The Influence of Object-Oriented Programming Language and Class Libraries on Software Development: A Mixed-Methods Investigation

Nguyen Tran Quang Minh
104179687
Tutor class: 1.00pm Monday
Swinburne University of Technology
Hanoi, Vietnam
104179687@student.swin.edu.au

***Abstract*— This research investigates the impact of the Strategy design pattern on code maintainability and adaptability in the context of handling multiple algorithms in an application. A small software project was developed, implementing the application using two different approaches: one without using the Strategy design pattern (baseline) and another employing the Strategy pattern. Code metrics, including readability, modularity, flexibility, and adherence to OOP principles, were measured to assess code maintainability. Additionally, the adaptability of each implementation was evaluated by introducing new algorithms to the application. Through quantitative data analysis, this study reveals the strengths and limitations of the Strategy pattern in enhancing code maintainability and adaptability. The research provides practical recommendations for developers on the appropriate use of the Strategy design pattern to achieve more maintainable and flexible code solutions when dealing with multiple algorithms in software development.**

***Index terms*—Programming, Strategy Design Pattern, Code Metrics Results, CSharp**

I. INTRODUCTION

Object-Oriented Programming (OOP) design patterns are widely recognized as powerful tools that promote code organization, reusability, and maintainability [1]. Among these patterns, the Strategy design pattern stands out for its ability to handle multiple algorithms in an application by encapsulating each algorithm into a separate class and allowing clients to select and switch algorithms dynamically [2]. However, the practical impact of the Strategy pattern on code maintainability and adaptability remains a topic of interest for software developers.

This research aims to investigate the influence of the Strategy design pattern on code maintainability and adaptability when dealing with multiple algorithms in an application. The study aligns with the objectives of Learning Outcome 5, which emphasize describing factors contributing to a good object-oriented solution and drawing upon accepted good practices.

The research design involves the creation of a small software project representing a common application scenario where multiple algorithms need to be handled. Two different approaches will be implemented: one without using the Strategy pattern (baseline) and another applying the Strategy pattern. Code metrics, including readability, modularity, flexibility, and adherence to OOP principles, will be measured to assess code maintainability. Furthermore, the adaptability of each implementation will be evaluated by introducing new algorithms to the application.

By conducting this research, I seek to provide insights into the practical implications of adopting the Strategy design pattern in software development. The findings will help developers understand the strengths and limitations of the Strategy pattern in enhancing code maintainability and adaptability. Ultimately, the research aims to offer practical recommendations for effectively utilizing the Strategy pattern in handling multiple algorithms, contributing to creating more robust and maintainable software solutions. The following sections will present the research methodology, results, and discussion, providing a comprehensive understanding of the impact of the Strategy design pattern on code quality and flexibility in managing multiple algorithms in an application.

II. METHOD

A. Experiment Design

It was decided to conduct a controlled experiment to assess the impact of the Strategy design pattern on code maintainability and adaptability. A small software project was developed, representing a scenario where multiple algorithms must be handled in an application.

B. Implementation Approaches

Two different implementation approaches were chosen for the experiment. The first approach served as a baseline and did not utilize the Strategy design pattern. The second approach applied the Strategy pattern, encapsulating each algorithm into separate classes, allowing dynamic selection of algorithms by clients.

C. Development Environment

The project was implemented in an object-oriented programming language to ensure consistency and comparability between the two approaches.

D. Code Metrics:

To measure code maintainability, several code metrics were selected, including code readability, modularity, flexibility, and adherence to OOP principles. Static code analysis tools were employed to obtain quantitative data for these metrics.

E. Data Analysis:

The collected quantitative data were analyzed using appropriate statistical methods to compare code maintainability and adaptability between the two implementation approaches.

F. Code Snippet

Baseline Approach (Without Strategy Pattern) code snippet.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace COS20007_Research
{
    // Baseline approach without using the Strategy pattern

    public class ShoppingCart
    {
        private List<double> items = new List<double>();

        public void AddItem(double price)
        {
            items.Add(price);
        }

        public double CalculateTotal()
        {
            double total = 0;
            foreach (double price in items)
            {
                total += price;
            }
            return total;
        }
```

```
    }
}
```

Approach using the Strategy Pattern code snippet.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace COS20007_Research
{
    // Approach using the Strategy pattern

    // Strategy interface
    public interface IDiscountStrategy
    {
        double ApplyDiscount(List<double> items);
    }

    // Concrete strategy classes
    public class NoDiscountStrategy : IDiscountStrategy
    {
        public double ApplyDiscount(List<double> items)
        {
            return items.Sum();
        }
    }

    public class TenPercentDiscountStrategy : IDiscountStrategy
    {
        public double ApplyDiscount(List<double> items)
        {
            double total = items.Sum();
            return total - (total * 0.10);
        }
    }

    // Context class that uses the strategy
    public class ShoppingCartWithStrategy
    {
        private List<double> items = new List<double>();
        private IDiscountStrategy discountStrategy;

        public void SetDiscountStrategy(IDiscountStrategy strategy)
        {
            discountStrategy = strategy;
        }

        public void AddItem(double price)
        {
            items.Add(price);
        }
```

```
        public double CalculateTotal()
        {
            return discountStrategy.ApplyDiscount(items);
        }
    }
}
```

The Figure 1 shows the code that calculates the total price in two different approaches.

```
using COS20007_Research;

0 references
class Program
{
    0 references
    static void Main()
    {
        // Baseline approach
        ShoppingCart cart1 = new ShoppingCart();
        cart1.AddItem(50);
        cart1.AddItem(30);
        cart1.AddItem(20);
        double total1 = cart1.CalculateTotal();
        Console.WriteLine($"Total (Baseline): {total1}");

        // Approach using the Strategy pattern
        ShoppingCartWithStrategy cart2 = new ShoppingCartWithStrategy();
        cart2.AddItem(50);
        cart2.AddItem(30);
        cart2.AddItem(20);
        cart2.SetDiscountStrategy(new NoDiscountStrategy()); // Change strategy here
        double total2 = cart2.CalculateTotal();
        Console.WriteLine($"Total (Strategy Pattern): {total2}");
    }
}
```

Figure 1: Usage code snippet.

III. RESULTS

The code metrics analysis in Figure 2 demonstrated that the approach using the Strategy pattern exhibited higher code modularity, flexibility, and adherence to OOP principles, indicating improved code maintainability compared to the baseline approach without the pattern.

Code Metrics Results

Filter: Min: Max:

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code	Lines of Executable code
COS20007 Research (Debug)	85	10	1	8	96	26
Program	60	1	1	5	22	13
COS20007_Research	91	9	1	4	74	13
IDiscountStrategy	100	1	0	1	7	0
NoDiscountStrategy	96	1	1	4	8	1
ShoppingCart	85	3	1	1	21	6
ShoppingCartWithStrategy	93	3	1	2	21	4
TenPercentDiscountStrategy	83	1	1	4	8	2

Figure 2: Code Metrics Results.

IV. DISCUSSION

A. Code Metrics Results:

The analysis of code metrics provided valuable insights into the impact of the Strategy design pattern on code maintainability. The approach using the Strategy pattern demonstrated higher code modularity, flexibility, and adherence to OOP principles compared to the baseline approach without the pattern. The Strategy pattern allowed for encapsulating different algorithms in separate classes, promoting a more modular and flexible code structure. Additionally, it adhered to the Open/Closed Principle, enabling the application to easily accommodate new discount strategies without modifying existing code. These results indicate that the Strategy pattern positively influences code maintainability, making it a favorable choice for handling multiple algorithms in software development.

B. Recommendations and Limitations:

Based on the findings, it is recommended that developers consider adopting the Strategy design pattern when dealing with multiple algorithms in an application. Utilizing the Strategy pattern enhances code maintainability by promoting a cleaner, more modular, and flexible codebase [3]. It allows developers to isolate algorithm-specific code, making the application easier to understand and maintain. By employing the Strategy pattern, developers can effectively manage different algorithms and introduce new strategies without disrupting the existing codebase.

However, it is essential to acknowledge the limitations of this study. The research was conducted on a small-scale software project, and the results may vary for larger and more complex applications. Additionally, the Strategy pattern may introduce some overhead due to additional classes and abstractions. Therefore, developers should assess the specific needs of their projects and weigh the trade-offs before incorporating the Strategy pattern.

C. Future Research:

Future research in this area could explore the impact of the Strategy pattern in different application domains and larger software systems. Further investigations could consider other aspects of code quality, such as performance, scalability, and maintainability over extended development periods. Additionally, conducting experiments with more diverse participant groups and assessing the perception of developers about the Strategy pattern's practicality would provide additional insights. Exploring variations of the Strategy pattern or combining it with other design patterns could also be valuable for understanding its role in complex software architectures.

V. CONCLUSION

In conclusion, this research investigated the impact of the Strategy design pattern on code maintainability and adaptability when handling multiple algorithms in an application. Through a controlled experiment, two different implementation approaches were examined: one without using the Strategy pattern and another applying the pattern. The experiment focused on code metrics analysis and adaptability evaluation to assess the practical implications of the Strategy pattern.

The results indicated that the approach using the Strategy pattern exhibited higher code modularity, flexibility, and adherence to OOP principles, demonstrating improved code maintainability compared to the baseline approach. The Strategy pattern's ability to encapsulate algorithms in separate classes and facilitate dynamic algorithm selection contributed to a more modular and adaptable code structure. These findings highlight the Strategy pattern's significant role in creating maintainable and flexible software solutions.

As future work, further research could explore the applicability of the Strategy pattern in diverse application domains and larger-scale software projects. Investigating the Strategy pattern's impact on performance and scalability would provide a comprehensive understanding of its suitability in various scenarios. Additionally, exploring variations of the Strategy pattern or its combination with other design patterns could expand the knowledge base and contribute to the development of sophisticated software architectures. Overall, this research underscores the importance of design patterns in software development and emphasizes the value of the Strategy pattern as a powerful tool for enhancing code quality and adaptability.

References

[1] R. C. Martin, "Design principles and design patterns," *Object Mentor*, vol. 1, no. 34, p. 597, 2000.

[2] A. Christopoulou, E. A. Giakoumakis, V. E. Zafeiris, and V. Soukara, "Automated refactoring to the strategy design pattern," *Inf. Softw. Technol.*, vol. 54, no. 11, pp. 1202–1214, 2012.

[3] S. Herbold, J. Grabowski, and S. Waack, "Calculation and optimization of thresholds for sets of software metrics," *Empirical Softw. Eng.*, vol. 16, pp. 812–841, 2011.