The Strategy Design Pattern

CALVIN O. KERNS

Undergraduate Computer Science Student, Western Washington University, kernsc@wwu.edu

CCS CONCEPTS • Design Patterns • Object-oriented programming • Strategy Design Pattern

Design patterns are an essential part of software development and are responsible for the efficiency and organization that is seen in the technological world today. This paper explores the Strategy design pattern and its purpose to define a family of algorithms, encapsulate each one, and make them interchangeable at runtime. By examining what the design pattern is and the specifics of how it works, the benefits are clear. Illustrated are the effects of the Strategy pattern and some real-world examples of common implementations. The possible downsides of the pattern are also explored, although very few. The findings demonstrate that implementing the Strategy pattern not only simplifies code maintenance but also enables dynamic algorithm selection and promotes code reusability, ultimately leading to more flexible and maintainable software architectures.

1 INTRODUCTION

Design patterns are a way to simply lay out the process of software development to solve problems that may occur during the development process. These patterns provide a template that developers can use to solve recurring design challenges.

Design patterns are categorized into three main types: creational, structural, and behavioral. Creational patterns focus on object creation, mainly that they are created at the right time. Structural patterns deal with the relationship between creating classes and creating a structure of classes to be beneficial in terms of decoupling and organization. Behavioral patterns, on the other hand, target how specific objects interact and work together to get a common task done.

The significance of design patterns in modern software development can make or break the result of a project or application. As implementation grows in complexity and size, design patterns help to orginize and the planning and implementation process of software development. There are many known obstacles that developers have come across over the years and these patterns are the culmination of the knowledge and experience of these people in the past. Using design patterns will save developers the headache that comes with disorganization and poor modularity. On top of that, it provides a common structure for programs that is a universal language within the realm of object-oriented design. Due to this, programs are easier to create, maintain, and improve.

Within the behavioral category lies the Strategy design pattern. While each design pattern serves its unique purpose, the Strategy pattern stands out as a particularly powerful technique that addresses the common challenge of algorithm variation and selection in software systems. This pattern allows developers to seclude families of different algorithms for a similar purpose. This facilitates the pathway for variation of algorithms in runtime while allowing for straightforward additions of necessary algorithms in the future.

2 HOW THE STRATEGY DESIGN PATTERN WORKS

The Strategy design pattern is one of the central methods of encapsulating algorithms and promoting interchangeability within a program. At its core, Strategy allows for a program to pick from different concrete implementations of the same task during runtime. These concrete algorithms are different strategy classes that are all implementations of a similar interface. By having this common interface that strategy classes must abide by, the context and algorithms of a program are decoupled. This allows for interchangeability of algorithms during runtime.

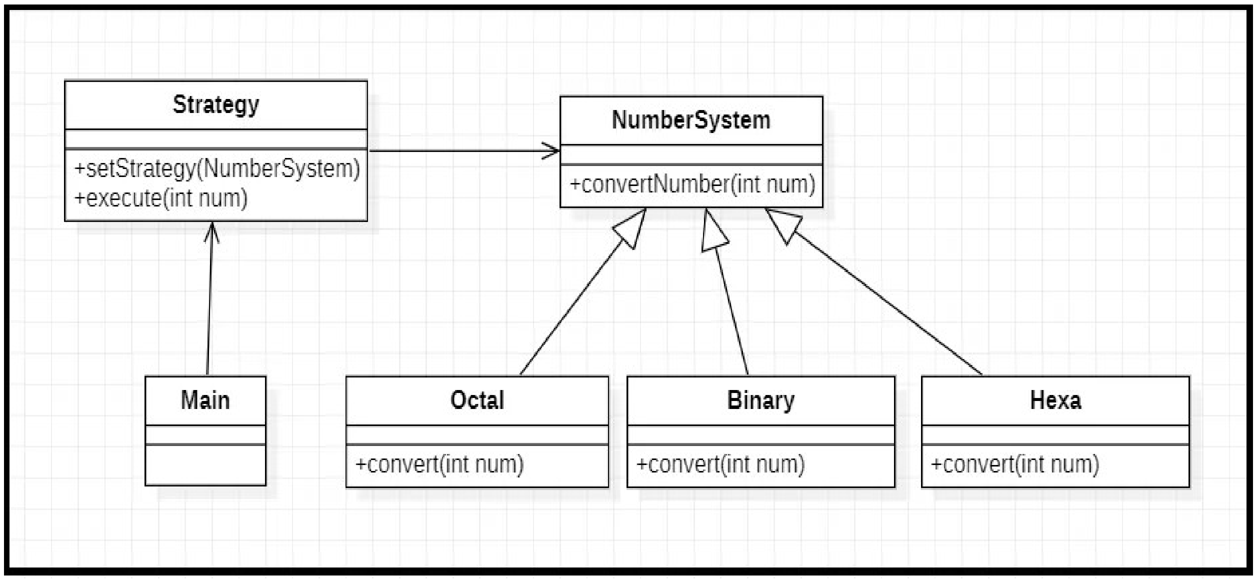

Figure 1: Ghosh, T. 2021. **Strategy Design Pattern**. *Medium*. (https://tusharghosh09006.medium.com/strategy-design-pattern-f12c3848e31c)

Figure 1 demonstrates an example familiar to all programmers. There are many different ways to represent numbers and sometimes the need to use one system over another is variable. The implementation of the Strategy design pattern illustrated shows the simplicity that the pattern provides when it comes to varying algorithms.

Octal, Binary, and Hexa classes each implement the same interface, NumberSystem(). Each implementation of the interface is required to provide a concrete implementation of the methods within the NumberSystem() interface, in this case, convertNumber(). The Main class has an object of type Strategy that has that has the method execute() to do some tasks given an integer. The input requirements for execute() may vary, hence the setStrategy() method. The client can use this method to swap between concrete implementations of the NumberSystem() during runtime. By doing this, the client running the main program simply can call the execute() method without worry about the implementation style of the different types of number systems. All the clients needs to know is which algorithm is expected for the program based off of the current context and can choose between all the implementations. This is an efficient and organized way to ensure that the execute() method receives its parameters in the proper form while also decoupling the implementation style of NumberSystem() and practical use of the of the class.

3 BENEFITS OF THE STRATEGY DESIGN PATTERN

As stated, the primary benefit of the Strategy design pattern is the ability to facilitate variable behavior at runtime unlike hardcoded algorithms or compile-time decisions. This ability is granted by the separation of concrete implementation of algorithms from the use of the algorithms in the main class. This is especially useful in systems that must adapt to changing conditions, user preferences, or varying optional requirements. “The pattern enables you to select the algorithm or behavior dynamically based on the specific context or requirements. To put it simply, the strategy pattern gives you the flexibility to choose the right strategy for a task, just like selecting the most suitable tool from a toolbox" [2].

Aside from providing ease of use for the client and adaptability in runtime, this design pattern greatly aids the process of maintenance and testing. Due to each strategy being implemented in a secluded concrete implementation class, the maintenance of each implementation is easier and has no affect on the rest of the program as long as it still fulfills it's required function and adheres to the interface's contract. Changes can be made to implementations, whether its for runtime efficiency or to simply write cleaning code, independently of other implementations of the algorithm or the main program functionality. The eliminates the risk of unintended changes causes bugs in the program. This also opens the door for addition algorithm implementations to be easily added to the current family of algorithms. Along with this comes the ability to independently test each algorithm. By being able to test in isolation, the algorithm, and therefore the program, will be more robust and reliable. "Strategies can be tested independently, simplifying the testing process and improving overall code quality" [3].

It is common in programs to see lengthy "if" statements. Although this works, it is hard to read, maintain, and test. It is often the case that long conditional statements are the root of logic errors. The Strategy design pattern eliminates this complexity by replacing conditional logic with polymorphism. This allows for developers to focus on implementation and maintaining the concrete implementation strategies. This also ties into the maintainability of the program. "The pattern promotes code maintainability by reducing the number of conditional statements and providing a clear separation between the Context and the Strategies" [3].

Along with the clear benefits for which the algorithms and their implementation were created, the option to reuse this code is available. Strategies, once implemented, can be reused across different contexts and applications. And this reusability extends beyond just the strategy implementations themselves. The well defined organization and modularity that the Strategy design pattern provides can serve as a template for other applications and any future development. When this aspect is taken advantage of, it results in saved development time and resources.

4 USE CASES

There are plenty of cases where the strategy design pattern will be crucial in having an effective system architecture. In most cases where the method of completing a task is variable, depending on the context of the situation or other factors, the decoupling of solution algorithms and the overall functionality of the system is required. That's where this design pattern comes in.

An example well known to most people is when it comes to navigation systems, such as google maps. There are numerous algorithms for calculating the fastest route locations. Some weigh traffic heavier while others weight distance heavier. The proper algorithm to use varies in a case by case basis and by using the Strategy pattern, it can interchangeable and will be most effect for users.

Another example could include sorting algorithms. There are a wide variety of these algorithms and they each have specific situations where they are prominent, especially in the case of best/worst case runtimes. The ability to apply any number of sorting algorithms in runtime is a large factor in a lot of software, specifically graphics, memory allocation, and other applications.

In short, if a program has aspects or functions that are variable to change in implementation, this design pattern will be essential to create that flexibility while maintaining functionality.

5 COSTS

While the Strategy design is effective in many ways, it is not always necessary. "If there is only one fixed algorithm that will be used throughout the lifetime of the application, and there is no need for dynamic selection or switching between algorithms, using the Strategy pattern might introduce unnecessary complexity" [4]. This is not often the case for larger systems, but if it ever is the case the this design pattern will only end up costing the development team time.

As stated before, the Strategy design pattern is dependent on the context of the program, thus it is effective in making changes during runtime. But if the program has a static context and is assuredly not changing in a way that will demand algorithmic change, the benefits of this design are not received. "If the context class tightly depends on a single algorithm and there is no need for flexibility or interchangeability, using the Strategy pattern may introduce unnecessary abstraction and complexity" [4].

6 CONCLUSION

The Strategy design pattern presents as a powerful solution for managing algorithmic complexity in modern software systems. It is clear that the pattern's ability to encapsulate algorithms and make them interchangeable at runtime provides significant advantages in software development. The resulting decoupling of algorithms and the context they are used in is the core behind enabling the dynamic behavior and facilitating the maintainable simplicity of the system.

The benefits of implementing the Strategy pattern extend beyond the code organization. By facilitating independent testing, reducing conditional complexity, and enabling runtime flexibility, the pattern addresses several critical challenges in software development. The practical applications discussed, such as navigation systems and sorting algorithms, illustrate how the pattern can be effectively applied to solve real-world problems.

While the pattern does present some drawbacks, particularly in scenarios with static algorithmic requirements, these limitations are generally outweighed by the pattern's benefits in most modern software systems, especially large ones.

The Strategy design pattern is a useful tool that should reside in every developers arsenal. It exponentially improves the organization and quality of applications.

REFERENCES:

[1] Tushar Ghosh. 2021. Strategy Design Pattern. Medium. Retrieved November 30, 2024 from https://tusharghosh09006.medium.com/strategy-design-pattern-f12c3848e31c.

[2] Thanos Karropoulos. 2021. A Deep Dive Into the Strategy Design Pattern. Dev.to. Retrieved November 30, 2024 from https://dev.to/tkarropoulos/a-deep-dive-into-the-strategy-design-pattern-5i5#:~:text=The%20pattern%20enables%20you%20to,suitable%20tool%20from%20a%20toolbox.

[3] Huawei Developers. 2021. Strategy Design Pattern: A Powerful Tool for Flexible Code. Medium. Retrieved November 30, 2024 from https://medium.com/huawei-developers/strategy-design-pattern-a-powerful-tool-for-flexible-code-fe6b4c9f47b5.

[4] GeeksforGeeks. 2021. Strategy Pattern - Set 1. GeeksforGeeks. Retrieved November 30, 2024 from https://www.geeksforgeeks.org/strategy-pattern-set-1/.