|  |
| --- |
| **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:**  Client:  C c = new C1();  P p = c.FM();  //Sử dụng p… |
| **Các biến thể (nếu có):** |
| **Những dấu hiệu nhận biết mẫu:**   * A * B * C |
| **Những lưu ý khác (nếu có):**   * A * B * C |
| **Ưu điểm:**   * A |
| **Hạn chế:**   * A |

|  |
| --- |
| **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. |

* CREATIONAL PATTERNS:
  + Factory Method
  + Abstract Factory
  + Builder
  + Singleton
* STRUCTURAL PATTERNS:
  + Adapter
  + Composite
  + Decorator
  + Bridge
  + Façade
* BEHAVIORAL PATTERNS
  + Iterator
  + Observer (Publisher –Subscriber / Broadcast – Receiver)
  + Strategy
  + State
  + Template Method
  + Visitor