— DESIGN PATTERNS —

IN ACTION

MASTERING THE STRATEGY PATTERN WITH 11 REAL-WORLD EXAMPLES

NIKOLAY NIKOLOV

Design Patterns in Action

Mastering the Strategy Pattern with 11 Real-World Examples

Nikolay Nikolov

Contents

About the author

Hi there! I'm Nikolay, and I've spent over 20 years diving deep into the world of software development. As the Head of Software Development at CONUTI GmbH and a Principal Software Engineer, I specialize in creating clean, maintainable, and well-organized code. Writing high-quality software isn't just my job; it's my passion.

Throughout my career, I've been a dedicated Clean Code Practitioner, always striving to promote and implement best practices in every project I touch. My journey has taken me through a wide array of technologies, including PHP, Symfony, Python, MySQL, Backend Services, Machine Learning, and RESTful WebServices.

Leading teams to build robust software architectures is something I excel at. I ensure that our projects are not just functional, but also scalable, performant, and reliable. Whether I'm working on backend services or machine learning, my goal is always the same: to deliver software solutions of the highest quality.

When I'm not coding, I love spending time with my wonderful wife, Dessy, and our sons, Nick and Eric. They keep me grounded and remind me why I love what I do.

Introduction

Over the years, I have observed the work of hundreds of programmers and concluded that only a small fraction truly knows how to write clean code from the beginning. Many tend to hack something together just to get it working, hoping they will have time to improve the code later.

However, any experienced software developer knows that this moment rarely comes. New feature requests, tasks, and bugs keep coming in, and there is seldom time to improve the existing code.

For this reason, I have been writing articles and books on the topic of clean code for some time now. My goal is to share this information from programmer to programmer. Even if it doesn't immediately help someone, it can at least inspire them to invest time in this subject, helping them to develop the qualities of a professional programmer. This, in turn, will make every piece of software a little better each day.

By promoting clean code practices and design patterns, I hope to encourage a culture of continuous improvement among developers. When we write clean, maintainable code, we not only make our own work easier but also contribute to the overall quality and longevity of the software we create. Every step taken towards cleaner code is a step towards better software and more efficient development processes.

Why Write Clean Code

So, why is it important to write clean code? Clean code allows for the rapid addition of new features and results in fewer bugs over time. Clean code is easier to test and read, which makes maintaining and extending the software much more manageable.

Enhanced Readability and Maintainability

Clean code is written with readability in mind. It uses clear, descriptive names for variables, functions, and classes, and follows consistent coding standards. This makes it easier for other developers (or even your future self) to understand the code quickly. When code is easy to read, it's also easier to maintain. Bug fixes and feature additions can be made with confidence, reducing the likelihood of introducing new issues.

Faster Development Cycles

When code is clean and well-organized, new features can be added more rapidly. Developers can navigate the codebase more efficiently, understand the existing functionality, and see where new code should be integrated. This reduces the time spent on understanding how the code works and allows more time for actual development.

Fewer Bugs and Easier Testing

Clean code is less prone to bugs because it follows clear and logical structures. Each function and module has a single responsibility, making it easier to isolate and test individual parts of the application. Automated tests can be written more easily for clean code, ensuring that new changes do not introduce regressions. This leads to more stable and reliable software over time.

Learning by Practice

To gain enough knowledge to master different design patterns, we have to practice as much as possible. That's why in this book, I have selected 11 different examples of the Strategy Pattern from various areas of software programming, and across different programming languages, to provide exercises for anyone willing to work through the examples.

By engaging with these practical exercises, you will be able to see how the Strategy Pattern can be applied in real-world scenarios, enhancing your ability to write clean and maintainable code. This hands-on approach ensures that you not only understand the theoretical aspects of design patterns but also develop the skills to implement them effectively.

Remember, mastering clean code and design patterns is a journey that requires continuous practice and dedication. Each example in this book is crafted to help you along this journey, making you a better, more proficient programmer.

Each example is structured in the same way, covering the following points:

1. **Introduction to the Problem**: A detailed description of the specific problem scenario.
2. **Bad Code Example**: A clear, detailed example of poorly written code that illustrates the problem.
3. **Understanding the Issues**: An explanation of the issues with the current implementation, such as rigidity, lack of scalability, and difficulty in maintenance.
4. **Step-by-Step Refactoring**: A breakdown of the refactoring process into clear, manageable steps, with code snippets and detailed explanations for each step.
5. **Benefits of the Refactored Code**: An analysis of the improvements and benefits of the refactored code, including any trade-offs or considerations.
6. **Exercises**: Practical exercises related to the chapter to reinforce learning, including practice problems for readers to apply the Strategy Pattern on their own.

11 Example Problem Scenarios

1. **Sorting Algorithms**
    - **Bad Code**: Hardcoded sorting algorithms.
    - **Refactored**: Use the Strategy pattern to select different sorting algorithms at runtime.
2. **Payment Processing Systems**
    - **Bad Code**: A monolithic payment processing function with conditional logic.

- **Refactored**: Use the Strategy pattern to handle different payment methods.

3. **File Compression**
    - **Bad Code**: A single class handling multiple compression algorithms.
    - **Refactored**: Apply the Strategy pattern to delegate compression tasks to different strategies.
4. **UI Themes**
    - **Bad Code**: A user interface with different themes (dark, light, custom) hardcoded.
    - **Refactored**: Use the Strategy pattern to apply different UI themes dynamically.
5. **Tax Calculation**
    - **Bad Code**: A tax calculation module with country-specific tax logic mixed together.
    - **Refactored**: Implement the Strategy pattern to handle different tax calculation strategies.
6. **Game Development (Character Behavior)**
    - **Bad Code**: Character behavior logic entangled in the main game loop.
    - **Refactored**: Use the Strategy pattern to define different behaviors for characters.
7. **Data Parsing**
    - **Bad Code**: A parser class that handles multiple data formats with conditional statements.
    - **Refactored**: Apply the Strategy pattern to parse different data formats dynamically.
8. **Notification Systems**
    - **Bad Code**: A notification system with various notification methods (email, SMS, push notifications) mixed together.
    - **Refactored**: Implement the Strategy pattern to handle different notification methods dynamically.
9. **E-commerce Promotions**
    - **Bad Code**: A promotion system with multiple promotion types (discount, buy-one-get-one, free shipping) mixed together.
    - **Refactored**: Implement the Strategy pattern to apply different promotion strategies dynamically.
10. **Data Encryption**
    - **Bad Code**: An encryption system with various encryption algorithms (AES, RSA, Blowfish) hardcoded.
    - **Refactored**: Use the Strategy pattern to encrypt data with different algorithms dynamically.

11. **Data Export**

    - **Bad Code**: A data export module with multiple export formats (CSV, Excel, JSON) entangled in one class.
    - **Refactored**: Use the Strategy pattern to export data in different formats dynamically.

Sorting Algorithms

Introduction to the Problem

Sorting is a common task in software development. In this example, we need a `Sorter` class capable of sorting arrays using different sorting algorithms, specifically Bubble Sort and Quick Sort. The initial implementation uses hardcoded conditional logic to choose the sorting algorithm.

Bad Code Example

The following Java code demonstrates a `Sorter` class that uses conditional logic to perform different types of sorting based on the provided sort type:

```java
public class Sorter {

    public enum SortType {
        BUBBLE_SORT,
        QUICK_SORT
    }

    public void sort(int[] array, SortType sortType) {
        if (sortType == SortType.BUBBLE_SORT) {
            bubbleSort(array);
        } else if (sortType == SortType.QUICK_SORT) {
            quickSort(array, 0, array.length - 1);
        } else {
            throw new IllegalArgumentException(
                "Unsupported sort type"
            );
        }
    }

    private void bubbleSort(int[] array) {
        int n = array.length;
        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - 1 - i; j++) {
                if (array[j] > array[j + 1]) {
                    int temp = array[j];
```

```
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                }
            }
        }
    }

    private void quickSort(int[] array, int low, int high) {
        if (low < high) {
            int pi = partition(array, low, high);
            quickSort(array, low, pi - 1);
            quickSort(array, pi + 1, high);
        }
    }

    private int partition(int[] array, int low, int high) {
        int pivot = array[high];
        int i = (low - 1);
        for (int j = low; j < high; j++) {
            if (array[j] <= pivot) {
                i++;
                int temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }
        int temp = array[i + 1];
        array[i + 1] = array[high];
        array[high] = temp;
        return i + 1;
    }
}
```

```
public class Caller {
    public static void main(String[] args) {
        Sorter sorter = new Sorter();
        int[] array = {5, 2, 9, 1, 5, 6};
        sorter.sort(array, Sorter.SortType.BUBBLE_SORT);
        System.out.println(
            "Bubble Sorted: " + java.util.Arrays.toString(array)
        );

        int[] array2 = {3, 8, 4, 6, 2, 7};
        sorter.sort(array2, Sorter.SortType.QUICK_SORT);
        System.out.println(
            "Quick Sorted: " + java.util.Arrays.toString(array2)
        );
    }
}
```