|  |  |
| --- | --- |
| **Design Pattern - Thiết kế theo mô hình mẫu** |  |
|  | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | **(Post 23/01/2007) Trong phát triển phần mềm hiện đại, kiến trúc tổng thể của dự án đóng một vai trò quan trọng, đặc biệt với bộ khung (framework) và mẫu thiết kế (design pattern). Bài viết này sẽ giúp các bạn hiểu được một cách tổng quan về pattern cũng như cách thức thiết kế một số pattern tiêu biểu.**   |  | | --- | | *Mô hình Adapter Pattern* | | http://aptech.fpt.edu.vn/images/CongNghe23-01-07_pic1_large.gif |   **PATTERN là gì?**  Pattern mô tả một giải pháp chung đối với một vấn đề nào đó trong thiết kế thường được “lặp lại” trong nhiều dự án. Nói một cách khác, một pattern có thể được xem như một “khuôn mẫu” có sẵn áp dụng được cho nhiều tình huống khác nhau để giải quyết một vấn đề cụ thể. Trong bất kỳ hệ thống phần mềm hướng đối tượng nào chúng ta cũng có thể bắt gặp các vấn đề lặp lại.  **Đặc điểm chung:**   * **Pattern được hiểu theo nghĩa tái sử dụng ý tưởng hơn là mã lệnh.** Pattern cho phép các nhà thiết kế có thể cùng ngồi lại với nhau và cùng giải quyết một vấn đề nào đó mà không phải mất nhiều thời gian tranh cãi. Trong rất nhiều trường hợp, dự án phần mềm thất bại là do các nhà phát triển không có được sự hiểu biết chung trong các vấn đề về kiến trúc phần mềm. Ngoài ra, pattern cũng cung cấp những thuật ngữ và khái niệm chung trong thiết kế. Nói một cách đơn giản, khi đề cập đến một pattern nào đấy, bất kỳ ai biết pattern đó đều có thể nhanh chóng hình dung ra “bức tranh” của giải pháp. Và cuối cùng, nếu áp dụng pattern hiệu quả thì việc bảo trì phần mềm cũng được tiến hành thuận lợi hơn, nắm bắt kiến trúc hệ thống nhanh hơn. * **Pattern hỗ trợ tái sử dụng kiến trúc và mô hình thiết kế phần mềm theo quy mô lớn.**Cần phân biệt design pattern với framework. Framework hỗ trợ tái sử dụng mô hình thiết kế và mã nguồn ở mức chi tiết hơn. Trong khi đó, design pattern được vận dụng ở mức tổng quát hơn, giúp các nhà phát triển hình dung và ghi nhận các cấu trúc tĩnh và động cũng như quan hệ tương tác giữa các giải pháp trong quá trình thiết kế ứng dụng đối với một chuyên khu riêng biệt. * **Pattern đa tương thích.** Pattern không phụ thuộc vào ngôn ngữ lập trình, công nghệ hoặc các nền tảng lớn như J2EE của Sun hay Microsoft .NET Framework.   Tiềm năng ứng dụng của pattern là rất lớn. Các thiết kế dựa trên pattern được sử dụng khá nhiều ở các phần mềm mã nguồn mở, trong nền tảng J2EE hoặc .NET... Trong các dạng ứng dụng này, có thể dễ dàng nhận ra một số tên lớp chứa các tiền tố hoặc hậu tố như Factory, Proxy, Adapter...  **PHÂN LOẠI PATTERN**  Pattern được phân loại ra làm 3 nhóm chính sau đây:   * **Nhóm cấu thành (Creational Pattern):** Gồm Factory, Abstract Factory, Singleton, Prototype, Builder... Liên quan đến quá trình khởi tạo đối tượng cụ thể từ một định nghĩa trừu tượng (abstract class, interface). * **Nhóm cấu trúc tĩnh (Structural Pattern):** Gồm Proxy, Adapter, Wrapper, Bridge, Facade, Flyweight, Visitor... Liên quan đến vấn đề làm thế nào để các lớp và đối tượng kết hợp với nhau tạo thành các cấu trúc lớn hơn. * **Nhóm tương tác động (Behavioral Pattern):** Gồm Observer, State, Command, Iterator... Mô tả cách thức để các lớp hoặc đối tượng có thể giao tiếp với nhau.   Dưới đây chúng ta sẽ tìm hiểu chi tiết một số pattern tiêu biểu nhất: Factory, Abstract Factory, Singleton, Proxy, Adapter và Wrapper. Chúng ta quy ước với nhau rằng “giao diện lớp” được hiểu như interface hoặc abstract class vì đây đơn thuần là các định nghĩa lớp.  **1. FACTORY PATTERN**  **Định nghĩa**  Factory Pattern định nghĩa một lớp (interface, abstract, class) đóng vai trò như một “nhà xưởng” có nhiệm vụ khởi tạo đối tượng “cụ thể” khi ứng dụng chạy. Tại thời điểm thiết kế đối tượng này được định nghĩa trừu tượng.  **Đặc điểm**  Kiểm soát được các hoạt động trong suốt chu kỳ sống của đối tượng, như khởi tạo đối tượng, huỷ đối tượng... Đảm bảo cho các đối tượng được thực thi an toàn. Nắm được thông tin về những đối tượng nào được tạo ra và được khởi tạo ra sao. Nói cách khác, các đối tượng được quản lý tốt hơn và an toàn hơn với Factory Pattern.  Đối tượng Factory thường được đặt tên theo những chuẩn khác nhau nhưng vẫn có thể dễ dàng nhận ra thiết kế Factory Pattern ẩn chứa trong đó. Ví dụ: BankFactory,...  Tính chất đóng gói (encapsulation) thể hiện rõ trong Factory Pattern; các thông tin liên quan đến truy cập đối tượng được che giấu trong Factory. Thiết kế Factory luôn có một thủ tục khởi tạo đối tượng, ví dụ createObject().  Factory Pattern tuân thủ nguyên tắc thiết kế DIP (Dependency Inversion Principle): không nên phụ thuộc vào những thứ quá cụ thể.  **Phân loại**  Factory Pattern được thiết kế theo một trong hai cách sau đây:   * **Based-class Factory Pattern:** Mẫu này sử dụng tính chất thừa kế để phân loại các đối tượng được tạo ra. * **Based-object Factory Pattern:** Sử dụng mối quan hệ kết hợp để tham chiếu tới một đối tượng sẽ được tạo ra. Đối tượng được tạo ra sẽ trở thành một phần hay thuộc tính của lớp Factory. Chúng ta thường hay gặp loại này trong Abstract Factory Pattern được trình bày ở phần tiếp theo.  |  | | --- | | *Mô hình Abstract Factory Pattern* | | http://aptech.fpt.edu.vn/images/CongNghe23-01-07_pic2.gif |   **2. ABSTRACT FACTORY PATTERN**  **Định nghĩa**  Abstract Factory cung cấp một giao diện lớp có chức năng tạo ra một tập hợp các đối tượng liên quan hoặc phụ thuộc lẫn nhau mà không chỉ ra đó là những lớp cụ thể nào tại thời điểm thiết kế.  Về bản chất, Abstract Factory Pattern chỉ khác Factory Pattern ở chỗ bản thân đối tượng Factory không được chỉ ra cụ thể tại thời điểm thiết kế, tức nó là một giao diện hoặc lớp trừu tượng (interface, abstract). Nếu như Factory Patttern phân loại đối tượng dựa trên tham số đầu vào thì đối với Abstract Factory Pattern, thủ tục createObject() còn phụ thuộc thêm vào các yếu tố phụ khác như môi trường hệ điều hành chẳng hạn. Ứng với mỗi yếu tố phụ thứ hai ta có một lớp Factory cụ thể.  **Thiết kế động với Abstract Factory**  Một trong những vấn đề gặp phải là khung giao diện Abstract Factory thường hay bị sửa đổi, thí dụ như bổ sung thủ tục chẳng hạn, khi đó các lớp cụ thể thực thi giao diện này sẽ phải được dịch và triển khai lại. Để giảm nhẹ vấn đề này người ta thường thiết kế giao diện Abstract Factory một cách linh động.  **3. SINGLETON PATTERN (Static Factory Pattern)**  **Định nghĩa**  Singleton Pattern đảm bảo một lớp chỉ có một thực thể (instance) duy nhất được tạo ra và đồng thời cung cấp một truy cập toàn cục đến đối tượng được tạo ra.  Chúng ta xét trường hợp có nhiều đối tượng có cùng chung một số tính chất nào đó được tạo ra ứng với mỗi một yêu cầu từ các đối tượng khách (client), lúc này độ phức tạp sẽ tăng lên và ứng dụng sẽ chiếm dụng nhiều vùng nhớ hơn. Singleton Pattern là một giải pháp đặc biệt của Factory Pattern ở chỗ đối tượng sinh ra là điểm truy cập toàn cục “duy nhất” đối với mọi chương trình gọi đến, hay nói một cách khác tất cả các đối tượng khách gọi đến đều chia sẻ đối tượng được tạo ra.  Ứng dụng rõ rệt nhất của Singleton Pattern có thể thấy trong dịch vụ web khi triệu gọi các đối tượng từ xa, ở đó đối tượng nằm trên server hoặc sẽ phục vụ chung cho tất cả các ứng dụng khách (singleton) hoặc sẽ chỉ đáp ứng một ứng dụng khách riêng lẻ nào đó rồi tự bị phá huỷ sau đó (single call).  Về các mẫu thiết kế tiêu biểu trong nhóm cấu thành: Factory, Abstract Factory và Singleton, các bạn có thể tham khảo thêm tài liệu về phương pháp xây dựng cụ thể cũng như mã nguồn chương trình viết bằng C#.NET [tại đây](http://www.codeproject.com/gen/design/CSharpClassFactory.asp)  **4. PROXY PATTERN**  **Định nghĩa**  Proxy Pattern là mẫu thiết kế mà ở đó tất cả các truy cập trực tiếp một đối tượng nào đó sẽ được chuyển hướng vào một đối tượng trung gian (Proxy Class).  Nếu như Factory Pattern giúp quản lý đối tượng tốt hơn thì Proxy Pattern lại có nhiệm vụ bảo vệ việc truy cập một đối tượng thông qua Proxy, hay còn gọi là truy cập gián tiếp. Proxy được ủy quyền về phía ứng dụng khách cho phép tương tác với đối tượng đích theo những cách khác nhau; như gửi yêu cầu một dịch vụ nào đó, theo dõi trạng thái và vòng đời đối tượng, xây dựng lớp vỏ bảo vệ đối tượng... Thí dụ chúng ta phát hiện ra một đối tượng trong một thư viện DLL có thể bị khai thác truy cập vào một số trường quan trọng, khi đó chúng ta không thể mở mã nguồn thư viện đã được dịch để vá lỗ hổng, giải pháp lúc này là xây dựng một proxy ngăn chặn truy cập các trường đó và cuối cùng biên dịch lại thành một DLL mới.  **Phân loại**  Độ phức tạp của giải pháp sử dụng Proxy Pattern phụ thuộc vào tình huống bài toán đưa ra, chúng ta sẽ lần lượt tìm hiểu nguyên tắc làm việc của các proxy dưới đây:   * **Remote Proxy:** Client truy cập qua remote proxy để tham chiếu tới một đối tượng được bảo vệ nằm bên ngoài ứng dụng (trên cùng máy hoặc máy khác) như dịch vụ Windows, dịch vụ web, ứng dụng ở xa... Mô hình này "che giấu" đối tượng được triệu gọi đang nằm ở rất xa đâu đó và client có vẻ như truy cập vào đối tượng nằm trên cùng một chuyên khu làm việc (domain). * **Virtual Proxy:** Virtual Proxy tạo ra một đối tượng trung gian mỗi khi có yêu cầu tại thời điểm thực thi ứng dụng, nhờ đó làm tăng hiệu suất của ứng dụng. * **Monitor Proxy:** Monitor Proxy sẽ thiết lập các ràng buộc bảo mật trên đối tượng cần bảo vệ, ngăn không cho client truy cập một số trường quan trọng của đối tượng. * **Protection Proxy:** Đối với proxy này thì phạm vi truy cập của các client khác nhau sẽ khác nhau. Protection Proxy sẽ kiểm tra các quyền truy cập của client khi có một dịch vụ được yêu cầu. * **Cache Proxy:** Cung cấp không gian lưu trữ tạm thời cho các kết quả trả về từ đối tượng nào đó, kết quả này sẽ được tái sử dụng cho các client chia sẻ chung một yêu cầu gửi đến và do đó làm tăng đáng kể hiệu suất chương trình. * **Firewall Proxy:** Bảo vệ đối tượng từ chối các yêu cầu xuất xứ từ các client không tín nhiệm. * **Smart Reference Proxy:** Là nơi kiểm soát các hoạt động bổ sung mỗi khi đối tượng được tham chiếu, ví dụ như kiểm soát vòng đời của đối tượng, lưu lại số lần tham chiếu vào đối tượng... * **Synchronization Proxy:** Đảm bảo nhiều client có thể truy cập vào cùng một đối tượng mà không gây ra xung đột. Thực tế có rất nhiều tình huống khiến chúng ta phải nghĩ đến thiết kế này. Một synchronization proxy được thiết lập có thể kiểm soát được nhiều yêu cầu cập nhật dữ liệu một cách đồng thời, tại thời điểm bắt đầu cập nhật chỉ có một client với mức ưu tiên cao nhất giành được khoá để đánh dấu rằng các client khác cần phải chờ đến lượt.   Synchronization proxy hoạt động rất hiệu quả và phổ biến trong thiết kế các bài toán đa tuyến. Một hiện tượng hay xảy ra với thiết kế này là khi một client nào đó chiếm dụng khoá khá lâu (và thậm chí là mãi mãi) khiến cho số lượng các client trong danh sách hàng đợi cứ tăng lên, và do đó hoạt động của hệ thống bị ngừng trệ, có thể dẫn đến hiện tượng “tắt nghẽn” là hiện tượng khoá được giữ vô thời hạn bởi một đối tượng nào đó. Trong trường hợp này người ta cải tiến thành mẫu thiết kế phức tạp hơn, đó là Copy-On-Write Proxy.   * **Copy-On-Write Proxy:** Cope-On-Write Proxy đảm bảo rằng sẽ không có client nào phải chờ vô thời hạn. Thiết kế này rất phức tạp do đó chỉ nên ứng dụng Copy-On-Write Proxy thay thế Synchronization Proxy khi hệ thống được dự đoán sẽ thường xuyên bị ngừng trệ hoặc có hiện tượng “tắt nghẽn” xảy ra.   **Đặc điểm chung**  Proxy Pattern có những đặc điểm chung sau đây:   * Cung cấp mức truy cập gián tiếp vào một đối tượng. * Tham chiếu vào đối tượng đích và chuyển tiếp các yêu cầu đến đối tượng đó. * Cả proxy và đối tượng đích đều kế thừa hoặc thực thi chung một lớp giao diện. Mã máy dịch cho lớp giao diện thường “nhẹ” hơn các lớp cụ thể và do đó có thể giảm được thời gian tải dữ liệu giữa server và client.  |  | | --- | | *Mô hình Proxy Pattern* | | http://aptech.fpt.edu.vn/images/CongNghe23-01-07_pic3.gif |   **5. ADAPTER PATTERN**  **Định nghĩa**  Adapter Pattern biến đổi giao diện của một lớp thành một giao diện khác mà các đối tượng client có thể hiểu được. Lớp với giao diện được tạo ra đó gọi là Adapter. Nguyên tắc cơ bản của Adapter Pattern nằm ở chỗ làm thế nào để các lớp với các giao diện không tương thích có thể làm việc được với nhau.  Nguyên lý xây dựng Adapter Pattern khá đơn giản: chúng ta xây dựng một lớp với một giao diện mong muốn sao cho lớp đó giao tiếp được với một lớp cho trước ứng với một giao diện khác.  Adapter Pattern không quản lý tập trung các đối tượng gần giống nhau như Factory Pattern, mà kết nối với nhiều lớp không có liên quan gì với nhau. Ví dụ lớp A sau khi thực thi giao diện của nó và vẫn muốn bổ sung các phương thức từ một lớp B nào đó, chúng ta có thể kết nối A với B thông qua hình thức kế thừa hoặc liên kết đối tượng như một thành phần. Adapter Pattern có sự giống nhau một chút với Proxy Pattern ở chỗ nó tận dụng tối đa tính chất “uỷ quyền” (delegation); lớp Adapter sẽ kết nối với một đối tượng nào đó gọi là Adaptee và Adapter sẽ được uỷ quyền truy cập vào Adaptee, lớp Adapter đóng vai trò như một kênh trung gian để client truy cập vào một số các thành phần quan trọng của lớp Adaptee.  **Đặc điểm**   * Adapter Pattern hướng tập trung vào giải quyết sự tương thích giữa hai giao diện đang tồn tại, giảm công sức viết lại mã lệnh xuống mức tối thiểu có thể được. * Tái sử dụng giao diện cũ và Adapter Pattern chỉ thực sự cần thiết khi mọi thứ đã được thiết kế từ trước.   **Phạm vi ứng dụng**  Adapter Pattern được ứng dụng trong các trường hợp:   * Cần tích hợp một vài module vào chương trình. * Không thể sát nhập trực tiếp module vào chương trình (ví dụ như module thư viện đã được dịch ra .DLL, .CLASS...). * Module đang tồn tại không có giao diện mong muốn như:   + Cần nhiều hơn phương thức cho module đó.   + Một số phương thức có thể được nạp chồng.   **6. WRAPPER PATTERN**  Wrapper Pattern là một trường hợp đặc biệt của Adapter Pattern. Nếu một Adapter chỉ đơn thuần là “nhúng” (wrap) các lớp với các giao diện không tương thích với nhau để chúng có thể hoạt động cùng nhau thì có thể được gọi bằng tên riêng Wrapper Pattern. Khi đó lớp Adapter còn được gọi là lớp Wrapper. Đây là quan hệ “có một”, tức là một giao diện không tương thích có thể được nhúng vào thành một phần của một giao diện khác.  **Đặc điểm**  Đối tượng Wrapper mô phỏng tất cả các hành vi (hàm, thủ tục) của giao diện được nhúng bởi các hành vi với tên y hệt. Thí dụ nếu lớp được nhúng A có thủ tục SpecificRequest() thì lớp Wrapper cũng phải có thủ tục SpecificRequest() tham chiếu đến thủ tục cùng tên của A. (Ngoài ra đối tượng Wraper có thể được bổ sung các phương thức khác nếu cần thiết). Đặc điểm này được đưa ra dựa trên nguyên tắc thiết kế “Law of Demeter” nói rằng không nên tham chiếu một đối tượng sâu hơn một lớp.  Các phương thức trong Adaptee được “nhúng” trong Wrapper bằng cách truyền lời gọi cùng với các tham số tới phương thức tương ứng trong Adaptee, và trả về kết quả giống như vậy. Các thành viên (thuộc tính, trường, sự kiện) được nhúng trong Wrapper có tính chất giống hệt như trong các lớp được nhúng (tên, kiểu dữ liệu, phạm vi truy cập...).  Từ các đặc điểm ở trên, có thể thấy rằng Wrapper Pattern cho phép một module chương trình tương tác được trong một môi trường khác biệt với môi trường phát triển của module đó (ví dụ C++ và Java).  **Khác biệt giữa Wrapper Pattern và Adapter Pattern**  Sự khác biệt giữa Wrapper và Adapter nằm ở mục đích sử dụng: Adapter Pattern định hướng cho một đối tượng đang tồn tại có thể làm việc được với các đối tượng khác và biến đổi logic theo một cách thức nào đó, trong khi Wrapper Pattern chỉ đơn thuần cung cấp một giao diện kết hợp các đối tượng được xây dựng từ cùng một ngôn ngữ hoặc khác ngôn ngữ, trên cùng một hệ điều hành hoặc trên những hệ điều hành khác nhau.  **Proxy Adapter Pattern**  Nếu một lớp Adapter đóng thêm vai trò như một proxy bảo vệ cho Adaptee thì ta có mô hình Proxy Adapter Pattern, trong trường hợp này chúng ta có thể đặt tên lớp với nghĩa kết hợp, ví dụ BankProxyAdapter.  **LỜI KẾT**  Bài viết này đưa ra một số mẫu thiết kế tiêu biểu giúp các bạn thấy được tầm quan trọng của design pattern trong việc nâng cao chất lượng phần mềm ở các yếu tố: hiệu suất ứng dụng, độ ổn định, tái sử dụng, tính bảo mật... Các bạn có thể tìm hiểu thêm về các mẫu thiết kế cao cấp khác ở rất nhiều tài liệu thiết kế phần mềm. | |