**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.