THỰC HÀNH TUẦN 08-09-10
MÔN KIẾN TRÚC VÀ THIẾT KẾ PHẦN MỀM

Mục tiêu:

- *Hiểu được các cách tạo mẫu thiết kế loại Creational Patterns, Structural Patterns và Behavioral Patterns.*
- *Vẽ được mô hình UML Class Diagram cho các mẫu thiết kế loại Creational Patterns, Structural Patterns và Behavioral Patterns.*
- *Hiện thực các mẫu Creational Patterns, Structural Patterns và Behavioral Patterns bằng một ngôn ngữ lập trình cụ thể (C#/Java).*
- *Cho một mô tả, lựa chọn được mẫu phù hợp cho mô tả đó. Hiện thực mẫu đã chọn bằng ngôn ngữ lập trình.*

Yêu cầu:

- *Sinh viên có thể sử dụng Visual Studio 2015/2017 với IIS, IIS Express HOẶC Eclipse Java EE Oxygen/Photon với GlassFish 5.*
- *Cuối mỗi buổi thực hành, SV phải nén (.rar hoặc .zip) thư mục làm bài và nộp lại bài tập đã thực hiện trong buổi đó.*

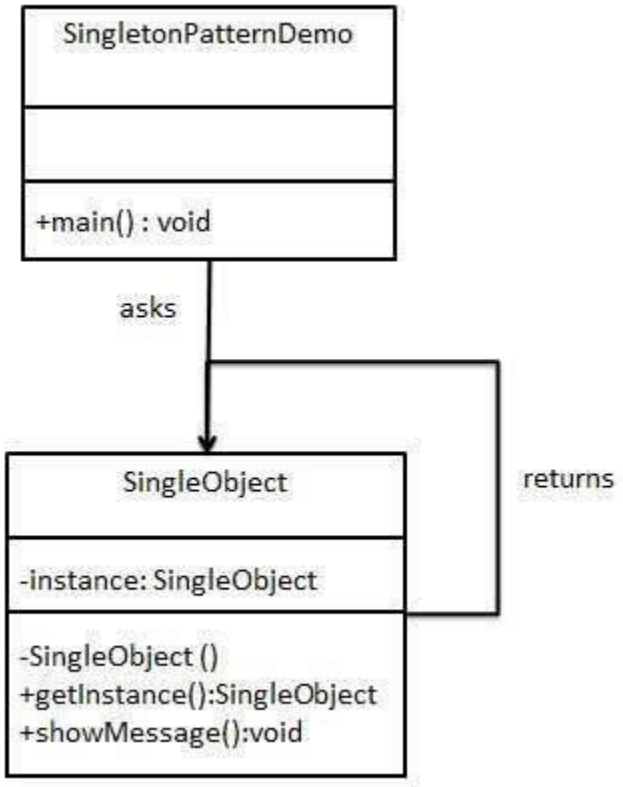
Phần 1: CREATIONAL PATTERNS

1.1. Singleton Pattern

Mẫu Singleton[1] (nhóm Creational Patterns), đảm bảo chỉ duy nhất 1 instance được tạo ra và cung cấp một method để (bên ngoài) có thể truy xuất được instance duy nhất.

Thực thi Singleton Pattern:

- private constructor để hạn chế truy cập từ class bên ngoài.
- Đặt private static final variable đảm bảo biến chỉ được khởi tạo trong class.
- Có một method public static để return instance được khởi tạo ở trên.



Sử dụng Singleton Pattern: Singleton chỉ tồn tại 1 instance nên thường được dùng cho các trường hợp giải quyết các bài toán cần truy cập vào các ứng dụng như: Shared resource, Logger, Configuration, Caching, Thread pool.

Thực thi Singleton Pattern

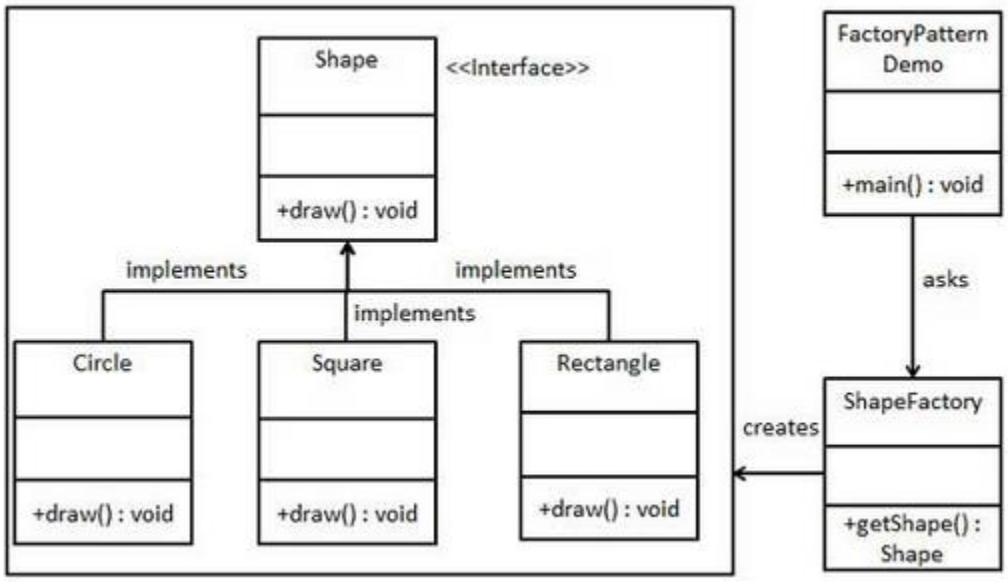
- Eager initialization
- Static block initialization
- Lazy Initialization
- Thread Safe Singleton
- Double Check Locking Singleton

[1] Singleton is a creational design pattern that lets you ensure that a class has only one instance and provide a global access point to this instance. (https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples)

- Bill Pugh Singleton Implementation
- Reflection
- Enum Singleton
- Serialization and Singleton

1.2. Factory Method Pattern

Factory Method is a creational design pattern that define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.



Thành phần của Factory Pattern gồm:

- *Super Class*: môt supper class trong Factory Pattern có thể là một interface, abstract class hay một class thông thường.
- *Sub Classes*: các sub class sẽ implement các phương thức của supper class theo nghiệp vụ riêng.
- *Factory Class*: một class chịu tránh nhiệm khởi tạo các đối tượng sub class dựa theo tham số đầu vào. Factory class sử dụng if-else hoặc switch-case để xác định sub class.

Factory Pattern được sử dụng trong trường hợp:

- Có một super class với nhiều class con và dựa trên yêu cầu input, cần trả về một sub class. Việc khởi tạo một lớp sẽ phụ thuộc vào lớp Factory thay vì do Client khởi tạo.
- Dùng Factory Pattern trong trường hợp cần làm giảm sự phụ thuộc giữa các module (loose coupling). Sử dụng hướng tiếp cận với Interface.
- Dùng Factory Pattern trong trường hợp cần tạo chương trình độc lập với những lớp cụ thể cần tạo 1 đối tượng, code ở phía client không thay đổi khi thay đổi logic ở factory hay sub class.
- Trường hợp cần mở rộng code sau khi thiết kế ứng dụng. Khi cần mở rộng, tạo ra sub class và implement thêm vào factory method.
- Trường hợp cần quản lý được life cycle của các object được tạo bởi Factory Pattern.

Bài tập Factory Pattern

1. Công ty bán xe hơi hiện tại bán 3 loại xe*: Honda, Nexus và Toyota*. Để hỗ trợ khách hàng xem thông tin về các chiếc xe này, đội ngũ công nghệ thông tin của công ty cần xây dụng ứng dụng cho phép xem thông tin. Mỗi loại xe có các thông tin về xe, cấu hình xe, nhà sản xuất và tính năng kỹ thuật khác nhau.

Tuy nhiên công ty còn có nhu cầu bán các loại xe khác (Mercedes, Porsche, Mazda …) trong tương lai, để giảm thiểu việc thay đổi code của ứng dụng, dùng Factory Method Pattern để thiết kế chương trình cho ứng dụng này.

- Vẽ mô hình lớp cho ứng dụng truy xuất thông tin xe hơi.
- Hiện thực ứng dụng này bằng ngôn ngữ lập trình cụ thể (Java/C#/C++).

2. Để truy cập thông tin dịch vụ và khuyến mãi của các ngân hàng, các hệ thống ngân hàng sẽ cung cấp thông tin liên quan. Cần thiết kế một chương trình cho phép người dùng (client) có thể truy cập và lấy các thông tin từ các ngân hàng mà không cần phải thay đổi source code. Hiện tại chương trình mới cung cấp cho người dùng về ngân hàng *VietinBank, Sacombank và ACB*.

   - Vẽ mô hình lớp cho ứng dụng truy xuất thông tin dịch vụ, khuyến mãi của ngân hàng.
   - Hiện thực ứng dụng này bằng ngôn ngữ lập trình cụ thể (Java/C#/C++).

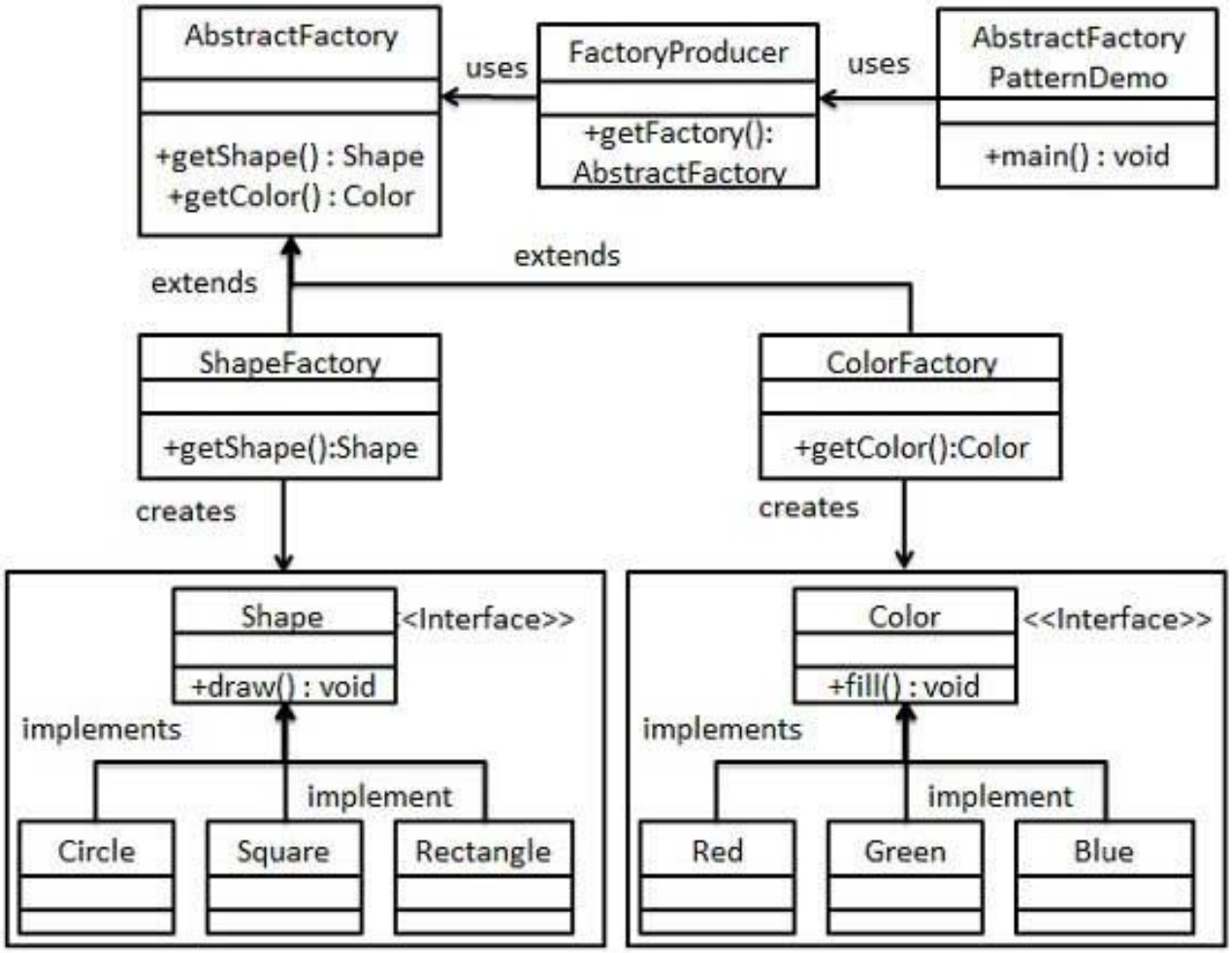
1.3. Abstract Factory Pattern

Abstract Factory is a creational design pattern that provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Là Pattern tạo ra Super Factory cho các Factory khác.

Một Abstract Factory Pattern bao gồm các thành phần:

- *AbstractFactory*: Khai báo dạng interface hoặc abstract class chứa các phương thức để tạo ra các đối tượng abstract.
- *ConcreteFactory*: Xây dựng, cài đặt các phương thức tạo các đối tượng cụ thể.
- *AbstractProduct*: Khai báo dạng interface hoặc abstract class để định nghĩa đối tượng abstract.
- *Product*: Cài đặt của các đối tượng cụ thể, cài đặt các phương thức được quy định tại Abstra*ctProduct.*
- *Clien*t: là đối tượng sử dụng AbstractFactory và các AbstractProduct thông qua Producer.



Bài tập Abstract Factory Pattern

1. Một công ty đồ nội thất chuyên sản xuất ghế: ghế nhựa và ghế gỗ. Với tình hình kinh doanh ngày càng thuận thợi nên công ty quyết định mở rộng thêm sản xuất bàn. Với lợi thế là đã có kinh nghiệm từ sản xuất ghế nên công ty vẫn giữ chất liệu là nhựa và gỗ cho sản xuất bàn. Tuy nhiên, quy trình sản xuất ghế/ bàn theo từng chất liệu là khác nhau.

   Nên công ty tách ra làm 2 nhà máy: 1 cho sản xuất vật liệu bằng nhựa, 1 cho sản xuất vật liệu bằng gỗ, nhưng cả 2 đều có thể sản xuất ghế và bàn. Khi khách hàng cần mua một món đồ nào, khách hàng chỉ cần đến cửa hàng để đặt mua. Khi có đơn đặt hàng của khách hàng, công ty xem xét thông tin ứng với từng hàng hóa và vật liệu sẽ được chuyển về phân xưởng tương ứng để sản xuất ra bàn và ghế.

2. Có 2 loại mặt hàng là quần áo và giày dép. Và cả 2 loại đều có 2 nhà máy sản xuất 1 là ở Anh và 1 là ở Mỹ. Khi được đem ra bán thì 2 loại mặt hàng được chia ra cho 2 đại lý phân phối chính. 1 đại lý chuyên cung cấp quần áo, 1 đại lý chuyên cung cấp giày dép.

   Khi khách hàng mua hàng thì sẽ được hỏi mua sản phẩm loại nào (quần áo hoặc giày dép) sau đó sẽ được hỏi mua hàng xuất xứ của quốc gia nào (Anh hoặc Mỹ). Sau khi mua hàng khách hàng sẽ được báo lại thông tin về sản phẩm mình vừa mua.

   Chương trình có thể mở rộng, ngoài bán quần áo, giày dép có thể bán thêm túi xách… Các mặt hàng này có thể nhập về từ nhiều nước khác nhau khác như Pháp, Ý, …

3. Abstract Factory Pattern có thể dùng trong các bộ công cụ giao diện (UI toolkits). Trong các hệ điều hành Windows, Mac và Linux, thành phần UI bao gồm: cửa sổ giao diện (windows), nút lệnh (buttons) và ô nhập dữ liệu (textfields). Việc thực thi các thành phần giao diện này However, the implementation of these widgets vary across platforms. You could write a platform independent client thanks to the Abstract Factory implementation.

1.4. Prototype Pattern

Prototype is a creational design pattern that specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Prototype lets you produce new objects by copying existing ones without compromising their internals. The new object is an exact copy of the prototype but permits modification without altering the original.

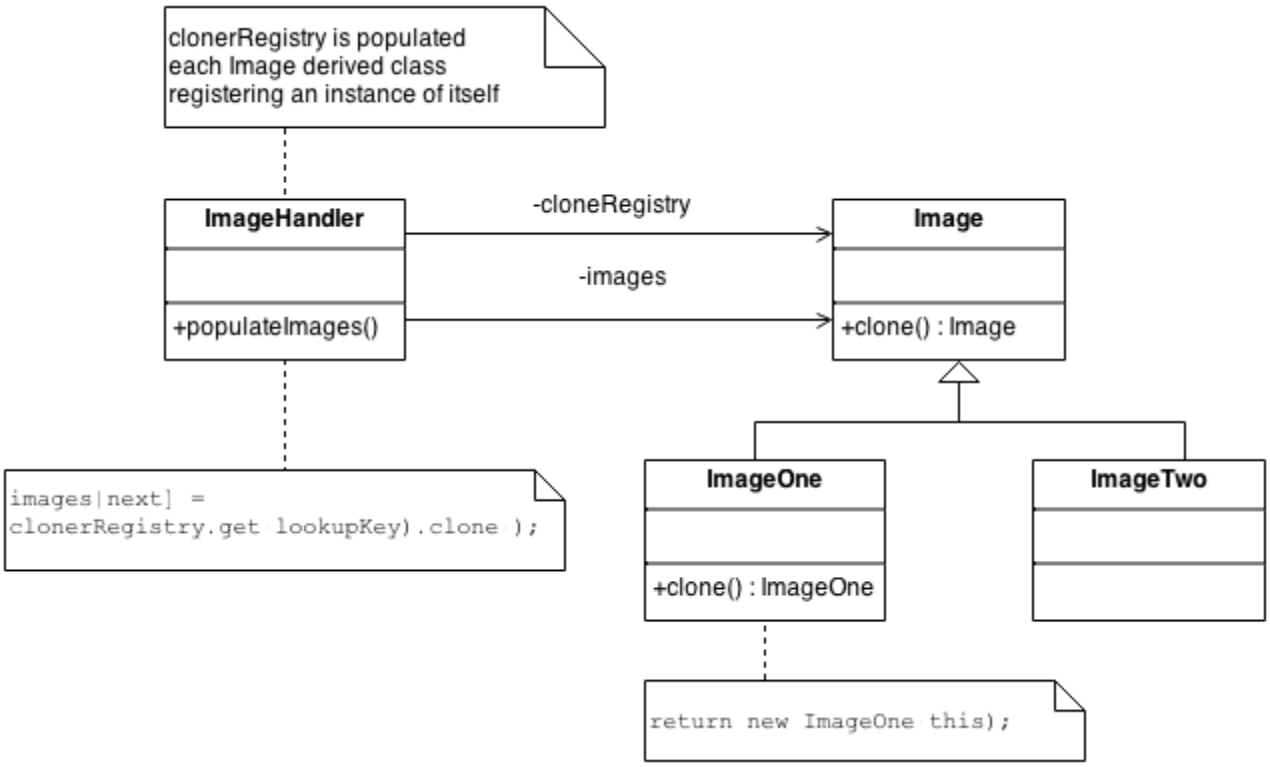
Prototype pattern có nhiệm vụ khởi tạo một đối tượng bằng cách *clone* một đối tượng đã tồn tại thay vì khởi tạo với từ khoá *new*. Đối tượng mới là một bản sao có thể giống 100% với đối tượng gốc, có thể thay đổi dữ liệu mà không ảnh hưởng đến đối tượng gốc. Prototype Pattern được dùng khi việc tạo một object tốn nhiều thời gian trong khi chương trình đã có một object tương tự tồn tại.

Prototype Pattern gồm các thành phần:

- *Prototype*: là class, interface hoặc abtract class cho việc clone.
- *ConcretePrototype* class: các lớp này thực thi interface (hoặc kế thừa từ lớp abstract) được cung cấp bởi Prototype để copy.
- *Client* class: tạo mới object bằng cách gọi Prototype thực hiện clone chính nó.

Sử dụng Prototype:

- Cần 1 ọbject mới khác dựa trên object ban đầu mà không sử dụng toán tử *new* hay các hàm *contructor* để khởi tạo.
- Khởi tạo đối tượng lúc *run-time*.
- Truyền tham trị trong lập trình.



Bài tập Prototype Pattern

1. Một công ty có cấu hình máy tính đều giống nhau cho tất cả nhân viên, bao gồm: Hệ điều hành (os), Phần mềm văn phòng (office), Phần mềm diệt virus (antivirus), Trình duyệt (Browser), và một số phần mềm khác (others) tùy theo nhu cầu của mỗi nhân viên sẽ được cài đặt thêm ứng dụng. Việc cài đặt tất cả phần mềm trên rất tốn thời gian, nên anh IT đã nghĩ ra một cách là sẽ tạo ra một bản chuẩn cho một máy tính và có thể clone() lại cấu hình đó cho một nhân viên khác mà không cần phải cài đặt lại từ đầu.

2. Một bàn cờ vua gồm có: 8 hàng (row) và 8 cột (column). Tương ứng với hàng và cột là các ô (cell) được tô màu đen (black) và trắng (white).

3. Suppose we have an Object that loads data from database. Now we need to modify this data in our program multiple times, so it's not a good idea to create the Object using new keyword and load all the data again from database.

   The better approach would be to clone the existing object into a new object and then do the data manipulation.

   Prototype design pattern mandates that the Object which you are copying should provide the copying feature. It should not be done by any other class. However whether to use shallow or deep copy of the Object properties depends on the requirements and its a design decision.

1.5. Builder Pattern

Builder Pattern được xây dựng để khắc phục một số nhược điểm của Factory Pattern và Abstract Factory Pattern khi mà Object có nhiều thuộc tính. Trường hợp Factory Pattern và Abstract Factory Pattern khi Object có nhiều thuộc tính thì cần nhiều tham số phải truyền vào từ phía client tới Factory Class.

- Một số tham số có thể là optional nhưng trong Factory Pattern, cần phải gửi tất cả tham số, với tham số tùy chọn nếu không nhập gì thì sẽ truyền là null.
- Nếu một Object có quá nhiều thuộc tính thì việc tạo sẽ phức tạp.

Thành phần cơ bản của Builder Pattern:

- *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. *ConcreteBuilder* 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*/ *Client*: là nơi sẽ gọi tới Builder để tạo ra đối tượng.

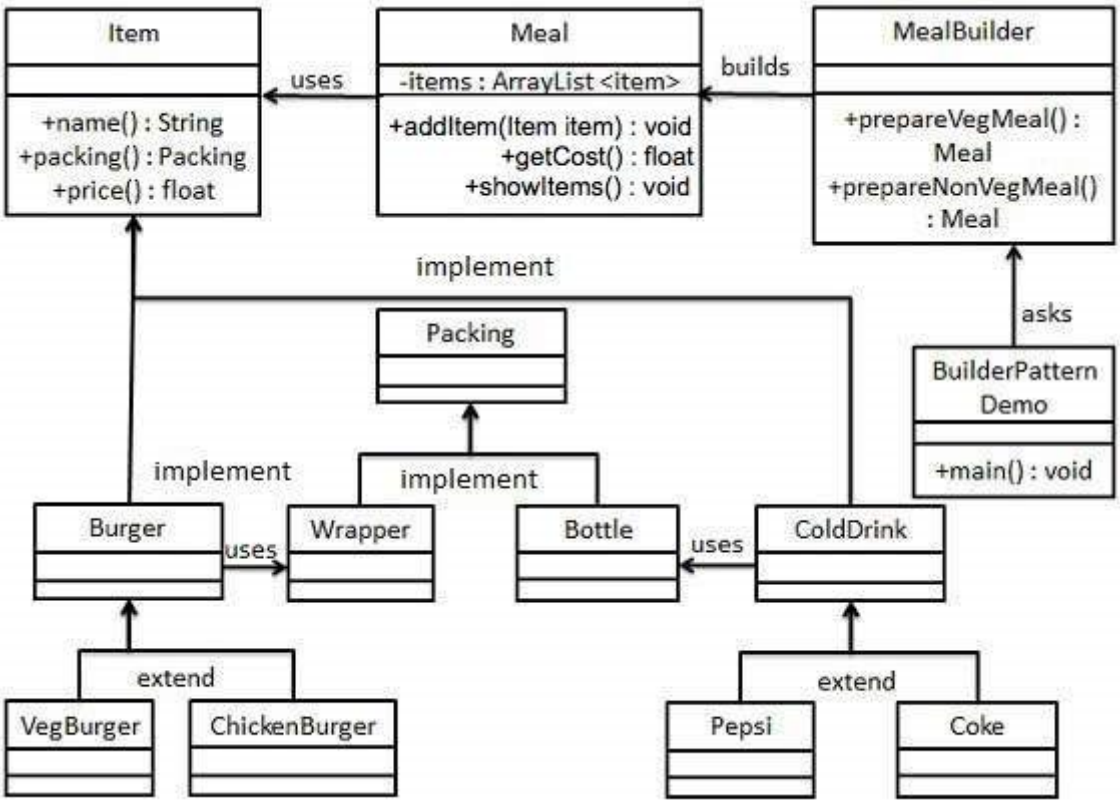
Bài tập Builder Pattern

1. Một tài khoản ngân hàng bao gồm các thông tin: Tên chủ tài khoản, số tài khoản, địa chỉ email, nhận thông báo, sử dụng mobile banking. Một tài khoản được tạo phải có tên chủ tài khoản và số tài khoản. Các thông tin khác tùy theo nhu cầu của khách hàng có thể đăng ký sử dụng.

2. Sử dụng Builder Pattern cho việc gọi món tại một cửa hàng thức ăn nhanh. We have considered a business case of fast-food restaurant where a typical meal could be a burger and a cold drink. Burger could be either a Veg Burger or Chicken Burger and will be packed by a wrapper. Cold drink could be either a coke or pepsi and will be packed in a bottle.

   We are going to create an *Item* interface representing food items such as burgers and cold drinks and concrete classes implementing the *Item* interface and a *Packing* interface representing packaging of food items and concrete classes implementing the *Packing* interface as burger would be packed in wrapper and cold drink would be packed as bottle.

   We then create a *Meal* class having *ArrayList* of *Item* and a *MealBuilder* to build different types of *Meal* objects by combining *Item*. *BuilderPatternDemo*, our demo class will use *MealBuilder* to build a *Meal*.


Item
+name() : String
+packing() : Packing
+price() : float
uses
Meal
-items : ArrayList <item>
+addItem(Item item) : void
+getCost() : float
+showItems() : void
builds
MealBuilder
+prepareVegMeal() : Meal
+prepareNonVegMeal() : Meal
implement
asks
Packing
BuilderPattern Demo
+main() : void
implement
implement
Burger
uses
Wrapper
Bottle
uses
ColdDrink
extend
extend
VegBurger
ChickenBurger
Pepsi
Coke

Phần 2: STRUCTURAL PATTERNS

2.1. Adapter Pattern

Adapter Pattern cho phép các inteface không liên quan tới nhau có thể làm việc cùng nhau. Đối tượng giúp kết nối các interface là Adapter.

Adapter Pattern giữ vai trò trung gian giữa hai lớp, Adapter Pattern chuyển đổi interface của một hay nhiều lớp có sẵn thành một interface khác, thích hợp cho lớp cần sử dụng. Adapter Pattern cho phép các lớp có các interface khác nhau có thể dễ dàng giao tiếp tốt với nhau thông qua interface trung gian, không cần thay đổi code có sẵn.

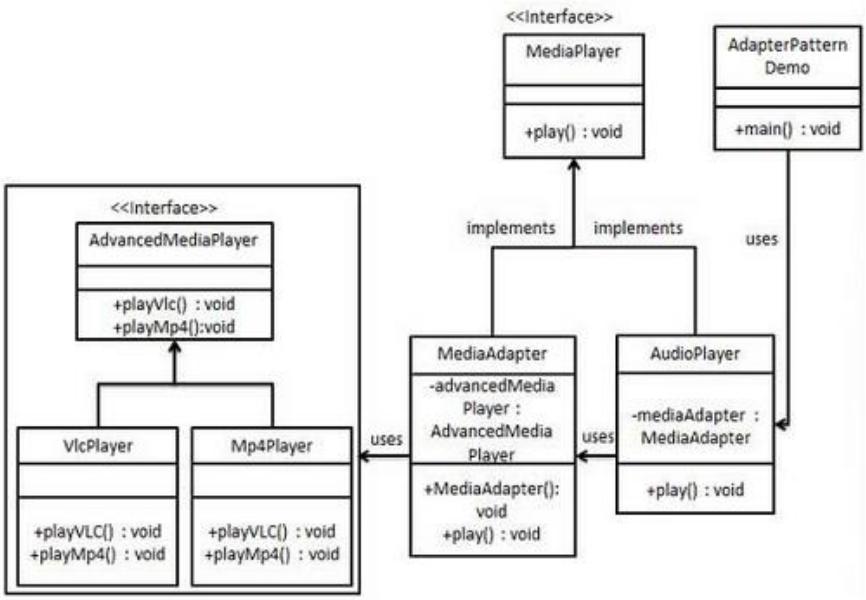
Adapter Pattern gồm thành phần:

- Adaptee: định nghĩa interface không tương thích, cần được tích hợp vào.
- Adapter: lớp tích hợp, giúp interface không tương thích tích hợp được với interface đang làm việc. Thực hiện việc chuyển đổi interface cho Adaptee và kết nối Adaptee với Client.
- Target: một interface chứa các chức năng được sử dụng bởi Client (domain specific).
- Client: lớp sử dụng các đối tượng có interface Target.

Hiện thực Adapter Pattern (2):

- *Object Adapter – Composition:* một lớp mới (Adapter) sẽ tham chiếu đến một hoặc nhiều đối tượng của lớp có sẵn với interface không tương thích (Adaptee), đồng thời cài đặt interface mà người dùng mong muốn (Target). Trong lớp mới này, khi cài đặt các phương thức của interface người dùng mong muốn, sẽ gọi phương thức cần thiết thông qua đối tượng thuộc lớp có interface không tương thích.

- *Class Adapter – Inheritance (*Kế thừa): trong mô hình này, một lớp mới (Adapter) sẽ kế thừa lớp có sẵn với interface không tương thích (Adaptee), đồng thời cài đặt interface mà người dùng mong muốn (Target). Trong lớp mới, khi cài đặt các phương thức của interface người dùng mong muốn, phương thức này sẽ gọi các phương thức cần thiết mà nó thừa kế được từ lớp có interface không tương thích.



So sánh Class Adapter với Object Adapter:

- Sự khác biệt chính là Class Adapter sử dụng Inheritance (kế thừa) để kết nối Adapter và Adaptee trong khi Object Adapter sử dụng Composition để kết nối Adapter và Adaptee.
- Trong cách tiếp cận Class Adapter, nếu một Adaptee là một class và không phải là một interface thì Adapter sẽ là một lớp con của Adaptee. Do đó, nó sẽ không phục vụ tất cả các lớp con khác theo cùng một cách vì Adapter là một lớp phụ cụ thể của Adaptee.

Có thể dùng Adapter Pattern trong những trường hợp sau:

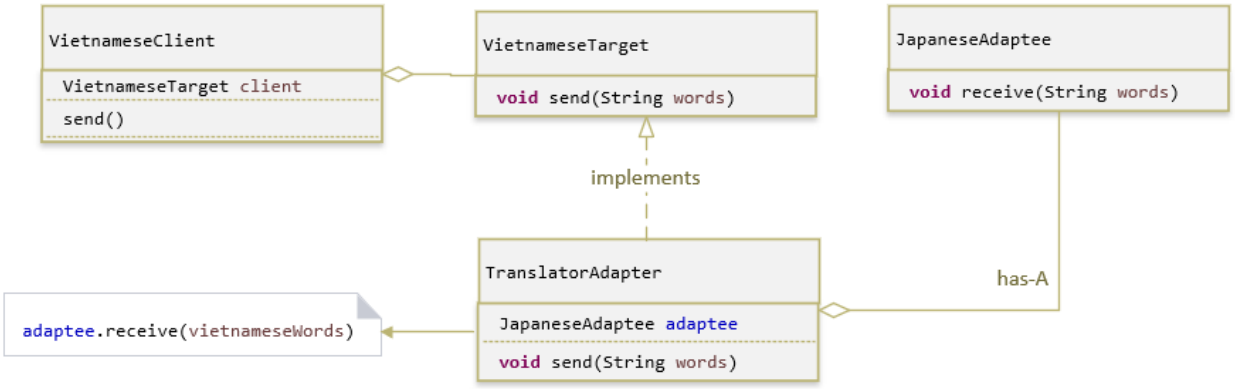
- Adapter Pattern giúp nhiều lớp có thể làm việc với nhau dễ dàng mà bình thường không thể. Một trường hợp thường gặp phải và có thể áp dụng Adapter Pattern là khi không thể kế thừa lớp A, nhưng muốn một lớp B có những xử lý tương tự như lớp A. Khi đó chúng ta có thể cài đặt B theo Object Adapter, các xử lý của B sẽ gọi những xử lý của A khi cần.
- Khi muốn sử dụng một lớp đã tồn tại trước đó nhưng interface sử dụng không phù hợp như mong muốn.
- Khi muốn tạo ra những lớp có khả năng sử dụng lại, chúng phối hợp với các lớp không liên quan hay những lớp không thể đoán trước được và những lớp này không có những interface tương thích.
- Cần phải có sự chuyển đổi interface từ nhiều nguồn khác nhau.
- Khi cần đảm bảo nguyên tắc *Open/ Close* trong một ứng dụng.

Bài tập Adapter Pattern

1. Giả sử có 1 xưởng sản xuất đồng hồ chuyên tạo ra 2 loại đồng hồ là đống hồ dây cót và đồng hồ quả lắc. Bây giờ họ muốn mở rộng sản xuất them đồng hồ điện tử nhưng dây chuyền sản xuất không cho phép do đó họ phải nhờ một xưởng chuyên sản xuất đồng hồ điện tử chế tạo sau đó đem về gián nhãn mác của họ.

2. Một người Việt muốn trao đổi với một người Nhật. Tuy nhiên, 2 người này không biết ngôn ngữ của nhau nên cần phải có một người để chuyển đổi từ ngôn ngữ tiếng Việt sang ngôn ngữ tiếng Nhật. Dùng Adapter Pattern?

   - Client: người Việt sẽ là Client, vì người Việt cần gửi một số message cho người Nhật.
   - Target: đây là nội dung message được Client cung cấp cho thông dịch viên (Translator / Adapter).
   - Adapter: thông dịch viên (Translator) sẽ là Adapter, nhận message tiếng Việt từ Client và chuyển đổi message sang tiếng Nhật trước khi gởi cho người Nhật.
   - Adaptee: đây là interface hoặc class được người Nhật sử dụng để nhận message được chuyển đổi từ thông dịch viên (Translator).



2.2. Bridge Pattern

Bridge Pattern tách tính trừu tượng (abstraction) ra khỏi tính hiện thực (implementation).

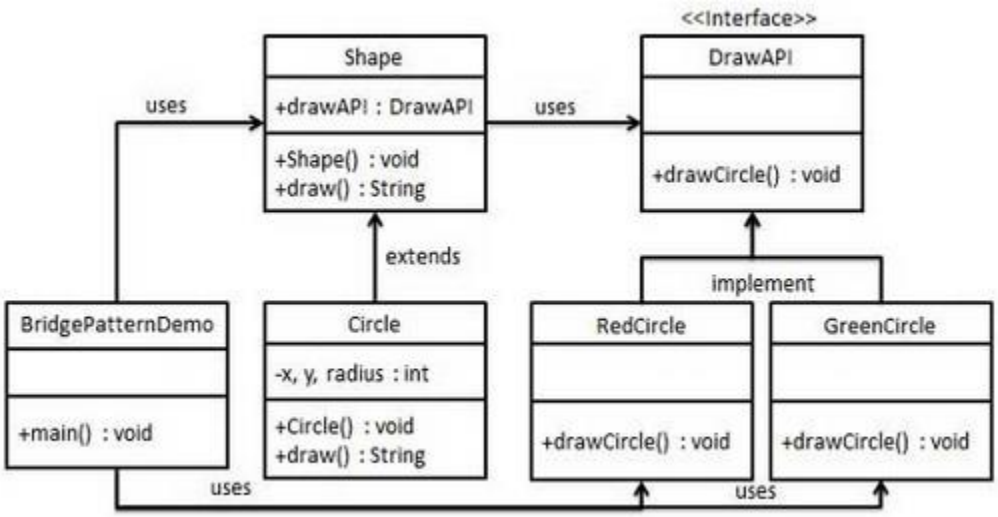
Với cách thiết kế cũ, một class chứa rất nhiều xử lý, với Bridge Pattern, các xử lý không còn để chung trong class đó nữa. Bridge Pattern sẽ tạo ra một class khác và di chuyển các xử lý đó qua class mới. Khi đó, trong lớp cũ sẽ giữ một đối tượng thuộc về lớp mới, và đối tượng này sẽ chịu trách nhiệm xử lý thay cho lớp ban đầu.

Bridge Pattern tương tự với Adapter Pattern ở chỗ nhờ vào một lớp khác để thực hiện một số xử lý nào đó. Tuy nhiên, ý nghĩa và mục đích sử dụng của hai mẫu thiết kế này hoàn toàn khác nhau:

- Adapter Pattern được dùng để biến đổi một class/ interface sang một dạng khác có thể sử dụng được. Adapter Pattern giúp các lớp không tương thích hoạt động cùng nhau mà bình thường là không thể.
- Bridge Pattern được sử dụng được sử dụng để tách thành phần trừu tượng (abstraction) và thành phần thực thi (implementation) riêng biệt.
- Adapter Pattern làm cho mọi thứ có thể hoạt động với nhau sau khi chúng đã được thiết kế (đã tồn tại). Bridge Pattern được thiết kế trước khi phát triển hệ thống để Abstraction và Implementation có thể thực hiện một cách độc lập.

Một Bridge Pattern bao gồm:

- Client: sử dụng các chức năng thông qua Abstraction.
- Abstraction: định ra một abstract interface quản lý việc tham chiếu đến đối tượng hiện thực cụ thể (Implementor).
- Refined Abstraction (AbstractionImpl): hiện thực (implement) các phương thức đã được định ra trong Abstraction bằng cách sử dụng một tham chiếu đến một đối tượng của Implementer.
- Implementor: định ra các interface cho các lớp hiện thực. Thông thường Implementor là interface định ra các tác vụ nào đó của Abstraction.
- ConcreteImplementor: hiện thực Implementor interface.



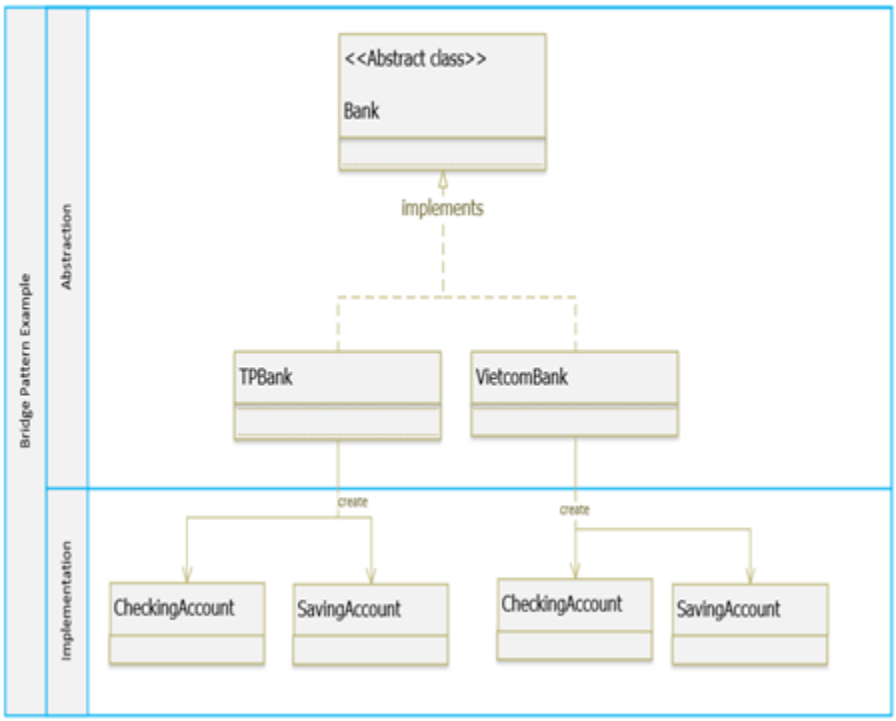
Sử dụng Bridge Pattern trong trường hợp:

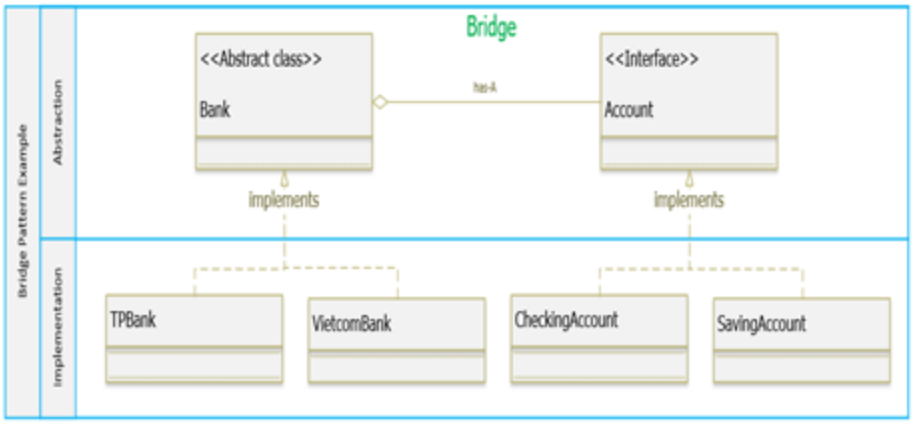
- Tách ràng buộc giữa Abstraction và Implementation, để có thể dễ dàng mở rộng độc lập nhau.
- Thay đổi được thực hiện trong implement không ảnh hưởng đến phía client.

Bài tập Bridge Pattern

1. Một hệ thống ngân hàng cung cấp các loại tài khoản khác nhau cho khách hàng, chẳng hạn: Checking account và Saving account. Với cách thiết kế như vậy, khi hệ thống cần cung cấp thêm một loại tài khoản khác, developer phải tạo class mới cho tất cả các ngân hàng, số lượng class tăng lên rất nhiều.

   Sử dụng Bridge Pattern để tái cấu trúc lại hệ thống khi có thêm một loại tài khoản mới, chỉ cần thêm vào một implement mới cho Account, các thành phần khác của Bank không bị ảnh hưởng. Hoặc cần thêm một ngân hàng mới, chẳng hạn VietinBank chúng ta chỉ cần thêm implement mới cho Bank, các thành phần khác cũng không bị ảnh hưởng và số lượng class chỉ tăng lên 1.





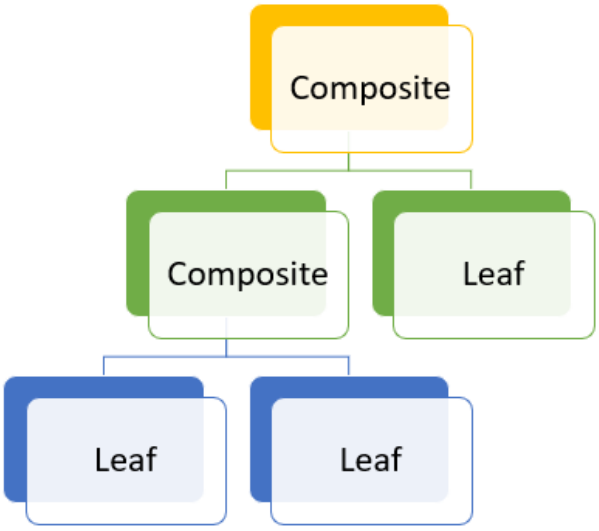
2. Mỗi hệ điều hành Windows, Linux, MacOS có cách hiển thị khác nhau các đối tượng hình học: hình chữ nhật, hình tròn. Khi hiện thực, cần vẽ các hình này trên các hệ điều hành, sau đó hiển thị các hình này lên mỗi hệ điều hành cụ thể.

   Abstraction: Hiển thị (thuộc tính Vẽ, phương thức show()) - Implementor: Vẽ (Quan hệ Aggregation, các phương thức hỗ trợ vẽ)
   Windows, Linux, MacOS extends HienThi và override show()
   HCN, HinhTron, HinhVuong extends/implements Vẽ.

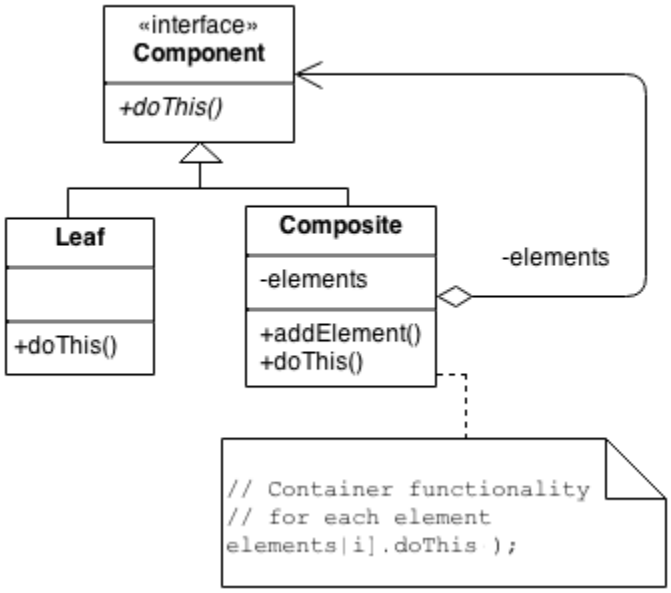
3. Giả sử ta có 3 loại máy tính mỗi máy tính có sử dụng 1 loại hệ điều hành khác nhau. Như máy Dell thì sử dụng Windows, máy Apple thì sử dụng MacOS, máy HP sử dụng hệ điều hành Linux. Thực hiện Bridge Pattern cho biết khi sử dụng máy loại nào thì Hệ Điều Hành của máy đó sẽ chạy.

2.3. Composite Pattern

Composite Pattern là sự tổng hợp những thành phần có quan hệ với nhau để tạo ra thành phần lớn hơn dùng cấu trúc phân cấp. Composite Pattern cho phép thực hiện các tương tác với tất cả đối tượng trong mẫu tương tự nhau. (Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.)



Composite Pattern sử dụng khi xử lý một nhóm đối tượng tương tự theo cách xử lý 1 object. Composite Pattern sắp xếp các object theo cấu trúc cây để diễn giải 1 phần cũng như toàn bộ hệ thống phân cấp. Composite Pattern tạo một lớp chứa nhóm đối tượng, cung cấp các cách để sửa đổi nhóm của cùng 1 object.

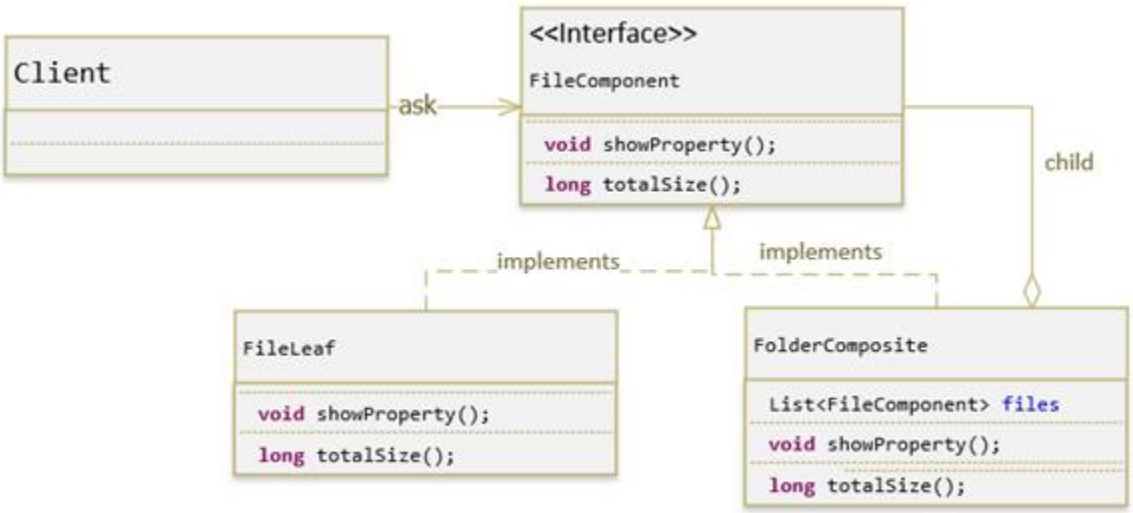


- Participants

- Component
  - Declares the interface for objects in the composition
  - Implements default behavior for the interface common to all classes, as appropriate
- Leaf
  - Represents leaf objects in the composition
  - Defines primitives in the system
- Composite
  - Defines behavior for components having children
  - Stores children components

Một Composite Pattern bao gồm các thành phần:

- Base Component: là một interface hoặc abstract class quy định các method chung cần phải có cho tất cả các thành phần tham gia vào mẫu này.
- Leaf: là lớp hiện thực (implements) các phương thức của Component.
- Composite: lưu trữ tập hợp các Leaf và cài đặt các phương thức của Base Component. Composite cài đặt các phương thức được định nghĩa trong interface Component bằng cách ủy nhiệm cho các thành phần con xử lý.
- Client: sử dụng Base Component để làm việc với các đối tượng trong Composite.

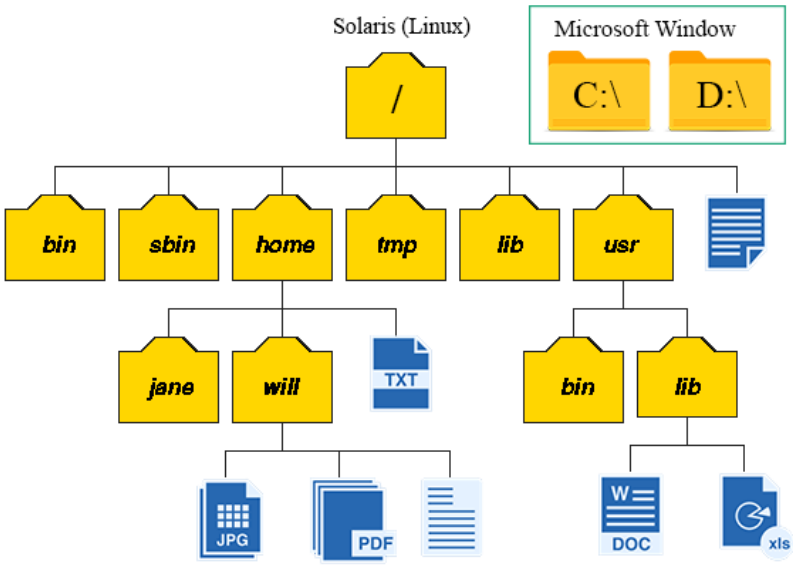


Sử dụng Composite Pattern trong trường hợp:

- Composite Pattern chỉ nên được áp dụng khi nhóm đối tượng phải hoạt động như một đối tượng duy nhất (theo cùng một cách).
- Composite Pattern có thể được sử dụng để tạo ra một cấu trúc giống như cấu trúc cây.

Bài tập Composite Pattern

1. Chương trình quản lý một hệ thống tập tin với cấu trúc:



Hệ thống gồm các thư mục (folder – composite), cũng như các nút lá là các tập tin (file – leaf). Một folder có thể chứa một hoặc nhiều file hoặc folder. Do đó, folder là một đối tượng phức tạp và file là một đối tượng đơn giản. File và Folder có nhiều thao tác và thuộc tính chung, chẳng hạn như: di chuyển, sao chép, liệt kê hoặc các thuộc tính thư mục như tên tập tin và kích thước.

Để quản lý file và folder thống nhất, xây dựng Interface có đầy đủ các phương thức và thuộc tính chung cho cả file và folder.

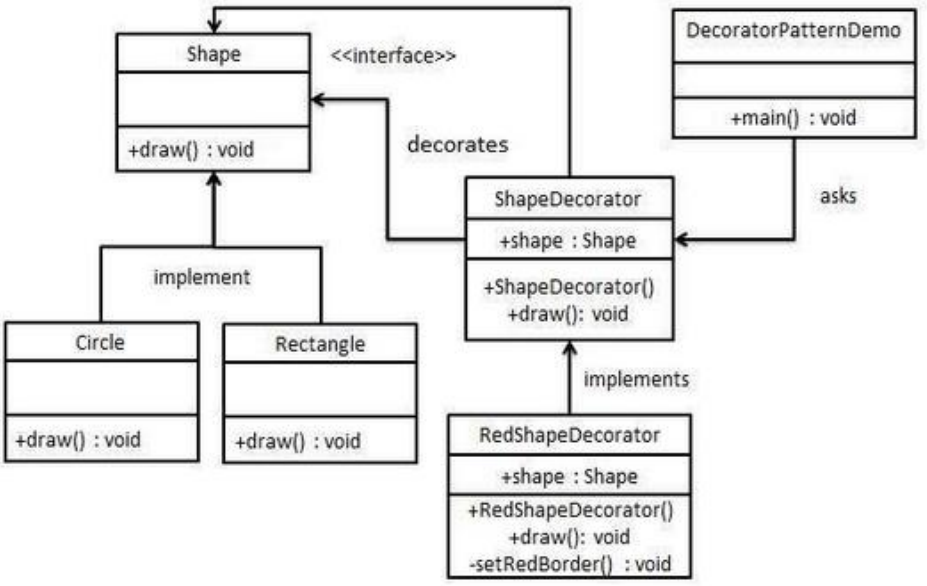
2. Chúng ta muốn xây dựng một dịch vụ cung cấp các pin sạc cho các khách hàng sử dụng chúng. Pin sạc được xây dựng thông qua 1 trong 2 phương pháp sau:
   - Tạo từng tế bào pin (cell), đây là loại pin cơ bản, mỗi cell có một công suất maximum xác định, tại từng thời điểm sử dụng, nó còn giữ một mức năng lượng xác định (nhỏ hơn hay bằng công suất maximum). Mỗi cell phải cung cấp 2 dịch vụ thiết yếu: nạp năng lượng vào và thải năng lượng ra cho khách hàng dùng.
   - Tạo pin có công suất lớn theo cơ chế tích hợp các pin có sẵn. Ta có thể ghép nhiều pin có sẵn để tạo ra một pin tích hợp có công suất lớn theo yêu cầu, công suất của pin này là tổng công suất của các pin thành phần, nó cũng cung cấp 2 dịch vụ thiết yếu: nạp năng lượng vào và thải năng lượng ra cho khách hàng dùng..

Như trên, về mặt vật lý, ta đã chế được 2 loại pin khác nhau (cell và battery) và có thể còn nhiều loại pin khác nữa. Tuy nhiên, code chương trình ứng dụng cần độc lập hoàn toàn với các loại pin trên. Để giải quyết yêu cầu chính đáng này của chương trình, ta phải thiết kế hệ thống cung cấp pin sạc như thế nào?

2.4. Decorator Pattern

Decorator pattern cho phép người dùng thêm chức năng mới vào đối tượng hiện tại mà không muốn ảnh hưởng đến các đối tượng khác. Kiểu thiết kế này có cấu trúc hoạt động như một lớp bao (wrap) cho lớp hiện có. Mỗi khi cần thêm tính năng mới, đối tượng hiện có được wrap trong một đối tượng mới (decorator class). (Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.)

Decorator pattern sử dụng composition thay vì inheritance (thừa kế) để mở rộng đối tượng.



Các thành phần trong mẫu thiết kế Decorator:

- *Component*: là một interface quy định các method chung cần phải có cho tất cả các thành phần tham gia vào Decorator Pattern.
- *ConcreteComponent*: là lớp hiện thực (implements) các phương thức của Component.
- *Decorator*: là một abstract class dùng để duy trì một tham chiếu của đối tượng Component và đồng thời cài đặt các phương thức của Component interface.
- *ConcreteDecorator*: là lớp hiện thực (implements) các phương thức của Decorator, nó cài đặt thêm các tính năng mới cho Component.
- *Client*: đối tượng sử dụng Component.

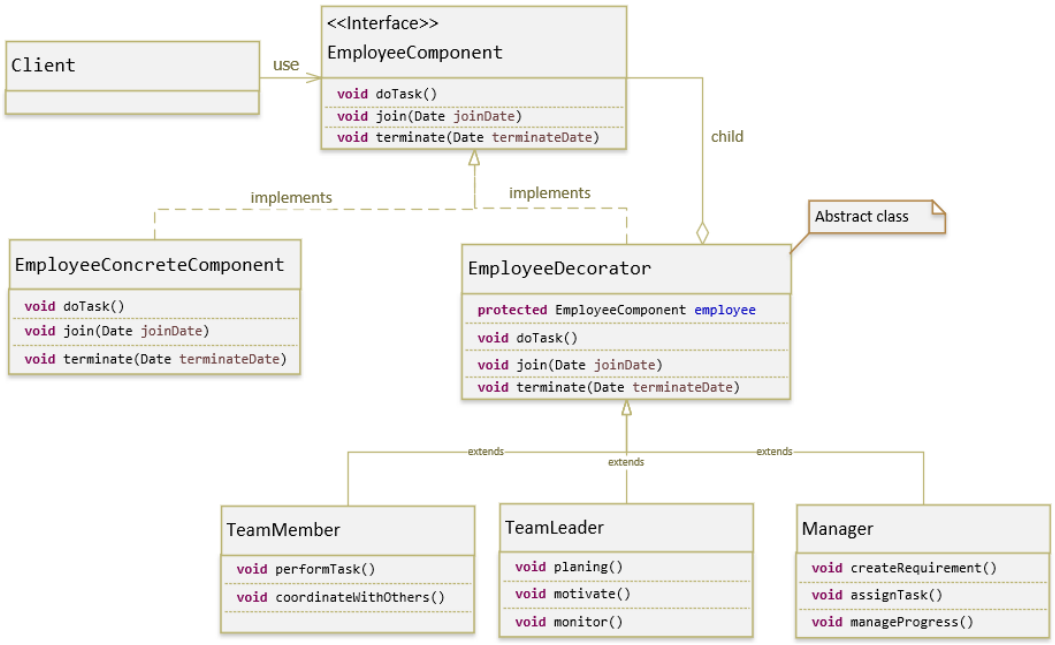
Bài tập Decorate Pattern

1. Hệ thống quản lý dự án, nơi nhân viên đang làm việc với các vai trò khác nhau, chẳng hạn như thành viên nhóm (team member), trưởng nhóm (team lead) và người quản lý (manager). Một thành viên trong nhóm chịu trách nhiệm thực hiện các nhiệm vụ được giao và phối hợp với các thành viên khác để hoàn thành nhiệm vụ nhóm. Mặt khác, một trưởng nhóm phải quản lý và cộng tác với các thành

viên trong nhóm của mình và lập kế hoạch nhiệm vụ của họ. Tương tự như vậy, một người quản lý có thêm một số trách nhiệm đối với một trưởng nhóm như quản lý yêu cầu dự án, tiến độ, phân công công việc.

Thành phần tham gia vào hệ thống và hành vi:

- Employee: thực hiện công việc (doTask), tham gia vào dự án (join), rời khỏi dự án (terminate).
- Team member: báo cáo task được giao (report task), cộng tác với các thành viên khác (coordinate with others).
- Team lead: lên kế hoạch (planning), hỗ trợ các thành viên phát triển (motivate), theo dõi chất lượng công việc và thời gian (monitor).
- Manager: tạo các yêu cầu dự án (create requirement), giao nhiệm vụ cho thành viên (assign task), quản lý tiến độ dự án (progress management).



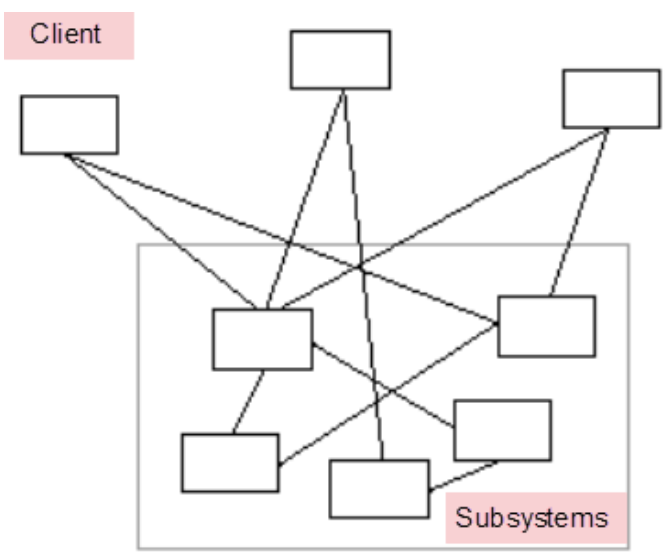
2. Một cửa hàng game có bán 2 loại game chính là game hành động và game đua xe. Mỗi loại game đều có tên và thể loại riêng, với game đua xe có thêm loại hình đua xe, với game hành động có thêm kiểu hành động.

   Nhưng game hành động thì có thêm phụ kiện là tay cầm, game đua xe thì có thêm phụ kiên là cái vô lăng để lái. Dùng Decorate Pattern để thiết kế chương trình này.

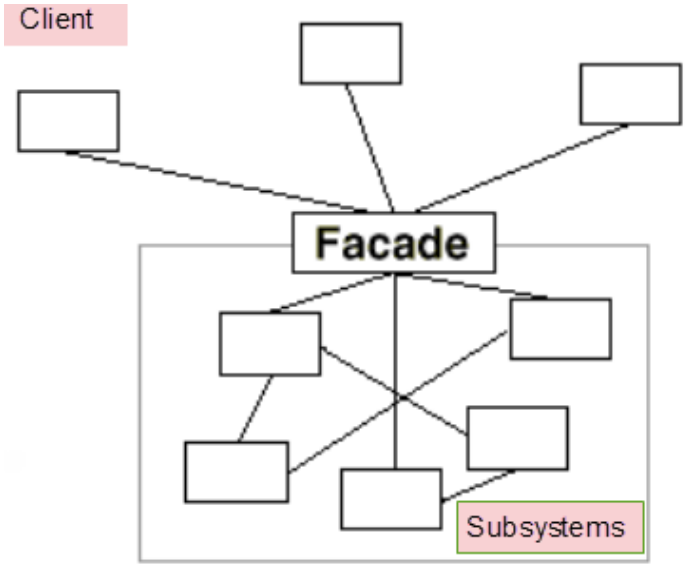
2.5. Façade Pattern

Facade Pattern cung cấp một giao diện chung đơn giản thay cho một nhóm các giao diện có trong một hệ thống con (subsystem). Facade Pattern định nghĩa một giao diện ở một cấp độ cao hơn để giúp cho người dùng có thể dễ dàng sử dụng hệ thống con này.

Facade Pattern cho phép các đối tượng truy cập trực tiếp giao diện chung này để giao tiếp với các giao diện có trong hệ thống con. Mục tiêu là che giấu các hoạt động phức tạp bên trong hệ thống con, làm cho hệ thống con dễ sử dụng hơn.



Hệ thống cũ

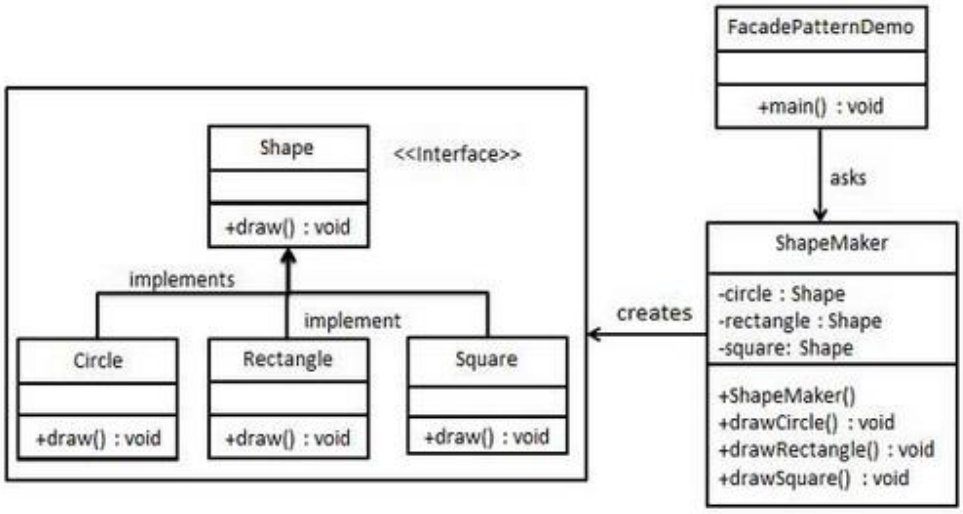


Hệ thống sử dụng Façade Pattern

Façade Pattern tương tự với Adapter Pattern. Hai Pattern này làm việc theo cùng một cách, nhưng mục đích sử dụng của chúng khác nhau. Adapter Pattern chuyển đổi mã nguồn để làm việc được với mã nguồn khác. Nhưng Façade Pattern cho phép bao bọc mã nguồn gốc để nó có thể giao tiếp với mã nguồn khác dễ dàng hơn.

Các thành phần của một Facade Pattern:

- Façade: biết rõ lớp của hệ thống con nào đảm nhận việc đáp ứng yêu cầu của client, sẽ chuyển yêu cầu của client đến các đối tượng của hệ thống con tương ứng.
- Subsystems: cài đặt các chức năng của hệ thống con, xử lý công việc được gọi bởi Facade. Các lớp này không cần biết Facade và không tham chiếu đến nó.
- Client: đối tượng sử dụng Facade để tương tác với các subsystem.



Sử dụng Facade Pattern trong trường hợp:

- Khi hệ thống có rất nhiều lớp làm người sử dụng rất khó để có thể hiểu được quy trình xử lý của chương trình. Sử dụng Facade Pattern để tạo ra một giao diện đơn giản cho người sử dụng một hệ thống phức tạp.
- Khi bạn muốn bao bọc, che giấu tính phức tạp trong các hệ thống con đối với phía Client.

Bài tập Façade Pattern

1. Một công ty bán hàng online, chẳng hạn Tiki cung cấp nhiều lựa chọn cho khách hàng khi mua sản phẩm. Khi một sản phẩm được mua, cần trải qua các bước xử lý: lấy thông tin về tài khoản mua hàng, thanh toán, vận chuyển, gửi Email/ SMS thông báo.

   Ứng dụng thiết kế với Facade Pattern, bao gồm các lớp:

   - Thông tin về tài khoản (AccountService): lấy thông tin cơ bản của khách hàng thông qua email được cung cấp.
   - Dịch vụ thanh toán (PaymentService): có thể thanh toán thông qua Paypal, thẻ tín dụng (Credit Card), tài khoản ngân hàng trực tuyến (E-banking), Tiền mặt (cash).
   - Dịch vụ vận chuyển (ShippingService): có thể chọn Free Shipping, Standard Shipping, Express Shipping.
   - Dịch vụ email (EmailService): có thể gửi mail cho khách hàng về tình hình đặt hàng, thanh toán, vận chuyển, nhận hàng.
   - Dịch vụ tin nhắn (SMS): có thể gửi thông báo SMS cho khách hàng khi thanh toán online.
   - ShopFacade: là một Facade Pattern, class này bao gồm các dịch vụ có bên trong hệ thống. Nó cung cấp một vài phương thức để Client có thể dễ dàng mua hàng. Tùy vào nghiệp vụ mà nó sẽ sử dụng những dịch tương ứng, chẳng hạn dịch vụ SMS chỉ được sử dụng nếu khách hàng đăng ký mua hàng thông qua hình thức thanh toán online (Paypal, E-banking, …).
   - Client: là người dùng cuối sử dụng ShopFacade để mua hàng.

2. Để mua được chiệc điện thoại di động, cần qua các bước: xem điện thoại, đặt hàng, sau đó thanh toán. Trong cửa hàng điện thoại có bán rất nhiều hãng điện thoại khác nhau. Để mua một chiếc điện thoại người dùng cần phải thông qua 3 bước, mà mỗi bước là một cách xử lý phức tạp. Thay vì phải viết code xử lý trong từng đối tượng, dùng Façade Pattern xử lý trong một class duy nhất và chỉ cần biết để mua điện thoại nào đó thì ta sẽ thực hiện qua 3 bước kia mà không quan tâm ba bước kia được xử lý phức tạp ra sao.
3. Áp dụng mô hình Façade cho quy trình khi đi vào nhận phòng và trả phòng khi đi vào một Motel:

   - Nhận phòng: đưa CMND, nhận chìa khoá phòng.
   - Trả phòng: nhận lại CMND, trả chìa khoá phòng.

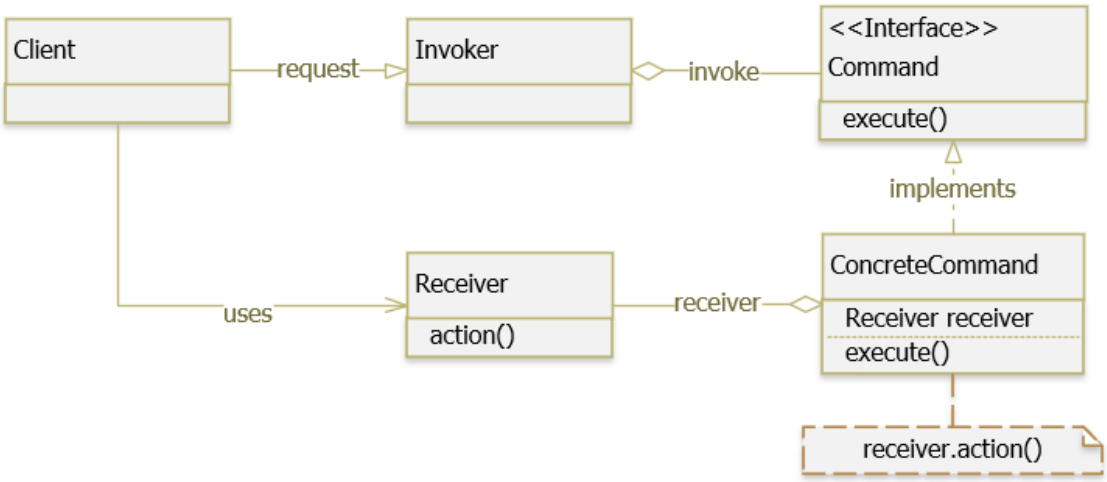
Phần 3: BEHAVIORAL PATTERNS

3.1. Command Pattern

Command Pattern cho phép chuyển yêu cầu thành đối tượng độc lập, có thể được sử dụng để tham số hóa các đối tượng với các yêu cầu khác nhau như log, queue (undo/redo), transtraction. Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Command Pattern cho phép tất cả những Request gửi đến object được lưu trữ trong chính object đó dưới dạng một object Command. Khái niệm Command Object giống như một class trung gian được tạo ra để lưu trữ các câu lệnh và trạng thái của object tại một thời điểm nào đó.

Command Pattern còn được biết đến như là *Action* hoặc *Transaction*.



Các thành phần tham gia trong Command Pattern:

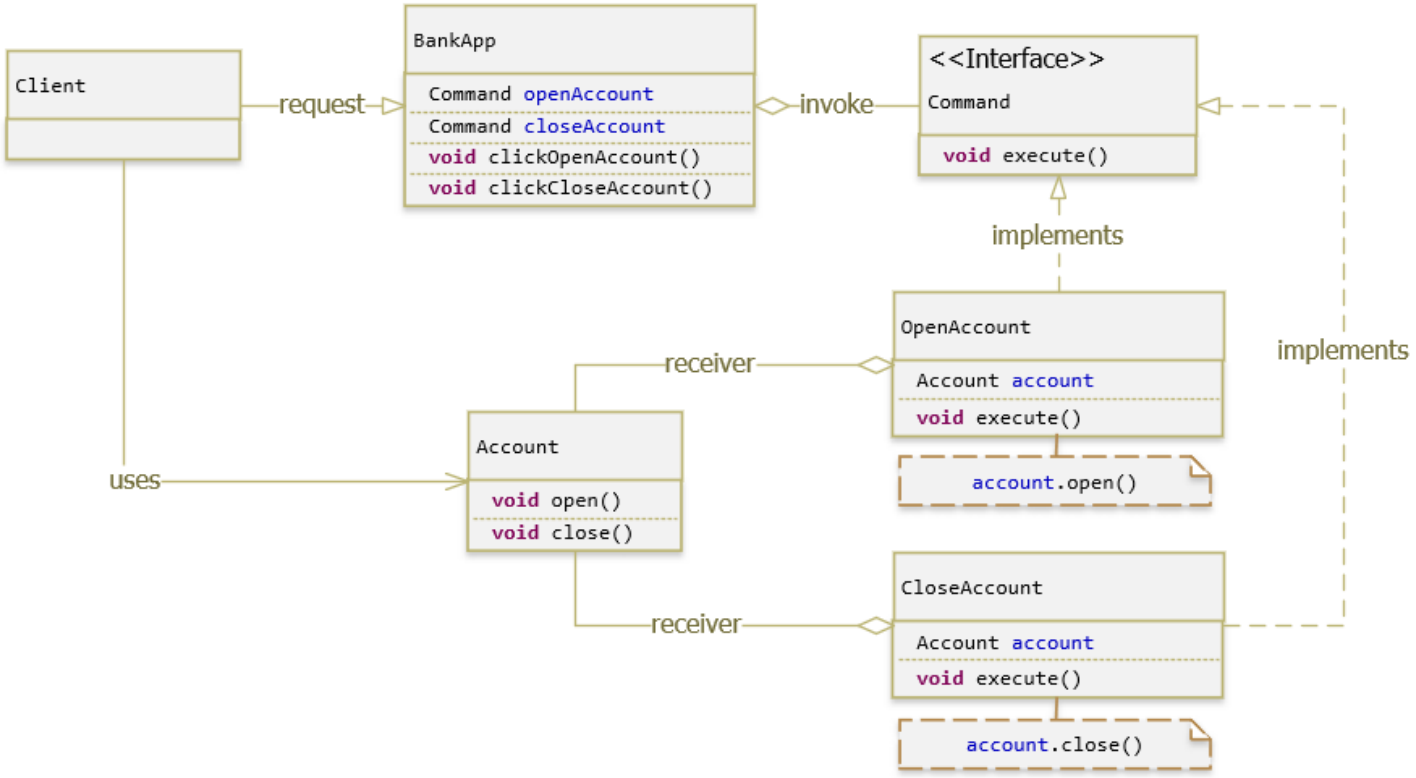
- *Command*: là một interface hoặc abstract class, chứa một phương thức trừu tượng thực thi (execute) một hành động (operation). Request sẽ được đóng gói dưới dạng Command.
- *ConcreteCommand*: là các implementation của Command. Định nghĩa một sự gắn kết giữa một đối tượng Receiver và một hành động. Thực thi execute() bằng việc gọi operation đang hoãn trên Receiver. Mỗi một ConcreteCommand sẽ phục vụ cho một case request riêng.

- *Client*: tiếp nhận request từ phía người dùng, đóng gói request thành ConcreteCommand thích hợp và thiết lập receiver của nó.
- *Invoker*: tiếp nhận ConcreteCommand từ Client và gọi execute() của ConcreteCommand để thực thi request.
- *Receiver*: đây là thành phần thực sự xử lý business logic cho case request. Trong phương execute() của ConcreteCommand chúng ta sẽ gọi method thích hợp trong Receiver.

Client và *Invoker* sẽ thực hiện việc *tiếp nhận* request. Còn việc *thực* thi request sẽ do *Command*, *ConcreteCommand* và *Receiver* đảm nhận.

Bài tập Command Pattern

1. Một hệ thống ngân hàng cung cấp ứng dụng cho khách hàng (client) có thể mở (open) hoặc đóng (close) tài khoản trực tuyến. Hệ thống này được thiết kế theo dạng module, mỗi module sẽ thực hiện một nhiệm vụ riêng, chẳng hạn mở tài khoản (OpenAccount), đóng tài khoản (CloseAccount). Do hệ thống không biết mỗi module sẽ làm gì, nên khi có yêu cầu client (chẳng hạn clickOpenAccount, clickCloseAccount), sẽ đóng gói yêu cầu này và gọi module xử lý.

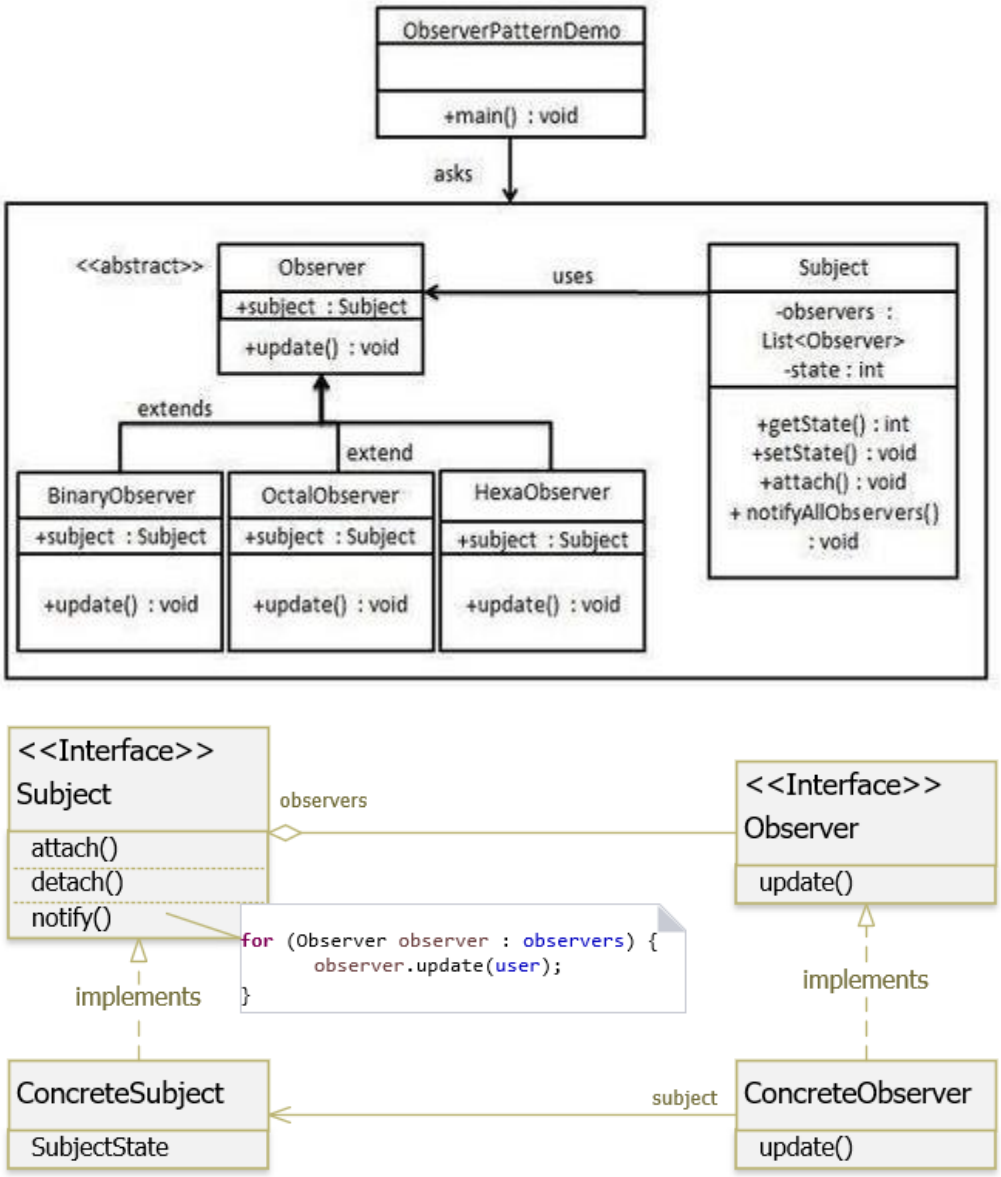


2. Xây dựng một ứng dụng nhỏ thể hiện các thao tác Undo và Redo. Khi nhập một text, Add nó vào một vùng hiển thị nào đó. Người dùng có thể sử dụng thao tác Undo để trở về bước kế trước hay Redo để trở về bước kế sau.
3. Áp dụng Command Pattern vào bài toán order món ăn tại một nhà hàng. Đầu tiên, khách hàng (customer) sẽ order món ăn. Với từng món ăn (steak, pizza, dessert) thì người phục vụ (waiter) sẽ chuyển order cho đầu bếp (chef) chuyên chế biến món đó.

    - Order đóng vai trò Command.
    - Steak và Pizza là các Concrete Command.
    - Steak Chef và Pizza Chef đóng vai trò Receiver.
    - Customer đóng vai trò Client.
    - Waiter là cầu nối giữa Customer và Chef: Invoker.

3.2. Observer Pattern

Observer Pattern định nghĩa mối phụ thuộc một – nhiều giữa các đối tượng để khi mà một đối tượng có sự thay đổi trạng thái, tất các thành phần phụ thuộc của nó sẽ được thông báo và cập nhật một cách tự động. (Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.)



Các thành phần tham gia Observer Pattern:

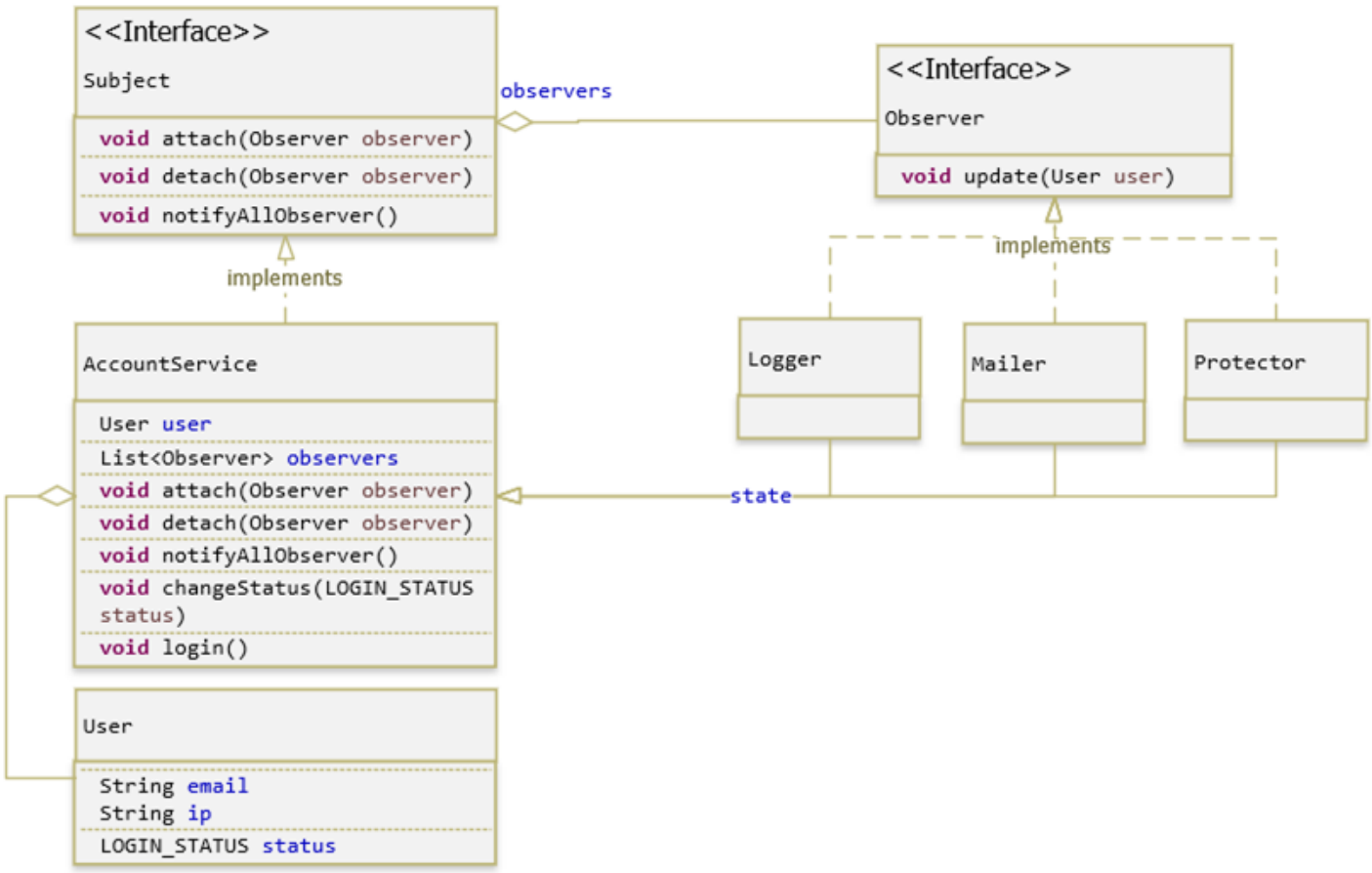
- *Subject*: chứa danh sách các observer, cung cấp phương thức để có thể thêm và loại bỏ observer.
- *Observer*: định nghĩa một phương thức update() cho các đối tượng sẽ được subject thông báo đến khi có sự thay đổi trạng thái.
- *ConcreteSubject*: cài đặt các phương thức của Subject, lưu trữ trạng thái danh sách các ConcreateObserver, gửi thông báo đến các observer của nó khi có sự thay đổi trạng thái.

- *ConcreteObserver*: cài đặt các phương thức của Observer, lưu trữ trạng thái của subject, thực thi việc cập nhật để giữ cho trạng thái đồng nhất với subject gửi thông báo đến.

Sự tương tác giữa subject và các observer như sau: mỗi khi subject có sự thay đổi trạng thái, nó sẽ duyệt qua danh sách các observer của nó và gọi phương thức cập nhật trạng thái ở từng observer, có thể truyền chính nó vào phương thức để các observer có thể lấy ra trạng thái của nó và xử lý.

Bài tập Observer Pattern

1. Observer Pattern với ứng dụng Tracking thao tác một Account
   Giả sử hệ thống cần theo dõi về tài khoản của người dùng. Mọi thao tác của người dùng đều cần được ghi log lại, sẽ thực hiện gửi mail thông báo khi tài khoản hết hạn, thực hiện chặn người dùng nếu truy cập không hợp lệ, …



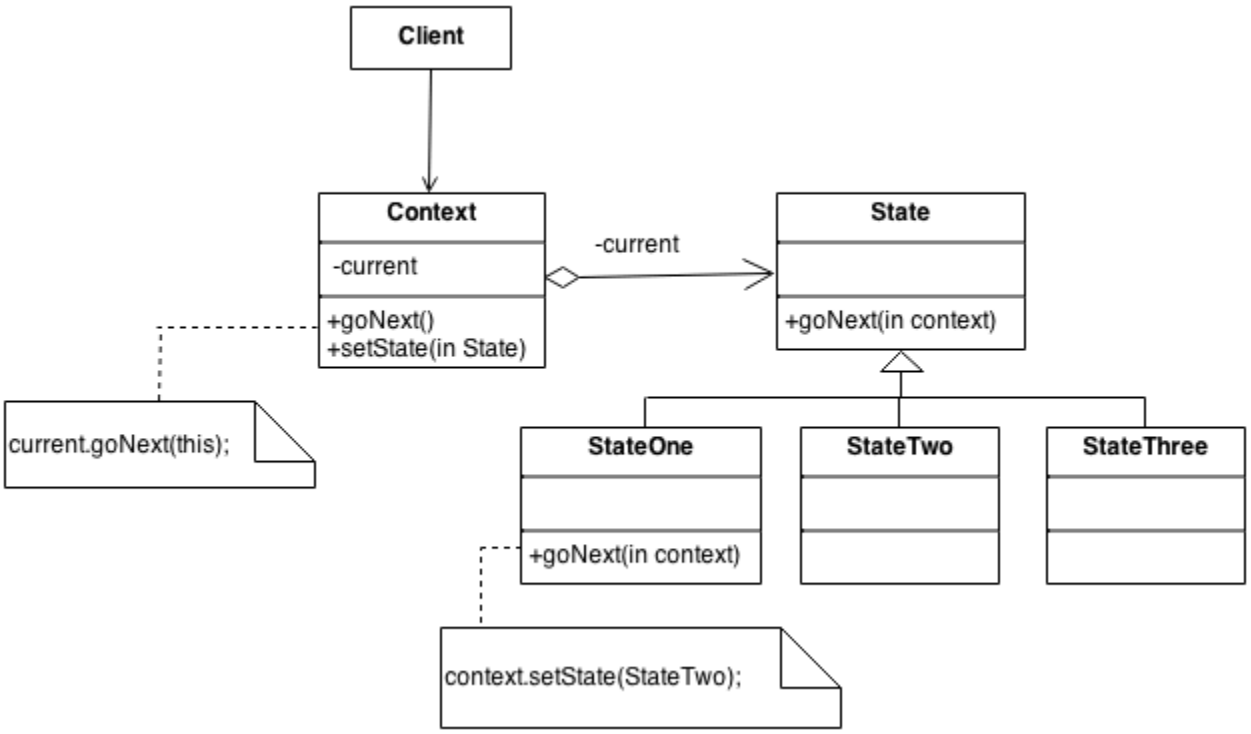
- *Subject*: cung cấp các phương thức để thêm, loại bỏ, thông báo observer.
- *AccountService*: đóng vai trò là ConcreteSubject, sẽ thông báo tới tất cả các observers bất cứ khi nào có thao tác của người dùng liên quan đến đăng nhập, tài khoản hết hạn.
- *Observer*: định nghĩa một phương thức update() cho các đối tượng sẽ được subject thông báo đến khi có sự thay đổi trạng thái. Phương thức này chấp nhận đối số là SubjectState, cho phép các ConcreteObserver sử dụng dữ liệu.

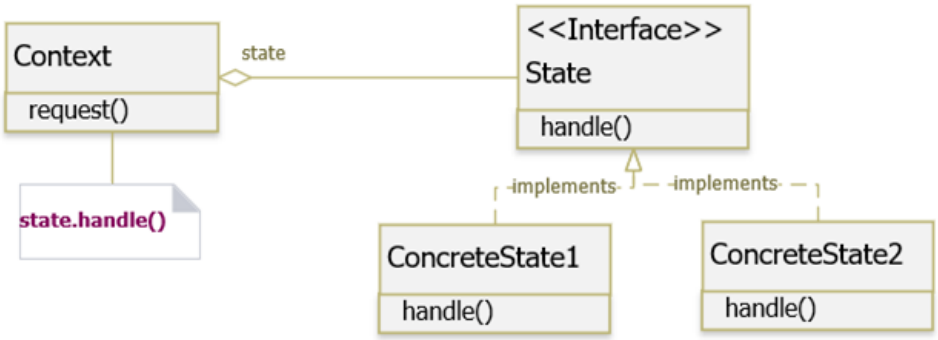
- *Logger*, *Mailer* và *Protector* là các ConcreteObserver. Sau khi nhận được thông báo rằng có thao tác với user và gọi tới phương thức update(), các ConcreteObserver sẽ sử dụng dữ liệu SubjectState để xử lý.

2. Phần mềm diệt virus và được sử dụng ở khắp nơi trong đó có Việt Nam và Mỹ. dù là 2 nước khác nhau nhưng mội khi phần mềm này cập nhật phiên bản mới thì cả 2 nước này đều nhận được thông tin phiên bản mới và cập nhật.

3.3. State Pattern

State Pattern cho phép một đối tượng thay đổi hành vi của khi trạng thái internal thay đổi. Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.



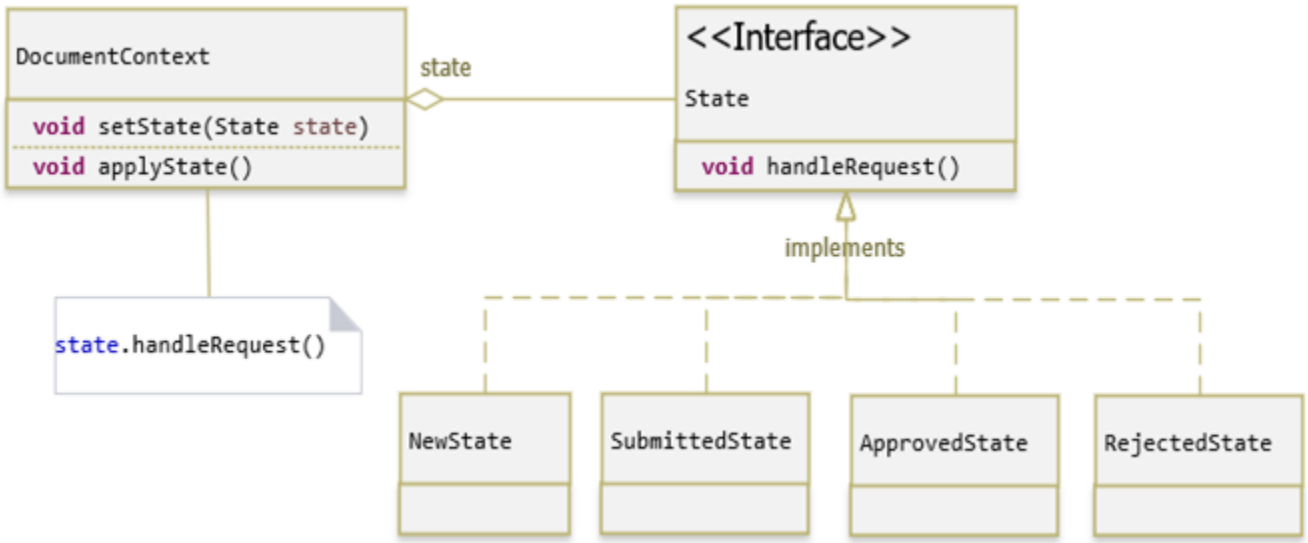


Các thành phần của State Pattern:

- *State*: là một interface hoặc abstract class bao gồm các đặc tính cơ bản của tất cả các đối tượng
- *Context*: được sử dụng bởi Client. Client không truy cập trực tiếp đến State của đối tượng. Lớp Context này chứa thông tin của ConcreteState object, cho hành vi nào tương ứng với trạng thái nào hiện đang được thực hiện.
- *ConcreteState*. Được sử dụng bởi đối tượng Context để truy cập chức năng có thể thay đổi.
- *ConcreteState*: cài đặt các phương thức của State. Mỗi ConcreteState có thể thực hiện logic và hành vi của riêng tùy thuộc vào Context.

Bài tập State Pattern

1. Quản lý thông tin bài báo (Paper) cho tạp chí nghiên cứu khoa học cho trường ĐHCN TPHCM. Khi một bài báo được tác giả gửi tới tạp chí, bài báo sẽ có trạng thái New. Thư ký tạp chí sẽ gửi đi các nhà phản biện cho bài báo, mỗi bài báo sẽ có tối thiểu là 2 người phản biện và tối đa là 3. Trạng thái bài báo lúc này là Reviewing, trường hợp bài báo cần sửa: Revised. Khi đã được các nhà phản biện phản hồi kết quả, thư ký sẽ đánh giá bài báo được chấp nhận (Approved) hay từ chối (Rejected). Áp dụng State Pattern cho chương trình trên.

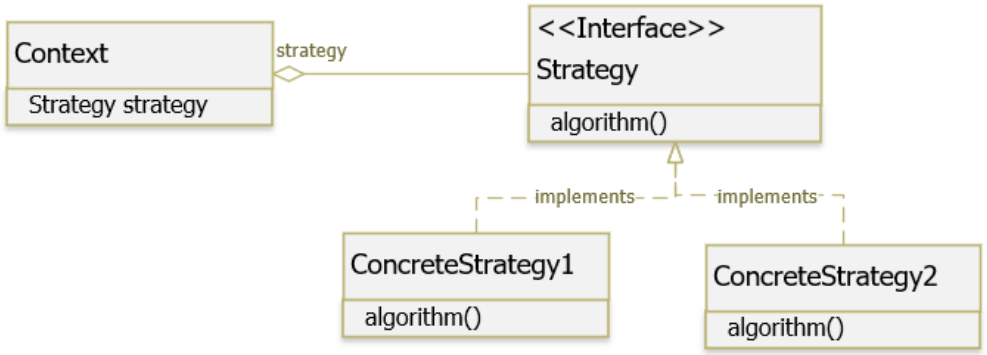


2. Xét trường hợp mẫu State Pattern cho máy bán hàng tự động. Máy bán hàng tự động có các trạng thái dựa trên hàng tồn kho, số lượng tiền gửi, khả năng thay đổi, mặt hàng được chọn, v.v. Khi tiền được đưa vào và có sự chọn lựa, máy bán hàng tự động sẽ hoặc là phân phối sản phẩm hoặc không không cung cấp sản phẩm do không đủ tiền, hoặc không cung cấp sản phẩm đã hết hàng hàng tồn kho.

3.4. Strategy Pattern

Strategy Pattern cho phép định nghĩa tập hợp các thuật toán, đóng gói từng thuật toán lại, và dễ dàng thay đổi linh hoạt các thuật toán bên trong object. Strategy cho phép thuật toán biến đổi độc lập khi người dùng sử dụng chúng.

Strategy Pattern cho phép các giải thuật khác nhau có thể được lựa chọn trong thời-gian-chạy (run-time).

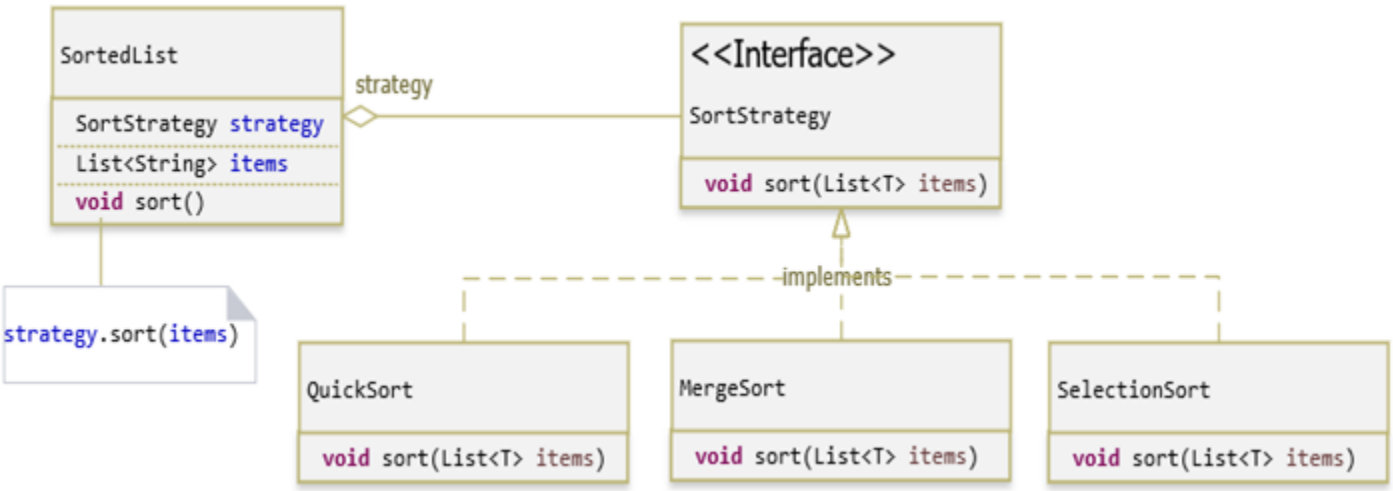


Các thành phần Strategy Pattern:

- *Strategy*: định nghĩa các hành vi có thể có của một Strategy.
- *ConcreteStrategy*: cài đặt các hành vi cụ thể của Strategy.
- *Context*: chứa một tham chiếu đến đối tượng Strategy và nhận các yêu cầu từ Client, các yêu cầu này sau đó được ủy quyền cho Strategy thực hiện.

Bài tập Strategy Pattern

1. Chương trình cung cấp nhiều giải thuật sắp xếp khác nhau: quick sort, merge sort, selection sort, heap sort, tim sort, …. Tùy theo loại dữ liệu, số lượng phần tử, … mà người dùng có thể chọn một giải thuật sắp xếp phù hợp.



2. Tìm kiếm sản phẩm trong một tập sản phẩm, ta có nhiều giải thuật tìm kiếm khác nhau như: Binary Search, Linear Search … Strategy giúp các giải thuật khác nhau độc lập với client sử dụng.

3. Cho một tình huống trong bài toán quản lý bán hàng như sau:
   - Một hóa đơn gồm một hoặc nhiều mặt hàng, có thể thêm mặt hàng vào hóa đơn, xóa mặt hàng ra khỏi hóa đơn, tính tổng tiền của hóa đơn.
   - Thông tin hóa đơn gồm: Mã số hóa đơn và ngày lập hóa đơn.
   - Thông tin mặt hàng gồm: Mã hàng, tên hàng và đơn giá.

   - Khi thực hiện thanh toán tiền hóa đơn, có thể lựa chọn một trong hai hình thức thanh toán sau: thanh toán bằng thẻ tín dụng (CreditCard) hoặc thanh toán trực tuyến (Paypal).
   - Thông tin CreditCard gồm: Số thẻ, tên in trên thẻ, mã bảo mật *(Card Verification Value)* và ngày hết hạn.
   - Thông tin Paypal gồm: Email và mật khẩu *(password)*.

4. Chiến lược nạp thải pin sạc trong bài tập Composite Pattern. Việc nạp / xả năng lượng cho pin cơ bản là phụ thuộc vào tính chất vật lý của pin này. Riêng loại pin tích hợp (Battery) gồm nhiều pin thành phần (Cell) ghép lại thì ta có thể dùng nhiều chiến lược nạp / xả khác nhau.
   - Chiến lược nạp / xả xoay vòng (RoundRobin): Lần lượt nạp / xả pin thành phần trong danh sách rồi lặp lại khi cần.
   - Chiến lược nạp / xả min – max: Chọn pin thành phần có năng lượng hiện hành nhỏ nhất để nạp năng lượng cho nó, chọn pin thành phần có năng lượng hiện hành lớn nhất để xả năng lượng của nó.