**Research design pattern**

**Strategy Design pattern：**

Aim：The main purpose of the policy pattern is to make it easier to add additional functionality and features to existing code while minimizing changes to existing code. It allows the flexibility to select and replace specific behaviors at runtime based on preset requirements. The motivation for using the policy pattern is that it follows the "open-closed principle" that code should be open to extensions, but closed to modifications. In this way, the policy pattern makes it easy to extend and add new policies without changing the policy interface. [1]

Advantage: The main benefits of using the policy pattern include increased code flexibility and reusability. With this mode, the algorithm can be changed or extended without modifying the client code, reducing the risk of introducing errors when modifying. In addition, the policy pattern also implements the separation of concerns, delegating the execution of the algorithm to the policy class, making the Context class more simplified and focused on its core responsibilities. Individual policies can be tested independently, enhancing the testability of your code. Finally, the policy pattern conforms to the open-close principle and can easily introduce new policies without affecting the existing system, especially for systems with frequent changes in requirements. [2]

Practical application：Design pattern is used to solve problems in industrial applications, especially the practical application of strategic design pattern in accounting framework. In the field of accounting, tax calculation is a key problem, in order to solve this problem, the strategy design model is chosen. By using policy patterns, the framework has the flexibility to handle different computational logic, especially in complex and potentially changing scenarios such as tax calculations, and policy patterns provide a way to dynamically select and replace algorithms. In addition, industrial applications often require multiple interfaces to interact with data storage and business logic. The practical application of the policy pattern here is to simplify these interactions, avoiding creating different interfaces for each purpose and instead interacting with the application's data and logic by defining common interfaces. In this way, the policy pattern helps simplify complex transaction processing and multi-resource coordination, ensuring that applications can effectively interact with system layers such as databases and middle tiers. [3]

**Decorate design pattern**

Aim: The goal of the decorator design pattern is to dynamically increase the behavior of objects by creating new classes (i.e., decorator classes) without changing the original class structure. This pattern is suitable for extending the function of objects, especially when the class hierarchy is complex, and can reduce the problem of subclassing.[4]

Advantage: The Decorator design pattern is an object-oriented solution that dynamically adds functionality to an object at run time without changing the object's class structure. One advantage of this pattern is that it can be implemented in any object-oriented programming language, as it does not rely on additional plug-ins or frameworks to complete its implementation, operation, and execution.[5] Design patterns in object-oriented design and programming effectively improve the performance of reusability mechanisms. Reusability is achieved through inheritance, and the decorator pattern provides a powerful alternative to inheritance. Using the decorator pattern avoids some of the problems associated with inheritance, especially the need to create a large number of subclasses to support each combination. Therefore, we chose the decorator mode to provide more flexibility in implementation and use.[4]

Practical application: Software applications that adopt design patterns can improve performance and facilitate maintenance. Design patterns enable software design to adapt more flexibly to changing requirements while improving the overall performance of the application. In our practice, the decorator pattern has been applied to Web applications based on HTML and C# languages. This pattern allows us to dynamically add new state behavior to objects at run time, giving us the flexibility to create decorator instances on demand. This provides a significant advantage for scenarios where additional functionality is needed and memory usage may increase.[6]

**Case study analyses**

**Strategy Design pattern：**

**Practical application:** The strategy mode is applied in the Ele. me software app. Users can choose different payment methods, such as credit card, PayPal or Alipay. Each payment method has a different implementation logic. By encapsulating each payment method as an independent policy object, the system dynamically selects the appropriate payment strategy according to the user's choice at runtime, without modifying the core code of the payment system.

**Challenge solving:** The problem of complexity can be solved, there are many hidden states in a software program that can constitute security traps, but in the strategy design pattern, each policy can be tested independently of context and other policies, which makes it easier to find and locate hidden errors. At the same time, since each strategy is independent, when we add other strategies, we do not have a negative impact on the existing strategy. It is also very maintainable, and changes to one policy do not affect other policies. strategy design pattern can also be used to solve changeability. Software often changes constantly due to changing requirements. Code that applies strategy design pattern can easily be changed or added to a strategy class.

**Decorate design pattern**

**Practical application:** In real life, some software that allows users to customize their coffee uses Decorate design pattern. They allow people to add ingredients to their coffee dynamically, such as sugar or milk, which may cause confusion.

**Challenge solving:** The decorator design pattern can effectively solve the problems of "maintainability" and "flexibility" mentioned in "No Silver Bullet" in practical application. For example, in graphical user interface (GUI) development, by using the decorator pattern, functionality can be dynamically added without modifying existing code, thus avoiding the situation of class bloat. The decorator pattern avoids the complexity of inheritance and increases the maintainability of the system by encapsulating functionality into separate decorator classes. As requirements change, developers can dynamically change the behavior of components by adding or removing decorators without modifying the core code, which not only reduces the effort of modification and debugging, but also improves the adaptability of the system. In addition, the decorator pattern avoids the problem of code duplication when functionality is extended by inheritance, and by combining decorators, the ability to select different functions on demand without creating multiple subclasses makes the code more concise and flexible.

Reference

[1] M. Nagaappan, "Strategy Design Pattern," \*Cheriton School of Computer Science, University of Waterloo\*, 1995. [Online]. Available: https://cs.uwaterloo.ca/~m2nagapp/courses/CS446/1195/Arch\_Design\_Activity/Strategy.pdf. [Accessed: Apr. 5, 2025].

[2] E. Shah, "Understanding the Strategy Design Pattern," \*Medium\*, Jul. 28, 2024. [Online]. Available: https://medium.com/@eshikashah2001/understanding-the-strategy-design-pattern-in-software-engineering-8774086a1895. [Accessed: Apr. 5, 2025].

[3] R. Bala and K. K. Kaswan, "Strategy Design Pattern," \*Global Journal of Computer Science and Technology: C Software & Data Engineering\*, vol. 14, no. 6, pp. 1-6, 2014. [Online]. Available: https://globaljournals.org/. [Accessed: Apr. 5, 2025].

[4] F. B. Bastien, A. K. Gupta, and M. K. Gupta, "A Design Pattern Approach to Improve the Structure and Implementation of the Decorator Design Pattern," \*Res. J. Appl. Sci. Eng. Technol.\*, vol. 13, no. 5, pp. 416–421, 2016. [Online]. Available: https://www.researchgate.net/publication/308118364\_A\_Design\_Pattern\_Approach\_to\_Improve\_the\_Structure\_and\_Implementation\_of\_the\_Decorator\_Design\_Pattern. [Accessed: Apr. 5, 2025].

[5] C. Pereira-Vásquez, C. Vidal-Silva, E. Madariaga, C. Jiménez, and L. Urzúa, "Modulating Crosscutting Concerns By The Decorator Design Pattern Vs. Aspect Oriented Programming In .NET," \*Int. J. Sci. Technol. Res.\*, vol. 9, no. 1, pp. 3862–3866, Jan. 2020. [Online]. Available: https://www.ijstr.org/final-print/jan2020/Modulating-Crosscutting-Concerns-By-The-Decorator-Design-Pattern-Vs-Aspect-Oriented-Programming-In-net.pdf. [Accessed: Apr. 5, 2025].

[6] V. K. Kerji, "Decorator Pattern in Web Application," \*Int. J. Adv. Inf. Technol.\*, vol. 3, no. 4, pp. 14–18, Aug. 2013. [Online]. Available: https://airccse.org/journal/IJAIT/papers/3413ijait02.pdf. [Accessed: Apr. 5, 2025].