym9kyyz7

Refactored Code https://tinyurl.com/3uxtzk47

Problem Statement

Imagine you're working on a test automation framework that requires frequent updates to handle new test cases, new browsers, or APIs. Each time a new feature is added or a change is required, you have to modify the core framework code. Over time, this can result in:

High Risk of Bugs

Modifying existing test framework code for new features risks breaking current functionality and introducing regression issues.

Hard to Extend

Adding support for new browsers or API versions requires changing core code, making framework enhancement complex and risky.

Increased Maintenance Effort

Constant modifications to base classes for new features create technical debt and make maintenance increasingly difficult.

Version Control Challenges

Managing multiple versions of the framework becomes complex as core changes affect all dependent test implementations.

Similar Use Cases

Adding New Browser Support

You need to add support for a new browser (e.g., Edge) to an existing test suite. Modifying the base code to accommodate the new browser can lead to bugs and regressions.

Extending Report Formats

Your framework generates test reports in CSV format, but now you need to support HTML and JSON reports. Modifying the core reporting logic violates OCP and increases maintenance efforts.

Handling Different APIs

You are testing multiple APIs (REST, SOAP, GraphQL). Modifying your base API handling logic every time a new API is introduced makes the code harder to maintain.

Adding New Test Scenarios

When adding new test scenarios or validation rules, modifying the core validation logic makes the system more complex and prone to errors.

Open/Closed Principle (OCP)

Problem

When your code isn't designed to follow the Open/Closed Principle, adding new functionality requires modifying existing code. This leads to:

- Increased Risk of Bugs
- Hard to Maintain
- Difficult to Extend

Solution

The Open/Closed Principle (OCP) states that software entities (e.g., classes, modules, functions) should be open for extension but closed for modification. This means you should be able to add new functionality without modifying the existing code. Instead, extend the system through new classes or modules.

The Open/Closed Principle ensures that a class is **open for extension** but **closed for modification**. This means that you can add new features or functionality (like new report formats) without changing existing code. This leads to code that is easier to extend, maintain, and scale.

Code Example

Bad Code https://tinyurl.com/4xd6vcya

Refactored Code https://tinyurl.com/28xftemn

Problem Statement

Imagine you're working on a test automation framework where different types of tests (UI, API, Mobile) inherit from a base test class. However, some child classes override methods in ways that break expected behavior. Over time, this can result in:

Unexpected Behavior

Test classes override parent methods in incompatible ways, causing tests to behave differently than the base class contract.

Framework Instability

Base class assumptions about method behavior are violated, leading to random test failures and inconsistent results.

Maintenance Confusion

Developers can't rely on base class contracts, making it difficult to understand and maintain test hierarchy.

Integration Issues

Test classes can't be used interchangeably, breaking framework features that depend on base class behavior

Similar Use Cases

WebDriver Substitutions

You have a WebDriver class that's extended by ChromeDriver and FirefoxDriver. If a subclass doesn't behave like the WebDriver (e.g., not implementing a crucial method), switching between drivers will cause tests to break.

API Clients

You're testing multiple APIs, and you have a base APIClient class that's extended by RESTClient and SOAPClient. If one of the subclasses alters how requests are handled, it could break tests that depend on consistent API behavior.

Database Drivers

You have a DatabaseConnection class that is extended by MySQLConnection and MongoDBConnection. If a subclass does not follow the same contract, switching databases could cause unexpected behavior in tests.

Test Validation Rules

You have a base class for test validations. If a subclass changes the logic in a way that violates the expected behavior of the parent class, test validations might fail unexpectedly.

Liskov Substitution Principle (LSP)

Problem

When subclasses don't follow the behavior expected from their parent class, this leads to:

- Inconsistent Behavior
- Unreliable Tests
- Hard-to-Fix Bugs

Solution

The Liskov Substitution Principle (LSP) states that objects of a subclass should be replaceable with objects of the parent class without affecting the correctness of the program. Subclasses should behave in a way that their parent class does, adhering to the same contract.

The Liskov Substitution Principle ensures that subtypes can replace their parent type without affecting the correctness of the program. This leads to consistent behavior, fewer bugs, and reliable test automation.

Code Example

Bad Code https://tinyurl.com/yck789ae

Refactored Code https://tinyurl.com/38sufnjv

Problem Statement

Imagine you're working on a test automation framework where test classes are forced to implement unnecessary interfaces with methods they don't need. For example, a simple API test having to implement UI-specific methods. Over time, this can result in:

Bloated Tests

Test classes are forced to implement unnecessary methods, creating empty or dummy implementations that add complexity.

False Dependencies

Tests depend on interfaces they don't use, making them harder to understand and maintain over time.

Implementation Overhead

Developers waste time implementing unused interface methods, leading to reduced productivity and confusion.

Violation of YAGNI

Tests contain unused code and methods, violating "You Aren't Gonna Need It" principle and creating maintenance burden.

Similar Use Cases

WebDriver Interfaces

You have a WebDriver interface that defines browser operations like clicking, navigating, and scrolling. However, mobile browsers might only need a subset of these methods. Instead of using one large interface for all browser operations, you should break it into smaller interfaces.

API Client Interfaces

You're testing multiple APIs (e.g., REST, SOAP, GraphQL), each requiring different methods (e.g., GET, POST, DELETE). Forcing all API clients to implement methods they don't need violates ISP.

Database Connection Interfaces

A DatabaseConnection interface might define methods for relational databases (e.g., SQL queries) as well as NoSQL databases (e.g., collections). Using a single interface forces every database connection to implement irrelevant methods.

Reporting Interfaces

A TestReport interface defines methods for both CSV and HTML reports, but if some tests only need CSV or HTML, forcing them to implement unused methods violates ISP.

Interface Segregation Principle (ISP)

Problem

When interfaces are too large or contain methods that are irrelevant to certain clients, this leads to:

- Fat Interfaces: Clients are forced to implement methods they don't need.
- Unnecessary Complexity: Classes become more complex because they have to handle irrelevant operations.
- Hard to Maintain: Changes to the interface force multiple classes to update unnecessarily.

Solution

The Interface Segregation Principle (ISP) states that clients should not be forced to depend on interfaces they do not use. Instead of one large interface, break it into smaller, more specific interfaces that contain only the relevant methods for each client.

The Interface Segregation Principle ensures that clients should not be forced to depend on methods they do not use. By breaking large interfaces into smaller, more specific ones, you reduce complexity and make the code easier to maintain and extend.

Code Example

Bad Code https://tinyurl.com/2p9pstbp

Refactored Code https://tinyurl.com/mr48d9eh

Problem Statement

Imagine you're working on a test automation framework where test classes are tightly coupled with concrete implementations of dependencies like webdriver, data providers, or loggers. This means test classes directly create their dependencies. Over time, this can result in:

Testing Difficulties

Tests become hard to isolate for unit testing because they're tightly coupled with concrete implementations.

Limited Flexibility

Switching implementations (like changing logging systems or browsers) requires modifying multiple test classes directly.

Complex Configuration

Test setup becomes complicated as each test needs to manage its own dependency creation and configuration.

Maintenance Issues

Changes in dependency implementations force updates across all test classes that directly create these dependencies.

Similar Use Cases

WebDriver Dependencies

A test runner directly depends on specific WebDriver implementations (e.g., ChromeDriver, FirefoxDriver). When switching browsers, the high-level test runner must be updated, violating DIP.

API Clients

Your test automation framework directly depends on specific API clients (e.g., RESTClient, SOAPClient). If the API changes, the high-level logic must be modified, making it harder to adapt.

Database Connections

The framework depends on specific database implementations (e.g., MySQLConnection, MongoDBConnection). Changes to the database layer require changes in the higher-level test logic.

Test Report Generation

The test runner directly depends on specific reporting formats (e.g., CSV or HTML). Adding new formats requires changing the core test runner, leading to tight coupling.

Dependency Inversion Principle (DIP)

Problem

When high-level modules depend on low-level implementations, it leads to:

- **Tight Coupling**: High-level modules are closely tied to specific low-level classes, making changes difficult.
- **Limited Flexibility**: Adding or swapping out low-level implementations requires changes to the high-level modules.
- **Difficult Testing**: Unit testing becomes difficult because you are working with concrete classes instead of abstractions.

Solution

The Dependency Inversion Principle (DIP) states that high-level modules should not depend on low-level modules. Both should depend on abstractions (e.g., interfaces). This ensures flexibility, easier testing, and less coupling between modules.

The Dependency Inversion Principle ensures that high-level modules depend on abstractions (e.g., interfaces) instead of concrete classes. This allows for easier code extension, testing, and flexibility. Both the high-level and low-level modules should depend on abstractions.

Code Example

Bad Code https://tinyurl.com/mr4bxnst

Refactored Code https://tinyurl.com/bxxej5uh

Is Over-Engineering Always the Right Choice?

Imagine you're working on a small test automation project for a team with limited technical expertise. The team needs a framework quickly and wants easy onboarding. You decide to:

1. Follow all best practices.
2. Introduce design patterns and SOLID principles.
3. Split responsibilities into multiple classes.

But over time, you realize:

- **Development Slows Down**: Overhead in maintaining too many components for a simple need.
- **High Learning Curve**: Team members struggle to understand the framework.
- **Unnecessary Complexity**: The solution is far more complex than the problem it solves.

Antipatterns

An antipattern is a common solution to a recurring problem that may initially seem effective but ultimately leads to negative consequences, such as increased complexity, poor maintainability, or technical debt.

in specific scenarios, antipatterns can be pragmatic solutions, especially when speed and simplicity outweigh scalability and maintainability.

When to Use Antipatterns

- When speed and simplicity outweigh long-term maintainability.
- For one-off scripts or small-scale projects.
- When the team lacks experience with advanced patterns or principles.
- To quickly prove a concept without worrying about scalability.

When to Avoid Antipatterns

- When building long-term frameworks or projects with future growth.
- When multiple teams will work on the same codebase.
- When technical debt becomes unmanageable.

Sometimes, an antipattern is not a bad pattern—it's a practical one.

Combination Of Patterns

Combination Of Patterns

Handle different browsers with different configurations

Factory + Strategy

WebDriverFactory creates browser instances, Strategy handles browser-specific behaviors

Handle multiple test environments with different configurations

Singleton + Strategy + Factory

EnvironmentManager (Singleton) uses Factory to create configurations, Strategy for environment-specific behavior

Generate different report formats with custom content

Strategy + Builder + Observer

Strategy determines format, Builder creates content, Observer collects test results

Configure test suites based on different criteria (environment, tags, priority)

Builder + Strategy + Observer

SuiteBuilder creates suite, Strategy determines inclusion rules, Observer tracks execution

Handle different API types with different authentication and validation

Builder + Strategy + Factory

RequestBuilder creates requests, Strategy handles different API types, Factory creates appropriate clients

Handle tests across iOS/Android with different implementations

Factory + Strategy + Adapter

DriverFactory creates appropriate driver, Strategy handles platform-specific logic, Adapter normalizes commands

Appendix

More Design Patterns

Problem Statement

Imagine you're maintaining an automation script that needs to intercept and modify browser behavior, API calls, or database interactions for testing purposes. When a test connects directly to external services, it becomes difficult to test edge cases, simulate errors, or work offline. Over time, this can result in:

- Tests that are tightly coupled to external services
- Inability to test error scenarios
- No way to record or monitor interactions
- Difficult debugging when tests fail

Similar Use Cases

API Testing

You need to intercept requests to add authentication headers, mock responses for offline testing, or validate request data.

Browser Automation

You want to proxy WebDriver to record interactions, handle different browser configurations, or add custom logging.

Test Data Management

When accessing databases, you need to cache data, record operations, or simulate failures.

Security Testing

For security scenarios, you need to intercept HTTP headers, simulate certificates, or monitor data access.

Proxy Pattern

Pattern Category: **Structural Pattern**

Problem

When tests directly interact with external services:

- High coupling makes tests fragile
- Can't test error scenarios effectively
- No control over external dependencies
- Hard to debug test failures

Solution

The Proxy Pattern adds a surrogate object that controls access to another object. This allows:

- Access control Logging and monitoring
- Caching responses
- Mocking external services

The Proxy Pattern provides a surrogate object that controls access to another object. This pattern is particularly useful in test automation for creating test doubles, adding cross-cutting concerns, and controlling access to external resources.

Code Example

Bad Code https://tinyurl.com/cxbprbsz

Refactored Code https://tinyurl.com/28epsazb

Problem Statement

Imagine you're maintaining an automation script that performs different actions such as clicking buttons, entering text, and selecting options from dropdowns. Each action has its own specific implementation, and these actions need to be executed in different sequences across various tests. Manually managing this becomes difficult and results in:

- **Code Duplication**: Repeating similar logic (e.g., clicking, typing) across multiple tests.
- **Difficult to Maintain**: Changing the sequence or implementation of actions requires updating all related tests.
- **Tight Coupling**: Test scripts are tightly coupled to the action implementations, making them harder to modify or extend.

Similar Use Cases

UI Test Automation

Clicking buttons, entering text, selecting dropdowns, and scrolling need to be executed in various orders. Hardcoding these actions in each test creates redundancy.

API Test Execution

You may need to execute various API requests in a specific sequence (e.g., creating a user, fetching user details, deleting the user). Managing the request order manually makes it difficult to change or extend the test.

Undo/Redo Functionality

In some tests (e.g., verifying state transitions), you might need to support undoing or redoing actions, but manually handling this logic becomes complex.

User Action Recording

Recording user actions (like a click, drag, or type) and replaying them requires flexibility in executing these actions in different sequences.

Command Pattern

Pattern Category: **Behavioral Pattern**

Problem

Manually managing different actions (like clicking buttons, entering text, making API requests) in specific sequences leads to:

- Code Duplication
- High Maintenance Effort
- Tight Coupling

Solution

The solution is to encapsulate each action (e.g., button click, text entry) into a command object. The Command Pattern allows you to package these actions as objects, enabling you to execute them in any sequence, reuse them across different tests, and add support for undo/redo functionality if needed.

The Command Pattern encapsulates a request (e.g., click, text entry) as an object, allowing you to parameterize different requests, queue them, and execute them dynamically. This decouples the object that invokes the operation from the one that knows how to perform it, making the system more flexible and extendable.

Code Example

Bad Code https://tinyurl.com/mrd44dr3

Refactored Code https://tinyurl.com/9dsp44se

Problem Statement

Imagine you're working on a test automation framework where you need to perform multiple validations or checks in sequence. For example, before running a UI test, you need to:

- Check browser compatibility
- Verify test environment
- Validate test data
- Check required permissions

Each validation could stop the test or let it continue to the next check.

Similar Use Cases

- **Test Prerequisites** - Validate all conditions before test execution: environment, data, configuration.
- **API Response Validation** - Check response status, headers, schema, and business rules in sequence.
- **Data Sanitization** - Clean and validate test data through multiple steps before using it.
- **Retry Mechanism** - Try different recovery strategies in sequence when a test fails.

Chain of Responsibility Pattern

Pattern Category: **Behavioral Pattern**

Problem

When handling multiple validations directly in tests:

- Tests become complex with multiple if-else checks
- Hard to change validation order
- Duplicate validation code across tests
- Difficult to add new validations

Solution

Chain of Responsibility allows you to:

- Pass requests through a chain of handlers
- Each handler decides to process or pass along
- Decouple request sender from receivers
- Add or remove validations easily

Creates a chain of validator objects where each validator has a chance to handle the request or pass it to the next validator in the chain.

Code Example

Bad Code https://tinyurl.com/zwcnkpcw

Refactored Code https://tinyurl.com/dpc4rhh3

Problem Statement

Imagine you're working on a test automation project for a complex application with multiple domains—like an e-commerce platform. There are distinct modules for User Management, Orders, Payments, and Inventory. Each module has unique logic and business rules. However, as the application scales, so does the complexity of the test suite, leading to challenges:

- **Unstructured Test Suite:** Tests are scattered and lack clear boundaries for different business domains, making it hard to understand and maintain.
- **Disconnected from Business Logic:** The tests don't reflect the actual business workflows or terminology, leading to gaps in test coverage and misunderstandings among stakeholders.
- **Hard-Coded and Redundant Test Data:** Each test handles its data independently, often duplicating or hard-coding values, which makes the suite difficult to maintain.
- **Increased Maintenance Over Time:** As business rules and modules evolve, the automation suite requires constant refactoring, leading to high maintenance costs and error-prone updates.

Domain-Driven Design

Pattern Category: **Architectural Approach**

Problem

Without a structured approach like DDD, test automation in complex applications faces:

- **Disorganized Tests**: Tests are hard to manage, especially in large applications.
- **Communication Gaps**: Tests may not reflect business terminology, making it difficult for stakeholders to understand.
- **Hard-Coded Test Data**: Lack of organized test data management leads to duplicated or unrealistic data.

Solution

Domain-Driven Design (DDD) helps structure test automation around the business logic, providing an organized, maintainable framework. By structuring tests within business domains and aligning language and test data management, DDD makes the suite easier to understand and maintain.

DDD focuses on structuring and organizing complex software around business domains and real-world concepts, rather than prescribing specific object creation or behavioral interaction patterns.

Code Example

Bad Code https://tinyurl.com/3xcbk6n6

Refactored Code https://tinyurl.com/4kcy98k7

Thank You

Sridhar Patnaik

bsridharpatnaik@gmail.com

/bsridharpatnaik