**Why Strategy Pattern guarantee the scalability and reusability of your code?**

**Introduction**

Imagine writing a code that cannot be modified to accommodate changes in the future. Won’t that risk your code becoming obsolete in the future? In other words, one should always keep in mind that change is the only constant thing in software development, and by acknowledging it, one saves your code from becoming obsolete in the future. Attaining scalability and reusability of code makes it adaptable to ever-changing requirements, hence becoming easy to not only handle increasing loads but also modify and reuse it across different contexts. This blog focuses on how to use the Strategy Pattern over other conditional statements to write scalable and reusable code.

**Conditional statements**

Switch statement, If-Else control, and Ternary operator are the most common conditional statements. Let us assume a client approaches you to create an app that allows customers to choose “Cash,” “PayPal,” “Stripe,” and “Debit card” as their payment method. Let us assume you decide to use either of the conditional statements above to write the code as follows:

**Switch statement code**

A screen shot of a computer program

AI-generated content may be incorrect.

**If-Else control code**

A screen shot of a computer program

AI-generated content may be incorrect.

**Ternary operator code**

A screen shot of a computer

AI-generated content may be incorrect.

**How do Switch statement, If-Else control, and Ternary operator threaten the future of your code?**

Now, we can see how different methods of payment can be implemented using Switch statement, If-Else control, and Ternary operator, respectively. He is the challenge. Suppose the client comes back to you to add another payment method after receiving several complaints from customers who want **“Apple Pay,” “Google Pay,” “Samsung Pay,” “Amazon Pay,” “Venmo,” and “Zelle” to be included in the payment method options. In the business world,** "the customer is always right" is a term often used to address the customer’s concerns to win their loyalty. By the way, who wants to lose a customer? **The client now wants you to include the missing methods of payment as suggested by the customers. If you stick to using either of the conditional statements used above, you must amend the base code to add the missing payment method. Do you remember SOLID principles? Yes, I mean** a set of five design principles that help make object-oriented software more maintainable, flexible, and scalable. Any modification of the base code violates this principle. So, what’s next? Here is where the “Strategy design” becomes a savior. A strategy pattern can be utilized in JavaScript in many situations to write code that observes SOLID principles, since it allows the code for each condition to remain independent and separately testable. See the different versions of the code to implement different payment methods using the Strategy pattern.

**Strategy pattern**

A screenshot of a computer program

AI-generated content may be incorrect.

In the Strategy pattern implementation above, adding more payment methods can be done without modifying the base code, hence addressing the problem of the Open/Closed principle, which you could not achieve using a Switch statement, an If-Else control, and a Ternary operator. All you will need to do is add the functions for each additional payment method under the strategy without interfering with the base code. See how the code looks after the missing payment methods.

A screen shot of a computer program

AI-generated content may be incorrect.

From the code above, we haven’t touched the function choosePaymentMethod(), which is the base of this code, but rather added the new payment method under the strategy. However, if you were to use Switch statement, If-Else control, and Ternary operator, you are compelled to touch the base code. So, in this blog, I aimed to answer the question “**Why Strategy Pattern guarantee the scalability and reusability of your code**? Nevertheless, do not misinterpret my question as discrediting these conditional statements. They are simple to write, consume less space, and work fine for simple conditions, but they have limitations.

**Conclusion**

Having gone through the reason for choosing Strategy pattern below is a table to summarize problems of using Switch statement, If-Else control, and Ternary operator, and how Strategy pattern solves them.

|  |  |  |
| --- | --- | --- |
| **Problem** | **Using If/Else, Switch & Ternary** | **Solved by Strategy pattern** |
| Scalability | Becomes bulky and hard to manage with many conditions | Each strategy is in its own class or function, making it modular |
| Open/Closed Principle | You must modify the control structure to add new behavior | You just add a new strategy without changing existing code |
| Reusability | Logic is embedded in one place – hard to reuse | Each strategy is reusable and testable individually |
| Testing | Hard to isolate behaviors for unit tests | Easy to test each strategy independently |
| Decoupling | Tight coupling between control logic and business logic | Decouples behavior from context (caller) |