|  |
| --- |
| **Mẫu thiết kế:**  **Factory Method** |
| **Mục đích sử dụng:**   * Dùng để tạo MỘT đối tượng thuộc một họ các đối tượng * Không quan tâm chi tiết đến quá trình tạo ra đối tượng * Không quan tâm cụ thể loại đối tượng nào được tạo ra |
| **Những trường hợp áp dụng mẫu:**   * Đơn giản hóa quá trình tạo đối tượng ở phía client * Mong muốn tạo ra một đối tượng mà chi tiết quá trình tạo ra nó có thể rắc rối và phức tạp * Việc sử dụng đối tượng được tạo không cần phân biệt giữa các loại đối tượng cùng họ |
| **Cấu trúc mẫu chuẩn:**   * Product: Đây là lớp cha hoặc giao diện chung của các đối tượng cần tạo. Tất cả các lớp con sẽ phải triển khai các phương thức của lớp cha này. * ConcreteProduct: Đây là các lớp con triển khai các đối tượng cụ thể mà Factory Method sẽ tạo ra. * Creator: Đây là lớp cha hoặc giao diện chung của Factory Method. Lớp này cung cấp một phương thức chung để tạo đối tượng Product, nhưng không cần biết cụ thể các đối tượng cụ thể nào sẽ được tạo ra. * ConcreteCreator: Đây là các lớp con triển khai Factory Method, có nhiệm vụ tạo ra các đối tượng cụ thể. Mỗi lớp con sẽ triển khai lại phương thức tạo đối tượng của Creator để tạo ra các ConcreteProduct khác nhau. |
| **Các biến thể (nếu có):**   * Abstract Factory Method: Một biến thể của Factory Method, trong đó Factory Method được sử dụng để tạo ra một tập hợp các đối tượng liên quan đến nhau, thường là trong một nhóm đối tượng cùng một lĩnh vực hoặc liên quan đến một chức năng cụ thể. Abstract Factory Method cung cấp một giao diện chung để tạo ra các đối tượng này, cho phép chương trình tạo ra các đối tượng một cách linh hoạt, tùy thuộc vào yêu cầu của hệ thống. * Static Factory Method: Một phương thức tĩnh (static method) trong lớp, được sử dụng để tạo ra đối tượng mà không cần tạo ra một thể hiện (instance) của lớp. Static Factory Method thường được sử dụng để đóng gói các thao tác tạo đối tượng phức tạp hoặc để đảm bảo tính duy nhất (singleton) của đối tượng. * Singleton Factory Method: Một biến thể của Static Factory Method, được sử dụng để tạo ra một đối tượng duy nhất trong toàn bộ chương trình. Singleton Factory Method đảm bảo rằng chỉ có một thể hiện (instance) của đối tượng được tạo ra, và cung cấp một giao diện để truy cập đối tượng này từ bất kỳ đâu trong chương trình. * Factory Method với Dependency Injection: Một biến thể của Factory Method, trong đó các phương thức tạo đối tượng được sử dụng để tạo ra các đối tượng có sự phụ thuộc vào các đối tượng khác. Dependency Injection giúp giảm thiểu sự phụ thuộc giữa các lớp trong hệ thống, đồng thời giúp tăng tính linh hoạt và tái sử dụng của mã nguồn. |
| **Những dấu hiệu nhận biết mẫu:**   * Tạo đối tượng được thực hiện trong một phương thức trừu tượng (abstract method) hoặc một phương thức của một lớp trừu tượng (abstract class). * Trong phương thức tạo đối tượng, có sử dụng từ khóa "new" để tạo đối tượng. * Trong phương thức tạo đối tượng, trả về một đối tượng (object) hoặc một danh sách đối tượng (list of objects) được tạo ra từ các tham số đầu vào. * Các lớp con (subclass) được sử dụng để triển khai phương thức tạo đối tượng, và tạo ra các đối tượng cụ thể (concrete objects) của lớp cha (parent class). * Các lớp con có thể được tạo ra và sử dụng một cách độc lập mà không ảnh hưởng đến các lớp khác trong hệ thống. |
| **Những lưu ý khác (nếu có):**   * Đảm bảo tên phương thức tạo đối tượng có ý nghĩa và dễ hiểu, để giúp người đọc dễ dàng nhận biết tác dụng và chức năng của phương thức. * Tạo ra một lớp trừu tượng để chứa phương thức tạo đối tượng, để giảm thiểu sự phụ thuộc giữa các lớp. * Sử dụng phương pháp tạo đối tượng thông qua Factory Method để giảm thiểu sự phụ thuộc giữa các lớp, đồng thời tăng tính linh hoạt và tái sử dụng của mã nguồn. * Sử dụng Abstract Factory Method để tạo ra một tập hợp các đối tượng liên quan đến nhau, thường là trong một nhóm đối tượng cùng một lĩnh vực hoặc liên quan đến một chức năng cụ thể. * Tránh sử dụng quá nhiều lớp con, vì điều này có thể dẫn đến sự phức tạp của hệ thống. * Sử dụng các phương pháp khác để tạo đối tượng, như Static Factory Method hoặc Singleton Factory Method, nếu phù hợp với yêu cầu của hệ thống. * Sử dụng Dependency Injection để giảm thiểu sự phụ thuộc giữa các lớp trong hệ thống. * Sử dụng các công cụ kiểm thử để đảm bảo tính đúng đắn của hệ thống khi sử dụng mẫu Factory Method. * Cẩn thận khi sử dụng các tham số đầu vào trong phương thức tạo đối tượng, để tránh sự ràng buộc về kiểu dữ liệu (type constraints) hoặc lỗi xảy ra trong quá trình chuyển đổi kiểu dữ liệu (type conversion). * Chú ý đến hiệu suất của hệ thống khi sử dụng mẫu Factory Method, vì việc tạo đối tượng có thể tốn kém thời gian và tài nguyên của máy tính. |
| **Ưu điểm:**   * Tách biệt quá trình tạo đối tượng và sử dụng đối tượng: Factory Method giúp tách biệt quá trình tạo đối tượng khỏi quá trình sử dụng đối tượng. Điều này có nghĩa là bạn không cần phải biết chi tiết cụ thể về cách tạo đối tượng, mà chỉ cần biết cách sử dụng chúng. * Mã linh hoạt: Với Factory Method, bạn có thể thay đổi loại đối tượng được tạo một cách linh hoạt chỉ bằng cách thay đổi Factory Method. Do đó, mã của bạn trở nên linh hoạt và dễ dàng mở rộng. * Tính mở rộng cao: Factory Method cho phép dễ dàng mở rộng chức năng của chương trình chỉ bằng cách tạo ra các lớp mới hoặc chỉnh sửa lớp hiện có mà không ảnh hưởng đến toàn bộ hệ thống. * Tính đa hình: Factory Method cho phép tạo ra các đối tượng khác nhau có cùng interface, giúp tăng tính đa hình của hệ thống. * Giảm sự phụ thuộc: Factory Method giúp giảm sự phụ thuộc giữa các lớp trong hệ thống bằng cách đưa quyết định về việc tạo đối tượng vào trong các lớp Factory Method. Điều này giúp mã nguồn trở nên dễ dàng bảo trì và phát triển. |
| **Hạn chế:**   * Tăng độ phức tạp của mã: Sử dụng Factory Method có thể làm tăng độ phức tạp của mã do cần phải tạo thêm các lớp Factory Method để tạo đối tượng. * Tạo ra quá nhiều lớp: Sử dụng Factory Method có thể dẫn đến việc tạo ra quá nhiều lớp trong hệ thống. Điều này có thể làm tăng độ phức tạp và giảm hiệu suất của ứng dụng. * Khó hiểu với những người mới học: Factory Method có thể làm cho mã nguồn khó hiểu đối với những người mới học do cần phải hiểu rõ về các lớp Factory Method và cách chúng tạo ra các đối tượng. * Tăng chi phí phát triển: Sử dụng Factory Method có thể làm tăng chi phí phát triển ứng dụng do phải tạo nhiều lớp mới để tạo đối tượng. * Đôi khi không cần thiết: Factory Method không phải là giải pháp cho tất cả các vấn đề liên quan đến tạo đối tượng. Trong một số trường hợp, nó có thể làm tăng độ phức tạp của mã mà không cải thiện được tính linh hoạt của ứng dụng. |

|  |
| --- |
| **Mẫu thiết kế:**  **Abstract Factory** |
| **Mục đích sử dụng:**   * Tạo ra các đối tượng có liên quan đến nhau một cách dễ dàng và linh hoạt hơn. * Cho phép tách biệt quá trình tạo đối tượng với cách triển khai cụ thể của chúng. * Giúp giảm thiểu sự phụ thuộc giữa các lớp trong hệ thống. * Tăng khả năng mở rộng và bảo trì của hệ thống, do không cần sửa đổi quá nhiều mã nguồn khi thêm mới hoặc thay đổi các đối tượng trong một nhóm. * Có thể sử dụng Abstract Factory để giúp đảm bảo tính chuyên nghiệp và chất lượng của hệ thống, bằng cách tạo ra các đối tượng đã được kiểm chứng và đảm bảo tính đúng đắn của chúng. * Tạo ra các đối tượng với các thuộc tính cấu hình khác nhau, tùy thuộc vào nhu cầu sử dụng của hệ thống. * Cho phép chúng ta dễ dàng thay đổi triển khai của các đối tượng trong một nhóm, mà không ảnh hưởng đến các đối tượng khác trong nhóm. |
| **Những trường hợp áp dụng mẫu:**   * Khi bạn cần tạo ra một tập hợp các đối tượng có liên quan đến nhau hoặc cùng thuộc một nhóm. * Khi bạn muốn đảm bảo rằng các đối tượng được tạo ra phù hợp với các quy tắc của một loại cụ thể và không phải loại khác. * Khi bạn muốn tách biệt logic tạo đối tượng khỏi mã khách hàng của mình. * Khi bạn muốn thay đổi loại các đối tượng được tạo ra mà không cần thay đổi mã khách hàng. * Khi bạn muốn mở rộng mã của mình để hỗ trợ các loại đối tượng mới mà không làm ảnh hưởng đến mã hiện có. * Khi bạn muốn cung cấp một giao diện chung để tạo các đối tượng liên quan đến nhau mà không cần chỉ định loại cụ thể của đối tượng được tạo ra. |
| **Cấu trúc mẫu chuẩn:**   * AbstractFactory: là một interface hoặc abstract class chứa các phương thức tạo (create) các đối tượng liên quan đến nhau. * ConcreteFactory: là các lớp cụ thể kế thừa AbstractFactory và triển khai các phương thức tạo để tạo ra các đối tượng cụ thể. * AbstractProduct: là một interface hoặc abstract class đại diện cho các sản phẩm có liên quan đến nhau. * ConcreteProduct: là các lớp cụ thể kế thừa AbstractProduct và triển khai các phương thức của nó. * Client: là lớp sử dụng các đối tượng được tạo ra bởi AbstractFactory và AbstractProduct. |
| **Các biến thể (nếu có):**   * Parametrized Abstract Factory: trong đó, các phương thức tạo của AbstractFactory nhận tham số đầu vào để xác định loại đối tượng được tạo ra. * Generic Abstract Factory: trong đó, AbstractFactory và AbstractProduct được thay thế bằng các tham số kiểu để tạo ra một loại factory có thể tạo ra bất kỳ loại sản phẩm nào. * Dependency Injection: trong đó, đối tượng AbstractFactory được truyền vào Client thông qua dependency injection, giúp giảm sự phụ thuộc của Client vào AbstractFactory. * Singleton Abstract Factory: trong đó, một AbstractFactory được đảm bảo chỉ có một thể hiện duy nhất được tạo ra trong toàn bộ ứng dụng, giúp đảm bảo sự liên tục và thống nhất cho các đối tượng được tạo ra. * Fluent Interface: trong đó, các phương thức tạo (create) trả về chính nó hoặc một sản phẩm liên quan đến đối tượng được tạo ra, cho phép gọi tiếp tục phương thức khác liên quan đến đối tượng đó. |
| **Những dấu hiệu nhận biết mẫu:**   * Có một Interface hoặc Abstract class để định nghĩa phương thức tạo các đối tượng trong một họ. Chúng ta gọi đây là Abstract Factory. * Các Concrete Factory được tạo ra từ Abstract Factory để triển khai trực tiếp các phương thức tạo đối tượng. Các Concrete Factory này loại hình đối tượng liên quan đến cùng một họ. * Các Concrete Product được tạo ra từ Concrete Factory, tạo ra các đối tượng cụ thể trong cùng một loại hình học. Mỗi Concrete Product tương ứng với một loại đối tượng cụ thể trong một họ. * Client được tạo ra để sử dụng các đối tượng được tạo bởi Abstract Factory. Client này sẽ tương tác với Abstract Factory thay vì với các Concrete Factory hoặc Concrete Product. * Các phương thức tạo đối tượng trong Abstract Factory không nhận bất kỳ tham số nào, trong khi các phương thức tạo đối tượng trong Concrete Factory có thể nhận đối số để tạo ra các đối tượng cụ thể. |
| **Những lưu ý khác (nếu có):**   * Đảm bảo sự tương thích giữa các sản phẩm được tạo ra: các sản phẩm được tạo ra bởi Abstract Factory phải tương thích với nhau, tức là có thể hoạt động chung trong một ứng dụng. * Không chứa logic phức tạp trong Abstract Factory: Abstract Factory chỉ nên chứa các phương thức tạo (create), và không nên chứa logic phức tạp hoặc các thuật toán xử lý dữ liệu. * Sử dụng interface hoặc abstract class: các lớp cụ thể của Abstract Factory và Abstract Product nên được triển khai thông qua interface hoặc abstract class để giảm độ phụ thuộc của các lớp với nhau. * Cân nhắc sử dụng Singleton: việc sử dụng Singleton Abstract Factory nên được cân nhắc kỹ lưỡng để tránh các vấn đề liên quan đến đa luồng (multi-threading) hoặc bảo mật. * Sử dụng cùng với các mẫu khác: mẫu Abstract Factory có thể được kết hợp với các mẫu khác như Singleton, Builder hoặc Prototype để tạo ra các ứng dụng linh hoạt và dễ bảo trì. * Không nên sử dụng quá nhiều: mẫu Abstract Factory nên được sử dụng trong những trường hợp cần thiết và không nên được sử dụng quá nhiều, vì điều này có thể làm tăng độ phức tạp của mã và làm giảm tính linh hoạt của ứng dụng. |
| **Ưu điểm:**   * Có thể đảm bảo các đối tượng product nhận được từ factory sẽ tương thích với nhau. * Tránh được những ràng buộc chặt chẽ giữa concrete products và client code. * Nguyên tắc đơn lẻ: có thể trích xuất code tạo product vào một nơi và hỗ trợ code dễ dàng. * Nguyên tắc mở/ đóng: có thể khởi tạo những bản mới của product mà không cần phá vỡ client code hiện có. |
| **Hạn chế:**   * Code có thể trở nên phức tạp hơn mức bình thường, vì có rất nhiều interfaces và classes được khởi tạo cùng với mẫu. |

|  |
| --- |
| **Mẫu thiết kế:**  **Builder** |
| **Mục đích sử dụng:**   * Để tách rời quá trình xây dựng một đối tượng phức tạp ra khỏi việc tạo đối tượng, giúp cho việc xây dựng đối tượng trở nên linh hoạt hơn và dễ dàng hơn. |
| **Những trường hợp áp dụng mẫu:**   * Xây dựng đối tượng phức tạp: Mẫu Builder được sử dụng để xây dựng các đối tượng phức tạp, gồm nhiều thuộc tính và các đối tượng khác được liên kết với nhau. * Tách rời quá trình xây dựng đối tượng: Mẫu Builder giúp tách rời quá trình xây dựng đối tượng ra khỏi việc tạo đối tượng, cho phép các phần khác nhau của quá trình xây dựng đối tượng được thực hiện độc lập. * Tạo ra nhiều biến thể của đối tượng: Mẫu Builder giúp tạo ra nhiều biến thể của một đối tượng phức tạp mà không làm tăng độ phức tạp của mã nguồn. * Cải thiện tính tái sử dụng: Mẫu Builder giúp cải thiện tính tái sử dụng của mã nguồn, cho phép tái sử dụng các phần xây dựng đối tượng phức tạp mà không cần phải viết lại hoặc chỉnh sửa mã nguồn. * Tăng tính linh hoạt: Mẫu Builder giúp tăng tính linh hoạt của mã nguồn, cho phép các phần khác nhau của quá trình xây dựng đối tượng được thay đổi hoặc thêm vào một cách dễ dàng. |
| * **Cấu trúc mẫu chuẩn:** * *Product* : đại diện cho đối tượng cần tạo, đối tượng này phức tạp, có nhiều thuộc tính. * *Builder* : là abstract class hoặc interface khai báo phương thức tạo đối tượng. * *ConcreteBuilder* : kế thừa Builder và cài đặt chi tiết cách tạo ra đối tượng. Nó sẽ xác định và nắm giữ các thể hiện mà nó tạo ra, đồng thời nó cũng cung cấp phương thức để trả các các thể hiện mà nó đã tạo ra trước đó. * *Director*: là nơi sẽ gọi tới Builder để tạo ra đối tượng. |
| **Các biến thể (nếu có):**   * Builder Pattern với Director: Trong biến thể này, một đối tượng Director được sử dụng để chỉ đạo việc xây dựng đối tượng, trong khi một hoặc nhiều đối tượng Builder được sử dụng để thực hiện các bước xây dựng cụ thể cho từng phần của đối tượng. * Fluent Builder Pattern: Biến thể này cho phép xây dựng đối tượng một cách dễ dàng và trực quan hơn bằng cách sử dụng các phương thức liên tiếp để thiết lập các thuộc tính của đối tượng. * Immutable Builder Pattern: Trong biến thể này, đối tượng được xây dựng là không thể thay đổi sau khi được tạo ra. Điều này đảm bảo tính an toàn và độ tin cậy của đối tượng. * Parameterized Builder Pattern: Biến thể này cho phép xây dựng đối tượng bằng cách sử dụng các tham số, giúp đơn giản hóa việc xây dựng đối tượng và cải thiện tính linh hoạt của mã nguồn. * Step Builder Pattern: Trong biến thể này, các bước xây dựng đối tượng được thực hiện thông qua các đối tượng Builder riêng lẻ, giúp tách rời quá trình xây dựng ra khỏi việc tạo đối tượng và cải thiện tính linh hoạt của mã nguồn. * Generic Builder Pattern: Biến thể này cho phép xây dựng đối tượng với các kiểu dữ liệu khác nhau, giúp tăng tính đa dạng và sử dụng lại mã nguồn. |
| **Những dấu hiệu nhận biết mẫu:**   * Khi cần xây dựng một đối tượng phức tạp, với nhiều thuộc tính và có thể có những tùy chọn khác nhau. * Khi muốn tách rời quá trình xây dựng đối tượng ra khỏi việc tạo đối tượng, giúp dễ dàng quản lý và bảo trì mã nguồn. * Khi muốn đảm bảo tính linh hoạt của mã nguồn, giúp cho việc thay đổi các thuộc tính của đối tượng trở nên dễ dàng mà không cần phải thay đổi nhiều mã nguồn. * Khi cần xử lý các trường hợp đặc biệt, ví dụ như đối tượng được tạo ra phải đảm bảo tính an toàn, độ tin cậy cao, hoặc là đối tượng không thể thay đổi sau khi được tạo ra. * Khi muốn tạo ra một đối tượng theo nhiều cách khác nhau, tùy thuộc vào các tùy chọn hoặc yêu cầu khác nhau từ phía người sử dụng. |
| **Những lưu ý khác (nếu có):**   * Xác định rõ yêu cầu cần thiết cho việc xây dựng đối tượng, bao gồm các thuộc tính, các bước xử lý, và các ràng buộc liên quan đến đối tượng. * Định nghĩa một giao diện Builder chung để định nghĩa các phương thức xây dựng đối tượng. * Xây dựng các lớp con Builder để hiện thực hóa giao diện Builder và định nghĩa các phương thức để thiết lập các thuộc tính của đối tượng. * Xây dựng một lớp Director để giám sát và điều phối quá trình xây dựng đối tượng. * Thực hiện kiểm tra lỗi và bảo mật cho các thuộc tính của đối tượng. * Đảm bảo tính linh hoạt của mã nguồn bằng cách cho phép thay đổi các thuộc tính của đối tượng mà không cần thay đổi nhiều mã nguồn. * Kiểm tra và tối ưu hóa mã nguồn để đảm bảo hiệu suất và khả năng mở rộng. * Sử dụng mẫu Builder kết hợp với các mẫu khác để tăng tính linh hoạt và giảm sự phức tạp của mã nguồn. * Chú ý đến việc thiết kế giao diện và tên phương thức để tạo ra mã nguồn dễ hiểu và dễ bảo trì. * Nên sử dụng mẫu Builder khi xây dựng đối tượng phức tạp có nhiều thuộc tính và các bước xử lý phức tạp, và khi cần tách rời quá trình xây dựng đối tượng ra khỏi việc tạo đối tượng. |
| **Ưu điểm:**   * Cho phép tách rời quá trình xây dựng đối tượng ra khỏi việc tạo đối tượng, giúp giảm sự phức tạp của mã nguồn và tăng tính linh hoạt. * Các đối tượng được xây dựng bởi Builder có tính kế thừa cao, giúp tăng tính mở rộng và khả năng tái sử dụng mã nguồn. * Builder giúp tăng tính kiểm soát và bảo mật cho các thuộc tính của đối tượng, tránh được việc các thuộc tính bị sửa đổi bởi các phương thức khác. * Giúp tăng tính hiệu quả khi xây dựng các đối tượng phức tạp có nhiều thuộc tính và các bước xử lý phức tạp, giúp giảm sự phụ thuộc vào các phương thức khác. * Cho phép thay đổi các thuộc tính của đối tượng mà không cần thay đổi nhiều mã nguồn, giúp tăng tính linh hoạt và dễ bảo trì của mã nguồn. * Khi kết hợp với các mẫu khác, Builder có thể giúp tăng tính linh hoạt và giảm sự phức tạp của mã nguồn. * Builder có thể giúp tách biệt quá trình xây dựng đối tượng ra khỏi việc sử dụng đối tượng, giúp giảm sự phụ thuộc và tăng khả năng mở rộng. * Giúp tăng khả năng kiểm soát mã nguồn và tạo ra mã nguồn dễ hiểu và dễ bảo trì. * Builder cho phép xây dựng đối tượng theo cách tùy ý, giúp tăng tính đa dạng và tùy chỉnh của đối tượng. * Mẫu Builder được sử dụng rộng rãi trong các ứng dụng thương mại điện tử, trong các hệ thống quản lý nội dung, và các hệ thống quản lý dữ liệu, giúp tăng tính linh hoạt và hiệu quả của các ứng dụng. |
| **Hạn chế:**   * Đối với các đối tượng đơn giản, việc sử dụng mẫu Builder có thể làm cho mã nguồn trở nên phức tạp và không cần thiết. * Việc xây dựng mẫu Builder có thể mất thời gian và tốn kém chi phí, đặc biệt là khi xây dựng các đối tượng phức tạp có nhiều thuộc tính và các bước xử lý phức tạp. * Mẫu Builder có thể tạo ra quá nhiều lớp và mối quan hệ giữa các lớp, làm tăng sự phức tạp của mã nguồn. * Việc xây dựng các đối tượng thông qua Builder có thể không linh hoạt, do phải tuân thủ một quy trình cụ thể trong quá trình xây dựng. * Việc xây dựng các đối tượng thông qua Builder có thể làm giảm tính đa dạng và tùy chỉnh của đối tượng. * Việc sử dụng mẫu Builder không phù hợp cho các ứng dụng có tính chất thay đổi liên tục và không định hướng trước được các thuộc tính của đối tượng. |

|  |
| --- |
| **Mẫu thiết kế:**  **Singleton** |
| **Mục đích sử dụng:**  Đảm bảo rằng chỉ có một đối tượng duy nhất được tạo ra từ một lớp và đối tượng đó có thể truy cập được từ bất kỳ đâu trong ứng dụng |
| **Những trường hợp áp dụng mẫu:**   * đảm bảo rằng chỉ có một đối tượng duy nhất được tạo ra từ một lớp và đối tượng đó có thể truy cập được từ bất kỳ đâu trong ứng dụngĐảm bảo chỉ có một đối tượng được tạo ra để quản lý tài nguyên chung trong toàn bộ ứng dụng, ví dụ như đối tượng quản lý cơ sở dữ liệu, nhật ký, bộ đếm, ... * Đảm bảo rằng các cấu trúc dữ liệu hoặc trạng thái được chia sẻ giữa các đối tượng khác nhau trong ứng dụng. * Đảm bảo việc sử dụng tài nguyên được tối ưu hóa trong trường hợp tài nguyên đó có giới hạn. * Đảm bảo truy cập đến đối tượng duy nhất và tránh việc tạo ra nhiều đối tượng khi không cần thiết. |
| **Cấu trúc mẫu chuẩn:**  Singleton:  Public class Singleton{  Private static final Singleton *instance* = new Singleton();  Private Singleton(){}  Public static Singleton getInstance(){  Return *instance*;  }  } |
| **Các biến thể (nếu có):**   1. Eager initialization: Trong biến thể này, đối tượng được tạo ra ngay khi lớp được tải vào bộ nhớ, thay vì chờ đến khi đối tượng được yêu cầu lần đầu tiên. Khi sử dụng cách tiếp cận này, đối tượng Singleton sẽ được tạo ra ngay khi chương trình khởi động, dẫn đến việc tiêu tốn tài nguyên nếu đối tượng không được sử dụng. 2. Lazy initialization: Trong biến thể này, đối tượng được tạo ra khi có yêu cầu sử dụng lần đầu tiên. Khi sử dụng cách tiếp cận này, đối tượng Singleton chỉ được tạo ra khi cần thiết, giúp giảm thiểu sự tiêu tốn tài nguyên, nhưng đồng thời làm tăng thời gian phản hồi cho lần đầu tiên sử dụng. 3. Double-checked locking: Đây là một biến thể của lazy initialization, trong đó kiểm tra lock được sử dụng để đảm bảo rằng chỉ có một đối tượng Singleton được tạo ra, đồng thời vẫn giữ tính chất "lazy". Biến thể này cần cẩn thận để tránh các vấn đề liên quan đến đồng bộ hóa. 4. Thread-local storage: Trong biến thể này, mỗi luồng có một phiên bản riêng của đối tượng Singleton, đảm bảo rằng các đối tượng được tạo ra độc lập và độc nhất.   Enum-based Singleton: Đây là một biến thể độc đáo của Singleton Pattern, trong đó đối tượng Singleton được định nghĩa như một hằng số enum. Đây là cách tiếp cận an toàn nhất và đơn giản nhất để triển khai Singleton Pattern trong Java. |
| **Những dấu hiệu nhận biết mẫu:**   * Constructor của lớp Singleton là private: Điều này ngăn cản việc tạo ra đối tượng Singleton từ bên ngoài lớp. * Lớp Singleton có một biến toàn cục: Biến toàn cục này được sử dụng để lưu trữ đối tượng Singleton. * Lớp Singleton có một phương thức tĩnh, thường có tên là getInstance(): Phương thức này được sử dụng để truy cập đối tượng Singleton. Nếu đối tượng chưa được tạo ra, phương thức này sẽ tạo ra đối tượng và trả về nó. * Phương thức getInstance() của lớp Singleton sử dụng lazy initialization hoặc eager initialization: Trong lazy initialization, đối tượng Singleton được tạo ra khi có yêu cầu sử dụng lần đầu tiên. Trong eager initialization, đối tượng Singleton được tạo ra ngay khi lớp được tải vào bộ nhớ. * Lớp Singleton có tính chất "thread-safe": Điều này đảm bảo rằng chỉ có một đối tượng Singleton được tạo ra, ngay cả khi có nhiều luồng sử dụng cùng lúc. * Lớp Singleton có khả năng serialization: Điều này đảm bảo rằng đối tượng Singleton có thể được lưu trữ trong bộ nhớ đệm hoặc truyền qua mạng. * Lớp Singleton có khả năng kế thừa: Nếu lớp Singleton được kế thừa, đối tượng Singleton sẽ vẫn duy trì tính chất "duy nhất" trong toàn bộ chương trình. |
| **Những lưu ý khác (nếu có):**   * Sử dụng đúng mục đích: Singleton Pattern phù hợp khi bạn muốn đảm bảo rằng chỉ có một đối tượng duy nhất của một lớp được tạo ra trong toàn bộ chương trình. Tuy nhiên, nếu một lớp không cần thiết phải đảm bảo tính duy nhất, việc sử dụng Singleton Pattern có thể không cần thiết và gây ra sự phức tạp không cần thiết cho mã nguồn. * Sử dụng phương thức getInstance() với cẩn thận: Phương thức getInstance() của Singleton Pattern là phương thức tĩnh, do đó không thể kế thừa. Việc sử dụng getInstance() trong mã nguồn phải được cẩn thận để tránh gây ra rắc rối khi thêm hoặc sửa đổi mã nguồn sau này. * Không sử dụng Singleton Pattern cho các đối tượng không thể tái sử dụng: Singleton Pattern không phù hợp cho các đối tượng không thể tái sử dụng, ví dụ như các đối tượng đại diện cho kết nối cơ sở dữ liệu hoặc các tài nguyên hệ thống. Đối với các trường hợp này, bạn nên sử dụng một mẫu thiết kế khác như Object Pool Pattern. * Đảm bảo tính thread-safe: Nếu Singleton Pattern được sử dụng trong môi trường đa luồng, bạn phải đảm bảo rằng mã nguồn của bạn là thread-safe. Điều này đảm bảo rằng chỉ có một đối tượng Singleton được tạo ra, ngay cả khi có nhiều luồng sử dụng cùng lúc. * Không sử dụng Singleton Pattern để truyền dữ liệu giữa các lớp: Singleton Pattern không nên được sử dụng để truyền dữ liệu giữa các lớp. Thay vào đó, bạn nên sử dụng Dependency Injection hoặc mẫu thiết kế khác như Observer Pattern. |
| **Ưu điểm:**   * có thể chắc chắn rằng mỗi class chỉ có một instance duy nhất. * có thể truy cập instance ở bất cứ đâu và bất cứ khi nào * Singleton chỉ khởi tạo khi bạn gọi chúng lần đầu tiên (gọi khi nào khởi tạo khi ấy). |
| **Hạn chế:**   * Vi phạm nguyên tắc Single Responsibility Principle – nguyên tắc đơn nhiệm, một pattern giải quyết cùng lúc 2 vấn đề. * Mẫu Singleton có thể ẩn đi những thiết kế xấu cho instance khi các component trong chương trình biết rõ về nhau. * Singleton yêu cầu xử lý đặc biệt trong mộ trường đa luồng, để nhiều luồng không tạo ra một đối tượng Singleton nhiều lần. * Singleton thường xuyên bị các lập trình viên lâu năm đánh giá là “Evil”, vì Singleton tạo ra quá nhiều phụ thuộc, không thể sử dụng đa hình và dễ tạo ra các bug làm họ phải [debug](https://wiki.tino.org/debug-la-gi/) “thâu đêm”. |

|  |
| --- |
| **Mẫu thiết kế:**  **Adapter** |
| **Mục đích sử dụng:**   * giúp các đối tượng có thể làm việc với nhau một cách linh hoạt và dễ dàng hơn bằng cách đưa chúng vào một lớp trung gian (adapter) để xử lý sự khác biệt về interface giữa chúng. |
| **Những trường hợp áp dụng mẫu:**   * sử dụng một đối tượng có sẵn nhưng không thể sử dụng trực tiếp vì không tương thích với interface của client. * tái sử dụng một đối tượng có sẵn, nhưng interface của nó không phù hợp với yêu cầu hiện tại. * kết nối giữa các đối tượng có sẵn nhưng không tương thích với nhau. |
| **Cấu trúc mẫu chuẩn:**  Client:  Method(data)  Adapter extend exist class from client:  Method(data)  Service:  ServiceMethod(specialData) |
| **Các biến thể (nếu có):**   1. Class Adapter Pattern: Trong biến thể này, adapter class sử dụng kế thừa để kết nối giữa interface của client và class không tương thích. Adapter class kế thừa interface của client và class không tương thích cùng một lúc và chuyển đổi các phương thức của class không tương thích để phù hợp với interface của client. 2. Object Adapter Pattern: Trong biến thể này, adapter class sử dụng composition để kết nối giữa interface của client và class không tương thích. Adapter class bao gồm một đối tượng của class không tương thích và implement interface của client, và chuyển đổi các phương thức của class không tương thích để phù hợp với interface của client. |
| **Những dấu hiệu nhận biết mẫu:**   * Có hai interface tách biệt nhau và không tương thích với nhau, và client chỉ có thể sử dụng một trong hai. * Một class đã tồn tại, nhưng không tương thích với interface của client, và không thể sửa đổi class đó. * Một class mới cần được sử dụng để kết nối giữa interface của client và class không tương thích. * Adapter class cung cấp một cách để chuyển đổi từ interface của class không tương thích sang interface của client. * Adapter class có thể được sử dụng để kết nối giữa nhiều class không tương thích và một interface của client. * Client không biết về sự tồn tại của class không tương thích, chỉ biết về adapter class. |
| **Những lưu ý khác (nếu có):**   * Adapter pattern thường được sử dụng khi bạn không thể sửa đổi class không tương thích để phù hợp với interface của client. * Nên sử dụng biến thể Object Adapter Pattern thay vì Class Adapter Pattern, vì Object Adapter Pattern là một cách linh hoạt hơn để kết nối giữa nhiều class không tương thích và một interface của client. * Nên đặt tên adapter class sao cho nó mô tả rõ mục đích của nó. * Nên tách biệt adapter class ra khỏi client code để giảm sự phụ thuộc giữa các thành phần và dễ dàng bảo trì, nâng cấp sau này. * Không nên sử dụng adapter pattern quá nhiều, vì nó có thể dẫn đến một số vấn đề như tăng thêm độ phức tạp, giảm hiệu suất và làm tăng số lượng code. Nên sử dụng adapter pattern chỉ khi cần thiết và khi không có giải pháp nào tốt hơn. * Nếu class không tương thích cần được sửa đổi, hãy cân nhắc sử dụng cấu trúc mẫu Bridge pattern thay vì Adapter pattern. * Có thể kết hợp Adapter pattern với các cấu trúc mẫu khác để tạo ra các giải pháp tốt hơn. Ví dụ: Decorator pattern có thể được sử dụng để thêm các chức năng bổ sung vào adapter class. |
| **Ưu điểm:**   * Single Responsibility Principle: Có thể tách interface hoặc các đoạn code chuyển đổi dữ liệu khỏi logic nghiệp vụ chính của chương trình * Open/Closed Principle: Giúp code không bị ảnh hưởng từ các thay đổi hoặc các lần cập nhật phiên bản mới từ API hoặc dịch vụ từ bên thứ ba (thay đổi tên hàm, tên lớp,…) |
| **Hạn chế:**   * Độ phức tạp tổng thể của mã tăng lên vì bạn cần giới thiệu/khai báo một tập hợp các interface và lớp mới. Đôi khi, việc thay đổi lớp service sao cho phù hợp với phần còn lại của mã của bạn sẽ đơn giản hơn. |

|  |
| --- |
| **Mẫu thiết kế:**  **Composite** |
| **Mục đích sử dụng:**   * Cho phép xây dựng một cây phân cấp các đối tượng một cách động và linh hoạt * Thao tác với các đối tượng trong cây dễ dàng và trực quan, mà không cần biết chúng là đối tượng đơn lẻ hay là một cấu trúc phức tạp |
| **Những trường hợp áp dụng mẫu:**   * Mô hình hóa các đối tượng có cấu trúc phức tạp dưới dạng cây phân cấp. * Thực hiện các thao tác đồng nhất trên các đối tượng trong cây, bao gồm cả đối tượng đơn lẻ và cấu trúc phức tạp. * Cho phép người dùng thao tác với các đối tượng trong cây một cách trực quan, dễ hiểu và không cần biết chi tiết về cấu trúc của chúng. * Thực hiện các thao tác trên tất cả các thành phần trong cây một cách đồng thời và linh hoạt, bao gồm cả thêm, sửa và xóa các đối tượng. |
| **Cấu trúc mẫu chuẩn:**  Cấu trúc mẫu Composite trong Design pattern bao gồm các thành phần sau:  Component: Định nghĩa một giao diện chung cho tất cả các đối tượng trong cấu trúc. Đây là thành phần cơ bản nhất trong mẫu Composite và có thể là một interface hoặc một abstract class.  Leaf: Đại diện cho các đối tượng lá trong cấu trúc, tức là các đối tượng không có thành phần con. Leaf thực hiện giao diện được xác định bởi Component.  Composite: Đại diện cho các đối tượng có chứa các thành phần con. Composite cài đặt các phương thức của Component và sử dụng chúng để quản lý các thành phần con của nó.  Client: Sử dụng cấu trúc Composite bằng cách gọi các phương thức của Component để thực hiện các hoạt động trên cấu trúc.  Ví dụ:  public interface Component {  void operation();  }  public class Leaf implements Component {  @Override  public void operation() {  // Do something  }  }  public class Composite implements Component {  private List<Component> components = new ArrayList<>();  @Override  public void operation() {  for (Component component : components) {  component.operation();  }  }  public void addComponent(Component component) {  components.add(component);  }  public void removeComponent(Component component) {  components.remove(component);  }  }  public class Client {  public static void main(String[] args) {  Component leaf1 = new Leaf();  Component leaf2 = new Leaf();  Composite composite1 = new Composite();  composite1.addComponent(leaf1);  composite1.addComponent(leaf2);  Component leaf3 = new Leaf();  Component leaf4 = new Leaf();  Composite composite2 = new Composite();  composite2.addComponent(leaf3);  composite2.addComponent(leaf4);  composite2.addComponent(composite1);  composite2.operation();  }  } |
| **Các biến thể (nếu có):**  Không có biến thể nào của Composite trong Design pattern. Composite chỉ có một cấu trúc mẫu chuẩn, không có các biến thể khác như Bridge hay Factory Method. |
| **Những dấu hiệu nhận biết mẫu:**   * Có một cấu trúc phân cấp đối tượng và chúng có cùng giao diện. * Phương thức gọi đệ quy được sử dụng để truy xuất các đối tượng con. * Phương thức thêm hoặc xóa đối tượng con được định nghĩa ở cả đối tượng lá và đối tượng gốc. * Có thể truy xuất đến các đối tượng lá hoặc gốc một cách độc lập hoặc đệ quy. * Phương thức tính toán có thể đệ quy qua toàn bộ cây đối tượng, tính toán tổng hợp từ các giá trị con. * Có thể sử dụng đa hình để xử lý các đối tượng lá và gốc một cách thống nhất. |
| **Những lưu ý khác:**   * Khi sử dụng Composite, cần chú ý đến việc quản lý memory, đặc biệt là khi xóa các phần tử trong cây. Việc xóa một nút có thể dẫn đến việc xóa các nút con của nó nếu chúng không được quản lý đúng cách. * Composite thường được sử dụng để xử lý dữ liệu phân cấp, ví dụ như cây thư mục, menu trang web, hoặc các biểu đồ phân cấp. * Composite có thể được kết hợp với các mẫu khác như Decorator, Iterator hoặc Visitor để tăng tính linh hoạt và hiệu quả của ứng dụng. * Khi thiết kế một hệ thống sử dụng Composite, cần đảm bảo rằng các đối tượng con và đối tượng lá đều phải tuân thủ các giao diện hoặc lớp trừu tượng chung. Điều này giúp cho việc xử lý các đối tượng này trở nên dễ dàng và linh hoạt hơn. |
| **Ưu điểm:**   * Dễ dàng thêm mới các thành phần và nhóm thành phần: Composite giúp cho việc thêm mới các thành phần và nhóm thành phần trở nên dễ dàng hơn bao giờ hết. Nó giúp cho các đối tượng của chúng ta có thể được tổ chức theo cấu trúc phân cấp, và chúng ta có thể thêm mới các thành phần và nhóm thành phần một cách linh hoạt, tùy thuộc vào yêu cầu của từng ứng dụng cụ thể. * Dễ dàng truy cập các thành phần: Composite cho phép chúng ta truy cập các thành phần cấu trúc phân cấp theo cách dễ dàng hơn. Chúng ta có thể truy cập các thành phần của một đối tượng Composite theo cách giống như chúng ta truy cập các thuộc tính của một đối tượng bình thường. * Giảm thiểu số lượng mã lặp: Composite giúp giảm thiểu số lượng mã lặp. Với mẫu Composite, chúng ta có thể tạo ra một số lượng lớn các đối tượng có cấu trúc tương tự nhau, và chúng ta không cần phải viết lại các phương thức và thuộc tính của chúng. * Dễ dàng xử lý các thành phần đối tượng: Composite giúp cho việc xử lý các thành phần đối tượng trở nên dễ dàng hơn. Với mẫu Composite, chúng ta có thể xử lý các thành phần đối tượng trong một cấu trúc phân cấp bằng các phương thức đệ quy, giúp cho việc xử lý trở nên đơn giản và hiệu quả hơn. |
| **Hạn chế:**   * Đôi khi, việc sử dụng mẫu Composite có thể dẫn đến hiệu suất kém. Vì mỗi đối tượng Composite phải duyệt qua tất cả các thành phần của nó để thực hiện một số thao tác, điều này có thể gây ra một lượng lớn các hoạt động duyệt trong trường hợp các cây Composite lớn. * Việc cài đặt mẫu Composite có thể trở nên phức tạp hơn so với một giải pháp đơn giản hơn nếu số lượng các thành phần trong cây là cố định và không thay đổi. * Việc định nghĩa quy tắc duyệt một cây Composite phải được đảm bảo để tránh các vấn đề về lỗi và hiệu suất. Nếu không, việc duyệt một cây có thể trở nên rất khó khăn và có thể dẫn đến việc thực hiện các tác vụ không mong muốn. * Mẫu Composite có thể không phù hợp cho các hệ thống có cấu trúc phức tạp và có nhiều quan hệ phức tạp giữa các đối tượng khác nhau. Trong trường hợp này, mẫu Composite có thể không thể hiện được các liên kết phức tạp giữa các thành phần của hệ thống và làm cho việc quản lý và bảo trì trở nên khó khăn hơn. |

|  |
| --- |
| **Mẫu thiết kế:**  **Decorator** |
| **Mục đích sử dụng:**   * Mở rộng tính năng của đối tượng: Decorator cho phép thêm các tính năng mới cho một đối tượng mà không ảnh hưởng đến cấu trúc của đối tượng đó. * Dễ dàng thêm và xóa tính năng: Bằng cách sử dụng Decorator, chúng ta có thể dễ dàng thêm hoặc xóa tính năng của một đối tượng bằng cách tạo và xóa các lớp bọc mà không cần sửa đổi các đối tượng gốc * Tăng tính linh hoạt của hệ thống: Decorator cho phép kết hợp nhiều lớp bọc với nhau để tạo ra các tính năng phức tạp hơn, do đó tăng tính linh hoạt của hệ thống |
| **Những trường hợp áp dụng mẫu:**   * Thêm tính năng động * Thay đổi tính năng * Xử lý dữ liệu đầu vào và đầu ra * Kiểm soát truy cập vào đối tượng * Thêm tính năng theo yêu cầu |
| **Cấu trúc mẫu chuẩn:**   * Component: định nghĩa giao diện chung cho các đối tượng trong cấu trúc. * ConcreteComponent: định nghĩa các đối tượng cơ bản (basic objects) để được bao bọc. * Decorator: định nghĩa một giao diện cho các decorator và giữ một tham chiếu đến một đối tượng Component. * ConcreteDecorator: thêm chức năng cho đối tượng cơ bản, bằng cách đóng gói và gọi các phương thức từ Component và thêm chức năng mới. |
| **Các biến thể (nếu có):**   1. Decorator Concrete: Lớp Concrete Decorator thực hiện các chức năng bổ sung cho đối tượng bao bọc. 2. Component: Định nghĩa interface chung cho các đối tượng được bao bọc và các Decorator. 3. Concrete Component: Lớp Concrete Component cung cấp triển khai cho interface Component. 4. Decorator Abstract: Lớp Abstract Decorator định nghĩa interface cho các lớp Decorator và giữ một tham chiếu đến Component. 5. Decorator Concrete A và B: Các lớp Concrete Decorator cung cấp các tính năng bổ sung cho Component. Chúng kế thừa từ Abstract Decorator và cũng giữ tham chiếu đến Component. 6. Client: Định nghĩa đối tượng cần được bao bọc. |
| **Những dấu hiệu nhận biết mẫu:**   * Thiết kế Decorator thường liên quan đến việc thêm hoặc bỏ các tính năng cho đối tượng mà không làm thay đổi cấu trúc của đối tượng đó. * Mẫu Decorator thường sử dụng kỹ thuật bọc đối tượng bên trong một lớp mới để cung cấp các tính năng bổ sung cho đối tượng đó. * Decorator Pattern định nghĩa một lớp Decorator trừu tượng, chứa một tham chiếu đến một đối tượng được bao bọc (Component), và triển khai các phương thức từ interface của Component. * Mỗi lớp Decorator cụ thể thực hiện các tính năng bổ sung mới và/hoặc thay đổi tính năng của đối tượng được bao bọc, trước khi gọi phương thức tương ứng của đối tượng được bao bọc. * Mẫu Decorator thường có sử dụng đệ quy, khi một lớp Decorator có thể bao bọc một lớp Decorator khác. |
| **Ưu điểm:**   * Tính linh hoạt: Decorator cho phép bạn thêm hoặc bỏ các tính năng của đối tượng mà không làm thay đổi cấu trúc của đối tượng đó. Điều này giúp bạn tùy chỉnh các đối tượng để đáp ứng yêu cầu cụ thể của hệ thống mà không phải thay đổi các đối tượng gốc. * Tính mở rộng: Decorator giúp bạn dễ dàng mở rộng các tính năng của đối tượng bằng cách thêm các lớp Decorator mới. Điều này làm cho việc phát triển và bảo trì ứng dụng trở nên dễ dàng hơn, vì bạn không cần phải thay đổi các đối tượng gốc. * Tính đa dạng: Decorator cho phép bạn kết hợp các tính năng của các đối tượng khác nhau bằng cách sử dụng nhiều lớp Decorator khác nhau. Điều này cho phép bạn tạo ra các đối tượng phức tạp hơn mà không cần phải tạo ra các lớp con mới cho từng tính năng. * Tính tương thích: Decorator làm cho các đối tượng tương thích với các phương thức gọi mà không cần phải hiểu cấu trúc của đối tượng. Điều này cho phép các đối tượng được sử dụng dễ dàng trong các hệ thống khác nhau mà không cần phải thay đổi code. * Tính tái sử dụng: Decorator cho phép bạn tái sử dụng các đối tượng và tính năng của chúng trong các ứng dụng khác nhau. Bằng cách sử dụng các lớp Decorator khác nhau, bạn có thể tạo ra các đối tượng có các tính năng khác nhau mà không cần phải viết lại code. |
| **Hạn chế:**   * Số lượng lớp: Sử dụng mẫu Decorator có thể dẫn đến tăng số lượng lớp trong hệ thống của bạn. Vì mỗi tính năng mới cần phải có một lớp Decorator mới, điều này có thể làm tăng độ phức tạp của hệ thống. * Hiệu suất: Sử dụng mẫu Decorator có thể ảnh hưởng đến hiệu suất của ứng dụng của bạn. Vì mỗi lớp Decorator phải truy cập đến đối tượng gốc để thực hiện các tính năng bổ sung, điều này có thể dẫn đến tốn thêm thời gian xử lý. * Độ khó trong việc hiểu: Sử dụng mẫu Decorator có thể làm cho code của bạn trở nên khó hiểu hơn. Vì mỗi đối tượng được bao bọc lại bằng nhiều lớp Decorator khác nhau, việc theo dõi các tính năng và hành vi của đối tượng có thể trở nên phức tạp. * Thứ tự gọi phương thức: Trong mẫu Decorator, việc sắp xếp thứ tự gọi các phương thức của các lớp Decorator là rất quan trọng. Nếu thứ tự này bị sai, có thể dẫn đến lỗi trong hệ thống của bạn. * Phân biệt giữa các đối tượng: Một số lớp Decorator có thể gây ra sự nhầm lẫn về các đối tượng trong hệ thống của bạn. Nếu không được xử lý cẩn thận, điều này có thể dẫn đến các lỗi trong ứng dụng của bạn. |

|  |
| --- |
| **Mẫu thiết kế:**  **Bridge** |
| **Mục đích sử dụng:**   * Tách biệt các lớp và các thành phần của chúng ra khỏi nhau để chúng có thể thay đổi mà không ảnh hưởng đến nhau * Cung cấp một cách để phân tách khỏi các lớp trừu tượng và các lớp cài đặt của chúng, đồng thời cho phép chúng tương tác với nhau thông qua giao diện. Điều này giúp giảm sự phụ thuộc giữa các lớp và làm cho chúng dễ dàng mở rộng và thay đổi trong tương lai |
| **Những trường hợp áp dụng mẫu:**   * Tách biệt giao diện người dùng (UI) và nghiệp vụ (business logic) để dễ dàng bảo trì và thay đổi code. * Cho phép thay đổi cài đặt của một thành phần mà không làm thay đổi các thành phần khác trong hệ thống. * Điều chỉnh cách các lớp tương tác với nhau, giúp tăng tính linh hoạt và tái sử dụng của hệ thống. * Cung cấp một cách để tách các tính năng chung ra khỏi các tính năng đặc thù, giúp đơn giản hóa mã nguồn và tăng khả năng kiểm thử và bảo trì. |
| **Cấu trúc mẫu chuẩn:**  Abstraction: Lớp Abstraction là lớp chứa đối tượng đang được xử lý, trong trường hợp này là lớp C. Lớp này sử dụng một đối tượng của lớp Implementation để thực hiện các hoạt động của nó.  public abstract class C {  protected P p;  public C(P p) {  this.p = p;  }  public abstract P FM();  }  Implementation: Lớp Implementation là lớp thực hiện các hoạt động cụ thể cho đối tượng được xử lý. Trong trường hợp này, lớp P là lớp Implementation.  public interface P {  public void operation();  }  public class P1 implements P {  public void operation() {  // Thực hiện một số hoạt động cụ thể cho P1  }  }  public class P2 implements P {  public void operation() {  // Thực hiện một số hoạt động cụ thể cho P2  }  }  Sau đó, sử dụng đối tượng của lớp C để thực hiện một số hoạt động, như sau:  C c = new C1(new P1());  P p = c.FM(); |
| **Các biến thể (nếu có):**  1. Trong biến thể này, cây Creator được suy biến thành một lớp duy nhất. Factory Method vẫn được sử dụng, tuy nhiên, nó có thêm một đối số để quy định loại đối tượng cụ thể sẽ được tạo ra. Bên trong FM() thường có câu lệnh switch...case để quyết định xem đối tượng cụ thể nào sẽ được tạo.  public class C {  protected P p;  public P FM(int type) {  switch(type) {  case 1:  p = new P1();  break;  case 2:  p = new P2();  break;  default:  p = new P1();  }  return p;  }  }  C c = new C();  P p = c.FM(1);  2. Giống biến thể 1 nhưng FM() được chuyển thành phương thức tĩnh và đặt ngay vào bên trong lớp Product(). Cây Creator không còn được sử dụng.  public abstract class P {  public static P FM(int type) {  switch(type) {  case 1:  return new P1();  case 2:  return new P2();  default:  return new P1();  }  }  }  P p = P.FM(1);  Cả hai biến thể trên đều giúp chúng ta tách biệt hoàn toàn đối tượng được xử lý và cách thức xử lý của nó. Mẫu Bridge cho phép chúng ta linh hoạt thay đổi cả đối tượng và cách thức xử lý của nó mà không làm ảnh hưởng đến nhau. |
| **Những dấu hiệu nhận biết mẫu:**   * Khi có sự phân cấp các lớp đối tượng và các lớp xử lý đối tượng khác nhau, và chúng có thể thay đổi độc lập với nhau. * Khi cần tách biệt hoàn toàn đối tượng được xử lý và cách thức xử lý của nó. * Khi có nhiều đối tượng khác nhau cần được xử lý theo nhiều cách khác nhau. * Khi sử dụng kỹ thuật Dependency Injection để giảm sự phụ thuộc giữa các lớp. * Khi một đối tượng có thể thực hiện nhiều chức năng khác nhau thông qua việc gắn kết với nhiều đối tượng khác nhau. |
| **Những lưu ý khác (nếu có):**   * Bridge cho phép chúng ta tách rời những thành phần khác nhau của hệ thống, giúp cho việc phát triển, bảo trì và mở rộng hệ thống dễ dàng hơn. * Việc sử dụng Bridge có thể làm tăng độ phức tạp của hệ thống nếu không được thiết kế đúng cách. Do đó, cần phải cân nhắc và thiết kế cẩn thận trước khi áp dụng mẫu này. * Bridge cũng có thể được sử dụng để tạo ra các hệ thống linh hoạt, cho phép thay đổi cách thức xử lý một đối tượng một cách dễ dàng mà không ảnh hưởng đến các thành phần khác của hệ thống. * Một số khía cạnh khác của Bridge là đảm bảo tính mở rộng, tính linh hoạt và tính tái sử dụng trong hệ thống. * Mẫu Bridge có thể được sử dụng để xây dựng các hệ thống lớn, phức tạp, có tính mở rộng cao, với nhiều thành phần đa dạng và các loại xử lý khác nhau. |
| **Ưu điểm:**   * Tách biệt hoàn toàn đối tượng được xử lý và cách thức xử lý của nó, giúp cho việc phát triển, bảo trì và mở rộng hệ thống dễ dàng hơn. * Giảm sự phụ thuộc giữa các lớp, giúp cho các thành phần của hệ thống có thể thay đổi độc lập với nhau. * Cho phép thay đổi cách thức xử lý một đối tượng một cách dễ dàng mà không ảnh hưởng đến các thành phần khác của hệ thống. * Có tính mở rộng, tính linh hoạt và tính tái sử dụng cao trong hệ thống. * Giúp cho việc phát triển hệ thống dễ dàng hơn bằng cách cho phép thay đổi cách thức xử lý của đối tượng mà không phải thay đổi cấu trúc của hệ thống. * Có khả năng xây dựng các hệ thống lớn, phức tạp và có tính mở rộng cao. * Tính linh hoạt của Mẫu Bridge cho phép các lớp con thực hiện các chức năng của chính nó, không bị ảnh hưởng bởi các lớp khác. |
| **Hạn chế:**   * Thiết kế và triển khai Mẫu Bridge có thể khá phức tạp, đặc biệt là đối với các hệ thống lớn và phức tạp. * Sử dụng Mẫu Bridge có thể làm tăng độ phức tạp của hệ thống và làm giảm hiệu suất của nó. * Các lớp con phải được thiết kế để tương thích với các lớp trừu tượng của Bridge, điều này có thể đòi hỏi một sự cân nhắc kỹ lưỡng về việc triển khai và bảo trì hệ thống. * Mẫu Bridge có thể không phù hợp với những ứng dụng đơn giản, không có nhiều thành phần phụ thuộc lẫn nhau. * Mẫu Bridge cũng có thể tạo ra sự rắc rối cho các lập trình viên mới bắt đầu học và sử dụng nó. |

|  |  |
| --- | --- |
| **Mẫu thiết kế:**  **Facade** | |
| **Mục đích sử dụng:**   * Cung cấp một lớp trung gian đơn giản để tương tác với hệ thống * Nó che giấu các chi tiết phức tạp của hệ thống, chẳng hạn như các lớp, các giao tiếp giữa các lớp, các đối tượng phụ thuộc và các quá trình xử lý phức tạp khác * Cung cấp một giao diện đơn giản để người dùng có thể tương tác với hệ thống một cách dễ dàng. | |
| **Những trường hợp áp dụng mẫu:**   * Hệ thống có nhiều phần phức tạp, với các lớp và đối tượng phụ thuộc lẫn nhau, và việc tương tác với hệ thống đòi hỏi nhiều sự tập trung và kiến thức chi tiết về hệ thống. * Người dùng cần một giao diện đơn giản để tương tác với hệ thống, mà không cần biết đến các chi tiết phức tạp của hệ thống. * Hệ thống cần được cải thiện hiệu suất bằng cách loại bỏ các phụ thuộc phức tạp trong hệ thống. * Hệ thống cần có khả năng mở rộng để phát triển thêm các tính năng mới trong tương lai mà không ảnh hưởng đến các phần khác của hệ thống. | |
| **Cấu trúc mẫu chuẩn:**   * Facade: Đây là thành phần trung tâm của mẫu thiết kế Facade. Nó cung cấp một interface đơn giản cho các client của hệ thống. Facade cũng đóng vai trò làm trung gian giữa client và các thành phần phức tạp trong hệ thống. * Subsystem: Đây là các thành phần phức tạp trong hệ thống. Chúng cung cấp các chức năng cần thiết để thực hiện các nhiệm vụ trong hệ thống. * Client: Đây là các thành phần sử dụng Facade để giao tiếp với hệ thống. Client không cần biết chi tiết về các thành phần phức tạp trong hệ thống. * Một số phương thức: Đây là các phương thức được định nghĩa trong Facade để cung cấp các chức năng cần thiết cho client. Facade sẽ sử dụng các phương thức này để truy xuất và sử dụng các thành phần phức tạp trong hệ thống. | |
| **Các biến thể (nếu có):**   * Protected Facade: Facade được bảo vệ bằng cách đặt các thành phần phức tạp của hệ thống trong một lớp bảo vệ, giới hạn truy cập đến các thành phần này chỉ cho Facade và các lớp kế thừa của nó. * Cached Facade: Facade được cải tiến bằng cách thêm chức năng cache, giúp tăng hiệu suất và giảm thời gian truy cập vào các thành phần phức tạp trong hệ thống. * Remote Facade: Facade được sử dụng để truy cập các thành phần của một hệ thống từ xa, thường thông qua các giao thức mạng như SOAP hoặc REST. * Multiple Facades: Hệ thống có nhiều Facade, mỗi Facade cung cấp một tập hợp các chức năng riêng biệt. Điều này giúp tăng tính linh hoạt và tái sử dụng của hệ thống. * Dynamic Facade: Facade được xây dựng theo cách cho phép thêm hoặc xóa các thành phần phức tạp trong hệ thống mà không cần sửa đổi code của Facade. Facade sẽ tự động phát hiện các thay đổi này và cung cấp các chức năng mới cho client. | |
| **Những dấu hiệu nhận biết mẫu:**   * Sử dụng một lớp Facade * Truy cập vào các thành phần phức tạp thông qua Facade * Facade cung cấp các phương thức đơn giản * Các thành phần phức tạp không phải là public * Giảm thiểu sự phức tạp của hệ thống | |
| **Ưu điểm:**   * Giảm thiểu sự phức tạp của hệ thống: Facade cung cấp một interface đơn giản cho client truy xuất các thành phần phức tạp trong hệ thống, giảm thiểu sự phức tạp của hệ thống và làm cho việc phát triển và bảo trì hệ thống trở nên dễ dàng hơn. * Tăng tính tái sử dụng và bảo trì: Facade làm cho việc tái sử dụng code trở nên dễ dàng hơn vì nó ẩn đi các chi tiết phức tạp của hệ thống. Nếu cần thay đổi code, bạn chỉ cần sửa đổi trong lớp Facade mà không cần sửa đổi nhiều phần khác trong hệ thống. * Giảm thiểu sự phụ thuộc của client vào các thành phần phức tạp của hệ thống: Client không cần biết về các chi tiết phức tạp của hệ thống, mà chỉ cần sử dụng các phương thức đơn giản được cung cấp bởi Facade. * Cải thiện tính linh hoạt của hệ thống: Việc sử dụng Facade giúp hệ thống trở nên linh hoạt hơn trong việc thay đổi, cập nhật, hoặc mở rộng, bởi vì client không bị phụ thuộc vào các thành phần phức tạp của hệ thống. * Tăng hiệu suất của hệ thống: Nếu được thực hiện đúng cách, Facade cũng có thể giúp tăng hiệu suất của hệ thống bằng cách sử dụng các kỹ thuật như caching hoặc giảm tải. | |
| **Hạn chế:**   * Giới hạn tính linh hoạt của hệ thống: Facade chỉ cung cấp một interface đơn giản cho client truy xuất các thành phần phức tạp trong hệ thống. Nếu client cần truy xuất các thành phần phức tạp hơn hoặc thực hiện các chức năng phức tạp hơn, họ sẽ phải truy cập trực tiếp vào các thành phần đó thay vì sử dụng Facade. * Tạo ra một lớp phụ trách quá nhiều: Nếu lớp Facade trở nên quá phức tạp hoặc có quá nhiều trách nhiệm, nó có thể trở nên khó hiểu và khó bảo trì. * Không phù hợp cho các ứng dụng lớn và phức tạp: Nếu hệ thống quá lớn hoặc phức tạp, sử dụng mẫu thiết kế Facade có thể không đáp ứng được các yêu cầu về hiệu suất hoặc tính linh hoạt của hệ thống. * Khó khăn trong việc bảo trì và nâng cấp hệ thống: Nếu cần thay đổi các thành phần phức tạp trong hệ thống, có thể cần phải sửa đổi lớp Facade và các client của nó. Điều này có thể làm cho việc bảo trì và nâng cấp hệ thống trở nên khó khăn hơn. | |
| **Mẫu thiết kế**  **State Strategy** |
| **Mục đích sử dụng:**  Mẫu thiết kế State Strategy được sử dụng để quản lý trạng thái của một đối tượng hoặc hệ thống thông qua việc định nghĩa các trạng thái khác nhau và quyết định chuyển đổi giữa các trạng thái đó dựa trên các sự kiện và điều kiện xảy ra. |
| **Những trường hợp áp dụng mẫu:**  Mẫu thiết kế State Strategy thường được áp dụng trong các hệ thống có nhiều trạng thái khác nhau, ví dụ như các ứng dụng quản lý đơn hàng, các trò chơi điện tử hoặc các hệ thống tự động hoá. |
| **Cấu trúc mẫu chuẩn:**  - State: Định nghĩa một giao diện cho các trạng thái trong hệ thống.  -ConcreteState: Các lớp cụ thể triển khai giao diện State.  -Context: Định nghĩa các phương thức để thay đổi trạng thái và quản lý các ConcreteState.  -Client: Sử dụng Context để tương tác với các ConcreteState. |
| **Các biến thể:**  -Finite State Machine: Sử dụng một số hữu hạn các trạng thái và các chuyển đổi giữa chúng.  -Hierarchical State Machine: Một phiên bản nâng cao của Finite State Machine, trong đó các -trạng thái được phân cấp và có thể có các trạng thái con.  -Orthogonal State Machine: Một phiên bản nâng cao hơn của Hierarchical State Machine, -trong đó các trạng thái con của một trạng thái cha có thể tồn tại đồng thời. |
| **Những dấu hiệu nhận biết mẫu:**  -Các lớp State và ConcreteState thường có các phương thức tương tự nhau, thường được đặt tên theo các trạng thái của hệ thống.  -Context thường chứa các phương thức để thay đổi trạng thái và các chuyển đổi giữa chúng.  -Mẫu thiết kế State Strategy thường được sử dụng để quản lý các trạng thái của một đối tượng hoặc hệ thống thông qua các chuyển đổi giữa các trạng thái dựa trên các sự kiện và điều kiện xảy ra. |
| **Ưu điểm:**  -Mẫu thiết kế State Strategy giúp tách riêng logic quản lý trạng thái ra khỏi logic xử lý sự kiện, giúp cho hệ thống dễ bảo trì và mở rộng hơn.  -Việc định nghĩa các trạng thái và chuyển đổi giữa chúng rõ ràng và dễ hiểu, giúp cho phát triển và bảo trì hệ thống dễ dàng hơn.  -Mẫu thiết kế State Strategy có thể giúp giảm số lượng mã lặp lại, giúp cho mã nguồn trở nên ngắn gọn và dễ đọc hơn. |
| **Hạn chế:**  -Mẫu thiết kế State Strategy có thể gây ra sự phức tạp về logic và thực hiện, đặc biệt là trong các hệ thống có nhiều trạng thái và chuyển đổi giữa chúng phức tạp.  -Nếu không được thiết kế và triển khai đúng cách, mẫu thiết kế State Strategy có thể dẫn đến các vấn đề về hiệu năng và độ tin cậy.  -Các biến thể của mẫu thiết kế State Strategy có thể làm cho hệ thống trở nên phức tạp hơn, đặc biệt là khi có nhiều trạng thái con và trạng thái đồng thời. |
| **Mẫu thiết kế**  **Iterator** |
| **Mục đích sử dụng:**  Mẫu thiết kế Iterator được sử dụng để duyệt qua các phần tử của một tập hợp mà không cần biết cấu trúc bên trong của nó.  Nó giúp tách riêng logic duyệt tập hợp ra khỏi logic xử lý của chương trình, làm cho mã nguồn trở nên đơn giản hơn. |
| **Những trường hợp áp dụng mẫu:**  Mẫu thiết kế Iterator thường được sử dụng trong các ngôn ngữ lập trình hướng đối tượng để duyệt qua các cấu trúc dữ liệu như danh sách liên kết, mảng, cây, tập hợp, ...  Nó được sử dụng trong các chương trình xử lý tập tin để đọc dữ liệu từ một tập tin lớn mà không cần phải đọc toàn bộ dữ liệu vào bộ nhớ trước. |
| **Cấu trúc mẫu chuẩn:**  - Iterator: lớp mẫu đại diện cho một trình duyệt cho phép duyệt qua các phần tử của một tập hợp.  -ConcreteIterator: lớp cụ thể của Iterator, cài đặt các phương thức của Iterator để duyệt qua các phần tử của tập hợp.  -Aggregate: lớp mẫu đại diện cho tập hợp các phần tử cần được duyệt.  ConcreteAggregate: lớp cụ thể của Aggregate, cài đặt các phương thức của Aggregate để cung cấp Iterator để duyệt qua các phần tử của tập hợp. |
| **Các biến thể:**  -External Iterator: Iterator được điều khiển bởi người dùng.  -Internal Iterator: Iterator được điều khiển bởi chính tập hợp, thường được cài đặt dưới dạng hàm hoặc lambda. |
| **Những dấu hiệu nhận biết mẫu:**  -Khi một đối tượng cung cấp phương thức để duyệt qua các phần tử của nó mà không tiết lộ cấu trúc bên trong của đối tượng đó.  -Khi phương thức duyệt qua các phần tử của đối tượng được cài đặt thông qua một lớp riêng biệt. |
| **Ưu điểm:**  -Mẫu thiết kế Iterator giúp tách riêng logic duyệt tập hợp ra khỏi logic xử lý của chương trình, làm cho mã nguồn trở nên đơn giản hơn.  -Nó cung cấp khả năng duyệt qua các phần tử của một tập hợp mà không cần biết cấu trúc bên trong của nó.  -Nó cho phép lặp lại tập hợp các phần tử một cách linh hoạt, mà không làm ảnh hưởng đến cấu trúc bên trong của tập hợp. |
| **Hạn chế:**  -Iterator chỉ duyệt qua các phần tử theo chiều đơn giản, không thể truy xuất đến các phần tử ngẫu nhiên trong tập hợp.  -Sử dụng Iterator có thể làm tăng thời gian xử lý trong một số trường hợp, do phải duyệt qua toàn bộ tập hợp để tìm kiếm phần tử cần thiết.  -Trong một số ngôn ngữ lập trình, việc sử dụng Iterator có thể gây ra tình trạng mất hiệu năng do số lần gọi hàm tăng lên. |
| **Mẫu thiết kế**  **Observer** |
| **Mục đích sử dụng:**  Thiết kế Observer được sử dụng để xây dựng các hệ thống có tính mềm dẻo và dễ mở rộng, cho phép các đối tượng phản ứng với sự thay đổi của đối tượng chủ động một cách linh hoạt. |
| **Những trường hợp áp dụng mẫu:**  -Khi muốn cho phép các đối tượng phản ứng với sự thay đổi của một đối tượng khác.  -Khi muốn giảm sự phụ thuộc giữa các đối tượng trong hệ thống. |
| **Cấu trúc mẫu chuẩn:**  -Subject: Là đối tượng chủ động, bao gồm danh sách các Observer và phương thức thêm, xóa, thông báo các Observer.  -Observer: Là đối tượng phản ứng với sự thay đổi của Subject, bao gồm phương thức update().  -ConcreteSubject: Là một lớp kế thừa từ Subject, quản lý trạng thái của đối tượng chủ động.  -ConcreteObserver: Là một lớp kế thừa từ Observer, thực hiện các hành động phản ứng với sự thay đổi của Subject. |
| **Các biến thể:**  -Push vs Pull: Trong Push, Subject thông báo cho Observer cập nhật dữ liệu khi có sự thay đổi; trong Pull, Observer yêu cầu dữ liệu từ Subject khi cần.  -Broadcast vs Multicast: Trong Broadcast, Subject thông báo cho tất cả các Observer khi có sự thay đổi; trong Multicast, Subject chỉ thông báo cho các Observer có liên quan đến sự thay đổi. |
| **Những dấu hiệu nhận biết mẫu:**  -Có một đối tượng chủ động, đăng ký danh sách các Observer để phản ứng với sự thay đổi của đối tượng này.  -Các Observer cập nhật thông tin của mình khi có sự thay đổi của đối tượng chủ động. |
| **Ưu điểm:**  -Đảm bảo tính linh hoạt và giảm thiểu sự phụ thuộc giữa các thành phần trong hệ thống.  -Cho phép chia sẻ dữ liệu giữa nhiều đối tượng mà không cần phải biết rõ thông tin chi tiết về các đối tượng đó.  -Đảm bảo tính rõ ràng và minh bạch trong việc xử lý sự kiện, giúp dễ dàng kiểm tra và sửa chữa lỗi.  -Giúp hệ thống dễ dàng mở rộng và thêm mới các thành phần mới mà không ảnh hưởng đến các thành phần hiện có.  -Tăng tính tái sử dụng của mã nguồn, giảm thiểu độ phức tạp trong việc phát triển hệ thống.  -Cho phép tách biệt quá trình xử lý sự kiện và quá trình hiển thị giao diện người dùng, giúp dễ dàng thay đổi giao diện mà không ảnh hưởng đến xử lý sự kiện. |
| **Hạn chế:**  -Việc sử dụng quá nhiều đối tượng Observer có thể dẫn đến tình trạng quá tải và ảnh hưởng đến hiệu suất của hệ thống.  -Việc thiết kế không cẩn thận có thể dẫn đến tình trạng xảy ra vòng lặp vô hạn trong quá trình truyền thông tin giữa các đối tượng.  -Đối với các hệ thống có quá nhiều sự kiện và đối tượng, việc theo dõi và xử lý các sự kiện có thể trở nên phức tạp và khó khăn.  -Khó khăn trong việc hiểu và sử dụng mẫu thiết kế Observer cho các lập trình viên mới bắt đầu hoặc không có kinh nghiệm về mẫu thiết kế. |

`

|  |
| --- |
| **Mẫu thiết kế**  **Template Method** |
| **Mục đích sử dụng:**  Mẫu thiết kế Template Method được sử dụng để định nghĩa các bước thực hiện của một thuật toán, nhưng để các lớp con xác định các chi tiết cụ thể của các bước đó mà không làm thay đổi cấu trúc chung của thuật toán. |
| **Những trường hợp áp dụng mẫu:**  -Khi muốn có một thuật toán với các bước khác nhau, nhưng các bước này vẫn có thể tái sử dụng được cho các lớp khác nhau.  -Khi cần định nghĩa một chuỗi các bước thực hiện để hoàn thành một nhiệm vụ cụ thể nhưng các bước này có thể được thực hiện theo nhiều cách khác nhau.  -Khi cần thực hiện một thuật toán mà các bước thực hiện được xác định trong lớp cha và các lớp con có thể thay đổi chúng theo cách mà chúng muốn. |
| **Cấu trúc mẫu chuẩn:**  -AbstractClass: Lớp cha chứa phương thức Template Method và các phương thức trừu tượng để các lớp con phải triển khai.  -ConcreteClass: Các lớp con triển khai phương thức trừu tượng của lớp cha. |
| **Các biến thể:**  -Non-Abstract Template Method: Trong trường hợp này, lớp cha có thể cung cấp một phiên bản mặc định của Template Method, nhưng vẫn có thể cho phép các lớp con ghi đè lên phương thức đó nếu cần thiết.  -Hook Method: Hook method là một phương thức mà lớp cha cung cấp nhưng không cần phải triển khai, và các lớp con có thể chọn triển khai hoặc không triển khai. |
| **Những dấu hiệu nhận biết mẫu:**  -Lớp cha chứa một phương thức Template Method và một số phương thức trừu tượng để các lớp con triển khai.  -Lớp con triển khai các phương thức trừu tượng của lớp cha.  -Phương thức Template Method được định nghĩa bằng cách gọi các phương thức trừu tượng để thực hiện các bước trong thuật toán. |
| **Ưu điểm:**  -Tăng tính linh hoạt: Template Method cho phép các lớp con thay đổi phần chức năng của phương thức template trong khi giữ nguyên cấu trúc tổng thể.  -Dễ dàng bảo trì: Tất cả các hành động chung của lớp cha được định nghĩa trong phương thức template, điều này giúp cho việc bảo trì và cập nhật phần mềm trở nên dễ dàng hơn.  -Tái sử dụng mã: Mẫu thiết kế này giúp tăng khả năng tái sử dụng mã, vì các chức năng chung được định nghĩa trong phương thức template có thể được sử dụng lại trong các lớp con.  -Tăng tính tuân thủ chuẩn: Khi sử dụng Template Method, các lớp con phải tuân thủ cấu trúc và yêu cầu của lớp cha, giúp cho việc phát triển phần mềm được thực hiện một cách tuân thủ chuẩn hơn.  -Dễ dàng thay đổi thuật toán: Việc định nghĩa các bước của thuật toán trong phương thức template giúp cho việc thay đổi thuật toán trở nên dễ dàng hơn.  -Tính đóng gói: Template Method giúp giữ cho các chi tiết thực hiện của thuật toán bên trong lớp, do đó giữ cho mã được giữ gìn tính đóng gói. |
| **Hạn chế:**  -Giới hạn linh hoạt: Template Method có thể giới hạn tính linh hoạt của lớp con trong việc định nghĩa phần cụ thể của thuật toán.  -Không phù hợp với những thuật toán động: Template Method chỉ phù hợp với những thuật toán có bước xác định trước cụ thể, không phù hợp với những thuật toán động hoặc phức tạp.  -Khó khăn trong việc kiểm soát thứ tự thực hiện: Nếu lớp con không được cài đặt đúng cách, có thể dẫn đến sự khó khăn trong việc kiểm soát thứ tự thực hiện của các phương thức của lớp cha. |