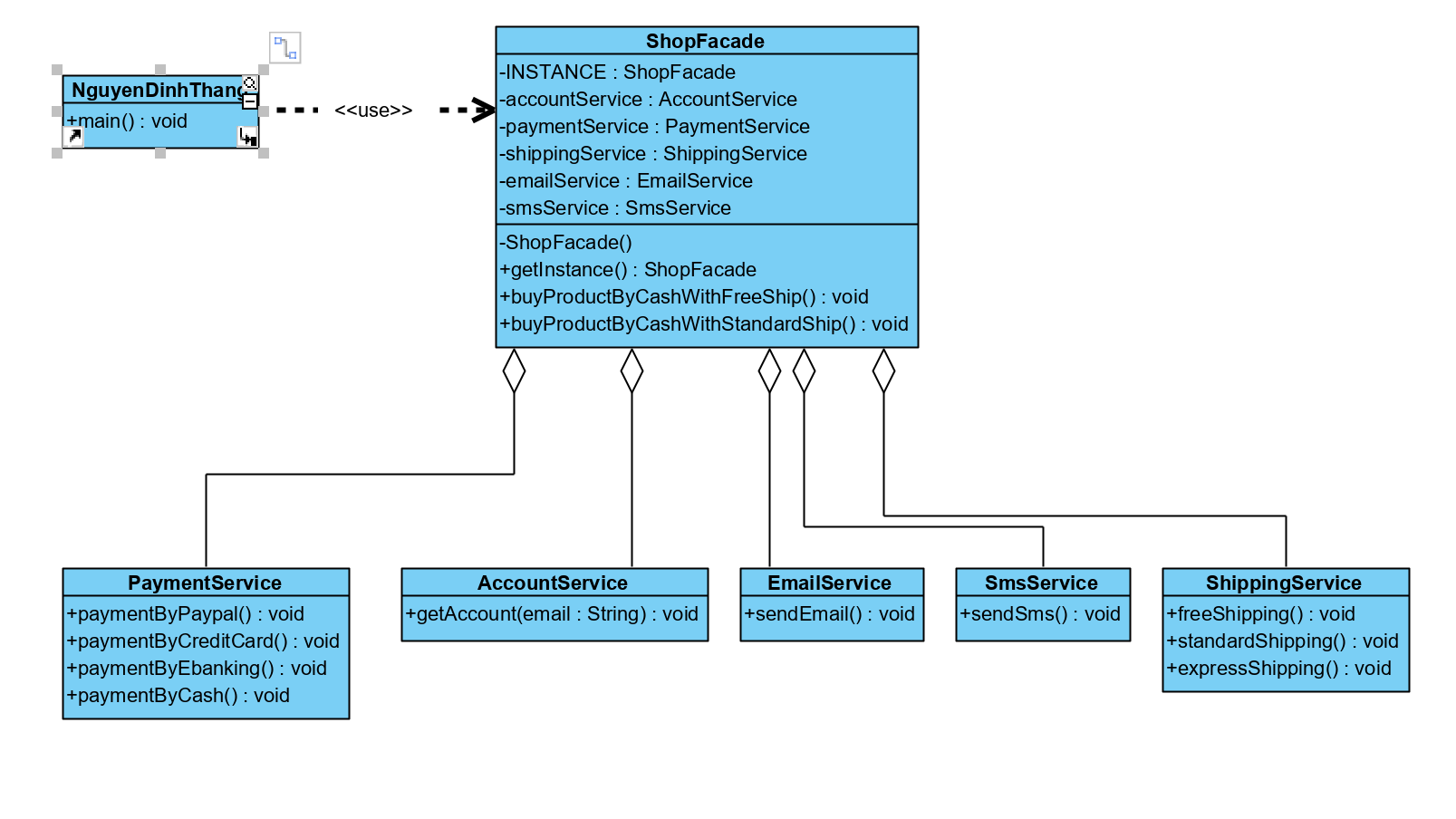
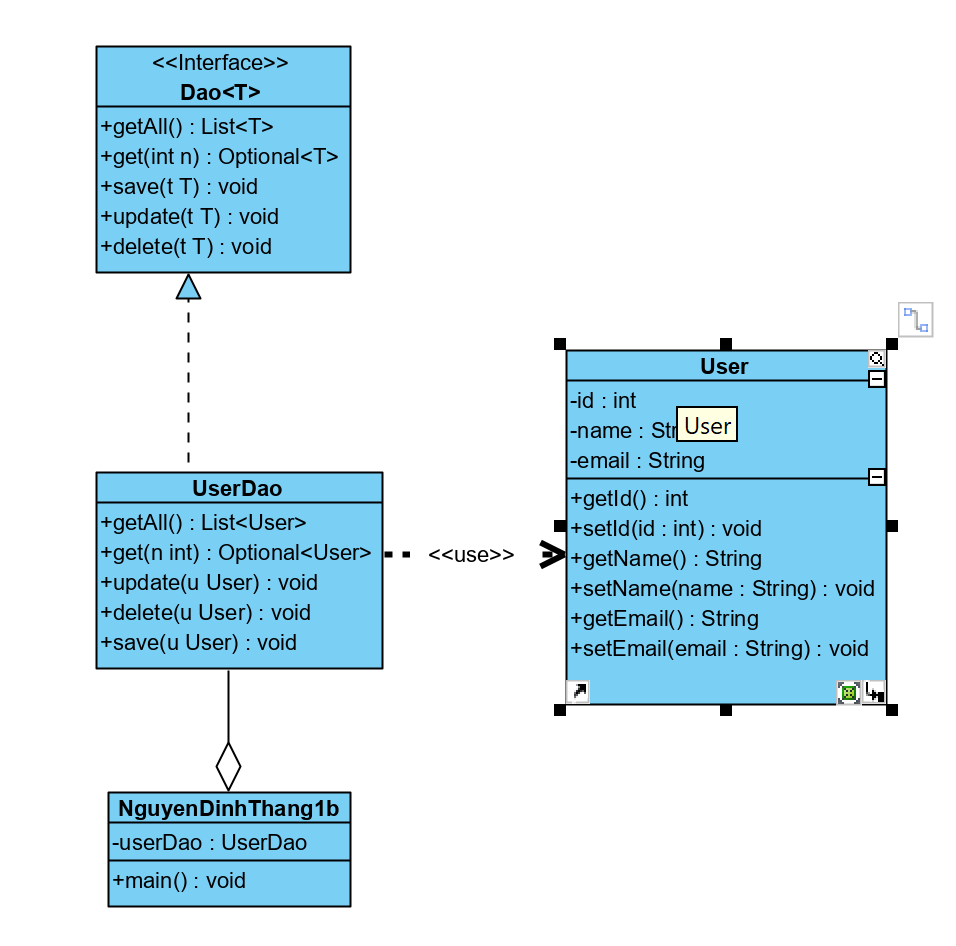
* 1. **Using facade for designing a program for processing various services**

Thiết kế: 

* Code:

|  |  |
| --- | --- |
| **Tên class** | **Code** |
| Class AccountService |  |
| Class EmailService |  |
| Class PaymentService |  |
| Class ShippingService |  |
| Class SmsService |  |
| Class ShopFacade |  |
| Class NguyenDinhThang |  |
| Kết quả chạy code |  |

* **Kinh nghiệm**:
* Lợi ích của Façade Pattern:  
  ▪ Giúp cho hệ thống của bạn trở nên đơn giản hơn trong việc sử dụng và  
  trong việc hiểu nó, vì một mẫu Facade có các phương thức tiện lợi cho các  
  tác vụ chung.  
  ▪ Giảm sự phụ thuộc của các mã code bên ngoài với hiện thực bên trong của  
  thư viện, vì hầu hết các code đều dùng Facade, vì thế cho phép sự linh động  
  trong phát triển các hệ thống.  
  ▪ Đóng gói tập nhiều hàm API được thiết kế không tốt bằng một hàm API  
  đơn có thiết kế tốt hơn.  
  - Sử dụng khi nào:  
  ▪ 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. Khi đó có thể sử dụng Facade  
  Pattern để tạo ra một giao diện đơn giản cho người sử dụng một hệthống phức tạp.  
  ▪ Khi người sử dụng phụ thuộc nhiều vào các lớp cài đặt. Sử dụng Façade  
  Pattern tăng khả năng độc lập và khả chuyển của hệ thống con, dễ  
  chuyển đổi nâng cấp trong tương lai.  
  ▪ Khi bạn muốn phân lớp các hệ thống con. Sử dụng Façade Pattern sẽ giảmsự phụ thuộc của các hệ thống con  
  ▪ Khi bạn muốn bao bọc, che giấu tính phức tạp trong các hệ thống con đối  
  với phía Client
  1. **DAO pattern**
* Thiết kế:



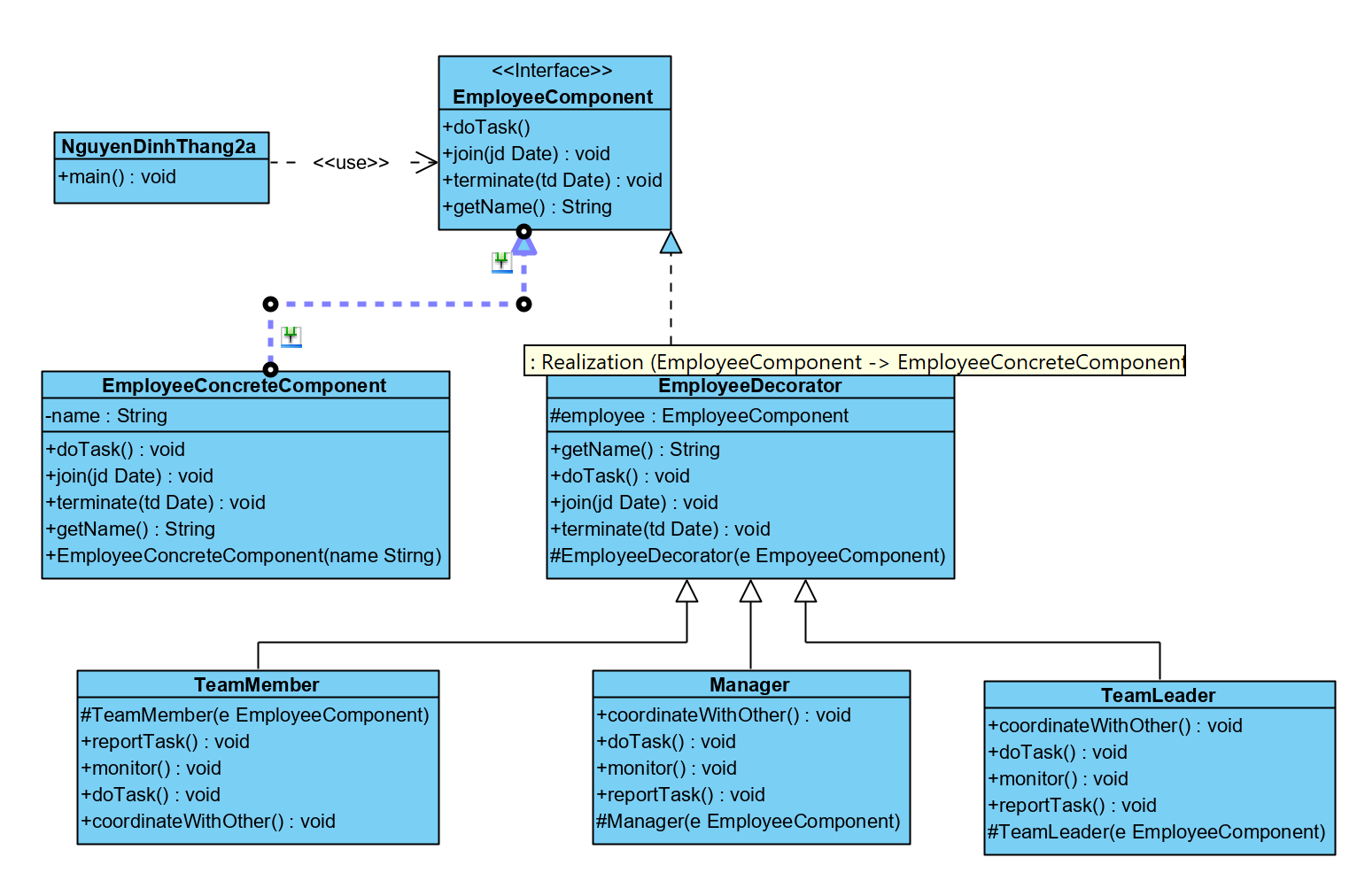
* Code:

|  |  |
| --- | --- |
| **Tên class** | **Code** |
| Interface Dao |  |
| Class User |  |
| Class UserDao |  |
| Class NguyenDinhThang |  |
| Kết quả chạy code |  |

* **Kinh nghiệm**:

Lợi ích:  
▪ Giảm sự kết nối (loose coupling) giữa logic nghiệp vụ (Business) và logic lưu  
trữ (Persistence).  
▪ Mẫu DAO cho phép đóng gói code để thực hiện thao tác CRUD, ngăn chặn  
việc implement riêng lẻ trong từng phần khác nhau của ứng dụng.  
▪ Dễ mở rộng, bảo trì: tất cả các chi tiết lưu trữ được ẩn khỏi phần còn lại của  
ứng dụng. Do đó, những thay đổi có thể được thực hiện bằng cách chỉ sửa đổi  
một implement của DAO trong khi phần còn lại của ứng dụng không bị ảnh  
hưởng. DAO hoạt động như một trung gian giữa ứng dụng và cơ sở dữ liệu.  
▪ Dễ hiểu: mọi người đều theo một quy chuẩn đã được định sẵn, nên dễ hiểu  
hơn, tiết kiệm được nhiều thời gian hơn.  
▪ Trong một dự án lớn hơn, các nhóm khác nhau làm việc trên các phần khác  
nhau của ứng dụng, mẫu DAO cho phép phân tách rõ ràng các thành phần này.  
- Sử dụng khi nào:  
▪ Khi muốn thay đổi nguồn dữ liệu sau này, như chuyển từ cơ dữ liệu MySQL  
sang Oracle, SQL Server, …  
▪ Khi muốn phân tách rõ ràng các thành phần của ứng dụng.

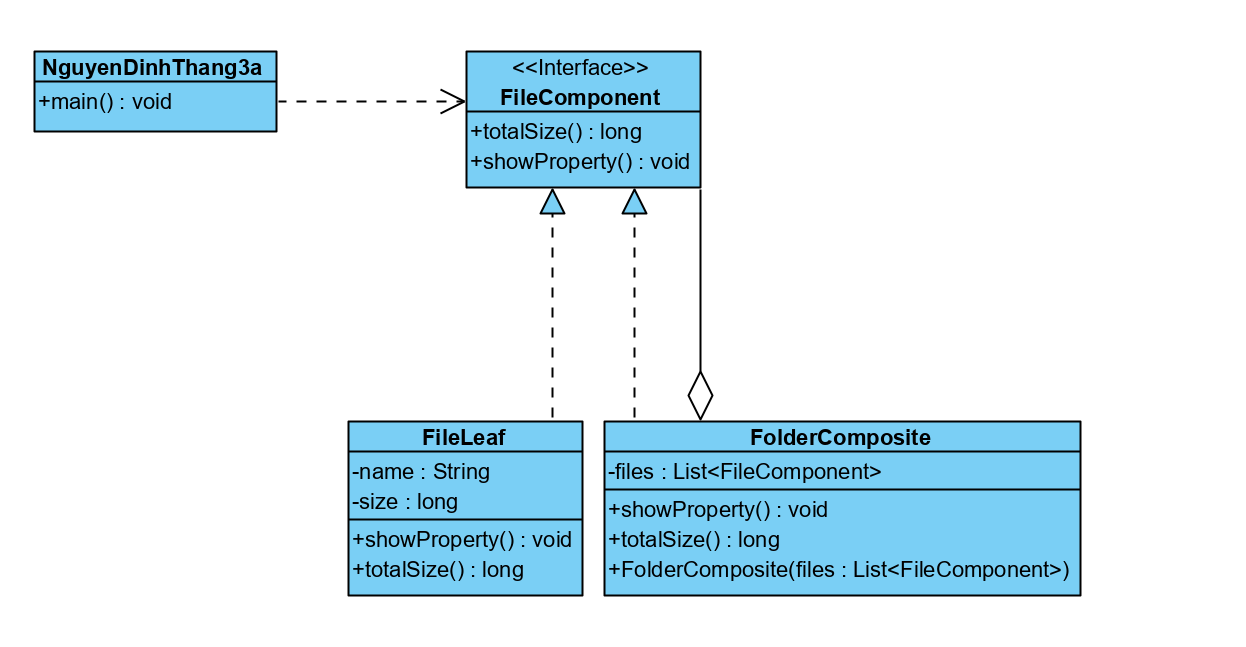
* 1. **Using decorator for designing Employee class with various functionalities**
* Thiết kế:



* Code:

|  |  |
| --- | --- |
| **Tên class** | **Code** |
| Interface EmployeeComponent |  |
| Class EmployeeConcreteComponent |  |
| Class EmployeeDecorator |  |
| Class Manager |  |
| Class TeamLeader |  |
| Class TeamMember |  |
| Class NguyenDinhThang |  |
| Kết quả chạy code |  |

* **Kinh nghiệm**:
* Lợi ích:  
  ▪ Tăng cường khả năng mở rộng của đối tượng, bởi vì những thay đổi được  
  thực hiện bằng cách implement trên các lớp mới.  
  ▪ Client sẽ không nhận thấy sự khác biệt khi bạn đưa cho nó một wrapper thay  
  vì đối tượng gốc.  
  ▪ Một đối tượng có thể được bao bọc bởi nhiều wrapper cùng một lúc.  
  15  
  ▪ Cho phép thêm hoặc xóa tính năng của một đối tượng lúc thực thi (run-time).  
  - Sử dụng khi nào:  
  ▪ Khi muốn thêm tính năng mới cho các đối tượng mà không ảnh hưởng đến  
  các đối tượng này.  
  ▪ Khi không thể mở rộng một đối tượng bằng cách thừa kế (inheritance).  
  Chẳng hạn, một class sử dụng từ khóa final, muốn mở rộng class này chỉ còn  
  cách duy nhất là sử dụng decorator.  
  ▪ Trong một số nhiều trường hợp mà việc sử dụng kế thừa sẽ mất nhiều công  
  sức trong việc viết code. Ví dụ trên là một trong những trường hợp như vậy.
  1. **Composite pattern**
* Thiết kế:



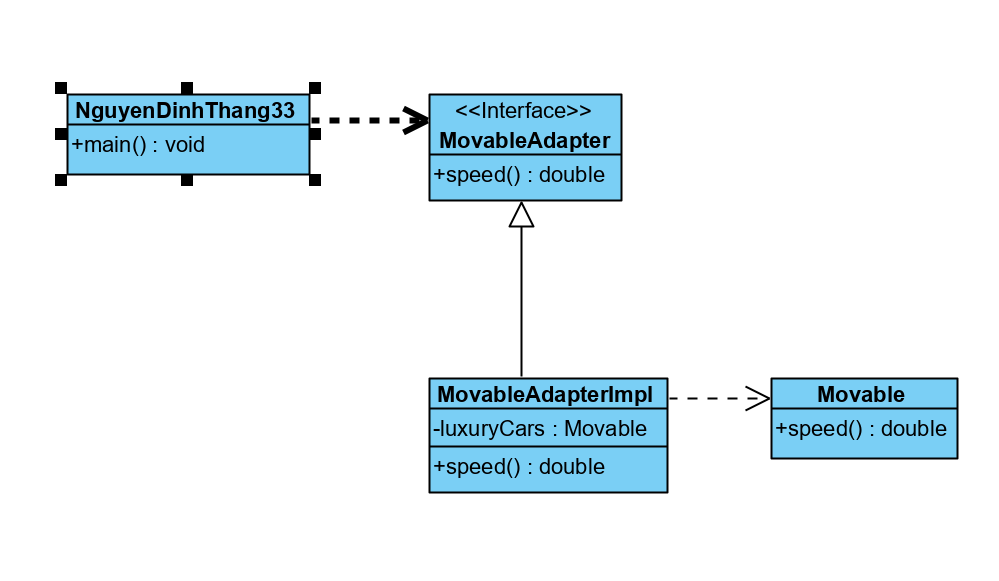
* Code:

|  |  |
| --- | --- |
| **Tên class** | **Code** |
| Class FileComponent |  |
| Class FileLeaf |  |
| Class FolderComposite |  |
| Class NguyenDinhThang |  |
| Kết quả chạy code |  |

* **Kinh nghiệm**:

Lợi ích:  
▪Cung cấp cùng một cách sử dụng đối với từng đối tượng riêng lẻ hoặc nhóm  
các đối tượng với nhau.  
- Sử dụng khi nào:  
▪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

* 1. **Adapter**
* Thiết kế:



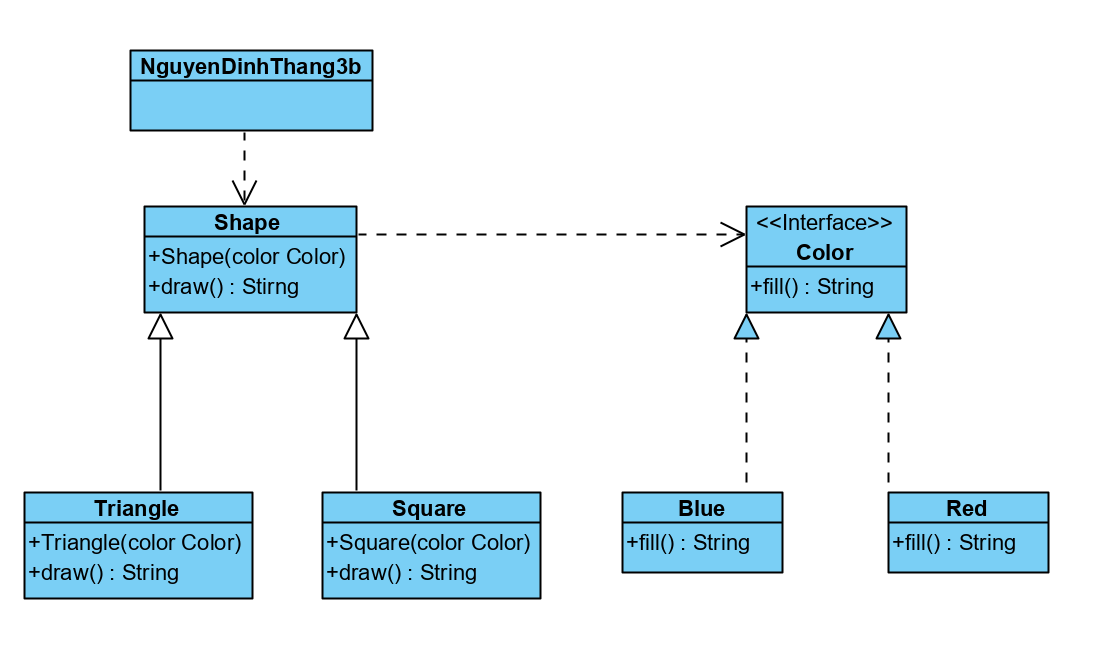
* Code:

|  |  |
| --- | --- |
| **Tên class** | **Code** |
| Class BugattiVeyron |  |
| Class Movable |  |
| Class MovableAdapter |  |
| Class MovableAdapterImpl |  |
| Class NguyenDinhThang |  |
| Kết quả chạy code |  |

* **Kinh nghiệm**:

Lợi ích:  
▪ Giảm sự phục thuộc giữa abstraction và implementation (loose coupling)  
▪ Giảm số lượng những lớp con không cần thiết  
▪ Code sẽ gọn gàn hơn và kích thước ứng dụng sẽ nhỏ hơn  
▪ Dễ bảo trì hơn  
▪ Dễ dàng mở rộng về sau  
▪ Cho phép ẩn các chi tiết implement từ client  
- Sử dụng:  
▪ Khi bạn muốn tách ràng buộc giữa Abstraction và Implementation, để có thể  
dễ dàng mở rộng độc lập nhau.  
▪ Cả Abstraction và Implementation của chúng nên được mở rộng bằng  
subsclass.  
▪ Sử dụng ở những nơi mà những thay đổi được thực hiện trong implement  
không ảnh hưởng đến phía client.

* 1. **Bridge**
* Thiết kế:



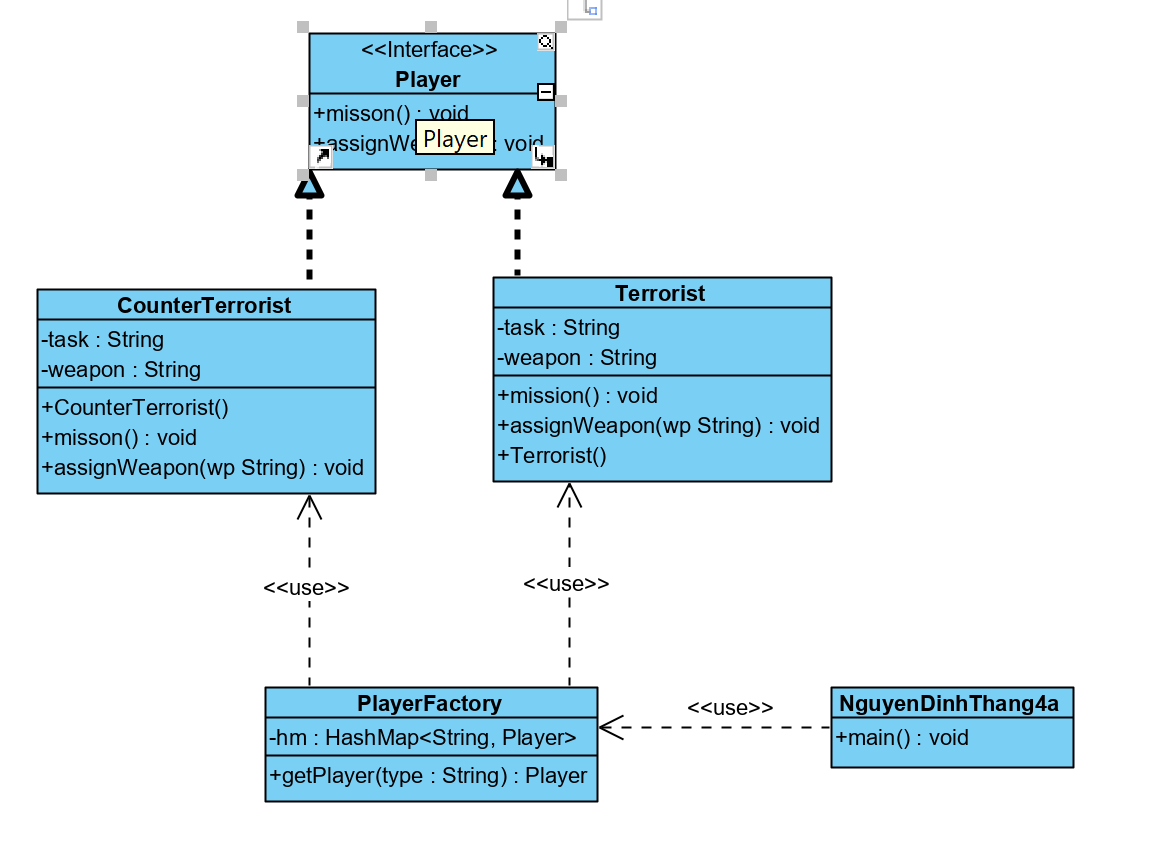
* Code:

|  |  |
| --- | --- |
| **Tên class** | **Code** |
| Class Blue |  |
| Class Color |  |
| Class Red |  |
| Class Shape |  |
| Class Square |  |
| Class NguyenDinhThang |  |
| Kết quả chạy code |  |

* **Kinh nghiệm**:

Lợi ích:  
▪ Cho phép nhiều đối tượng có interface giao tiếp khác nhau có thể tương tác  
và giao tiếp với nhau.  
▪ Tăng khả năng sử dụng lại thư viện với interface không thay đổi do không có  
mã nguồn.  
- Sử dụng:  
▪ 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

* 1. **Flyweight**
* Thiết kế:



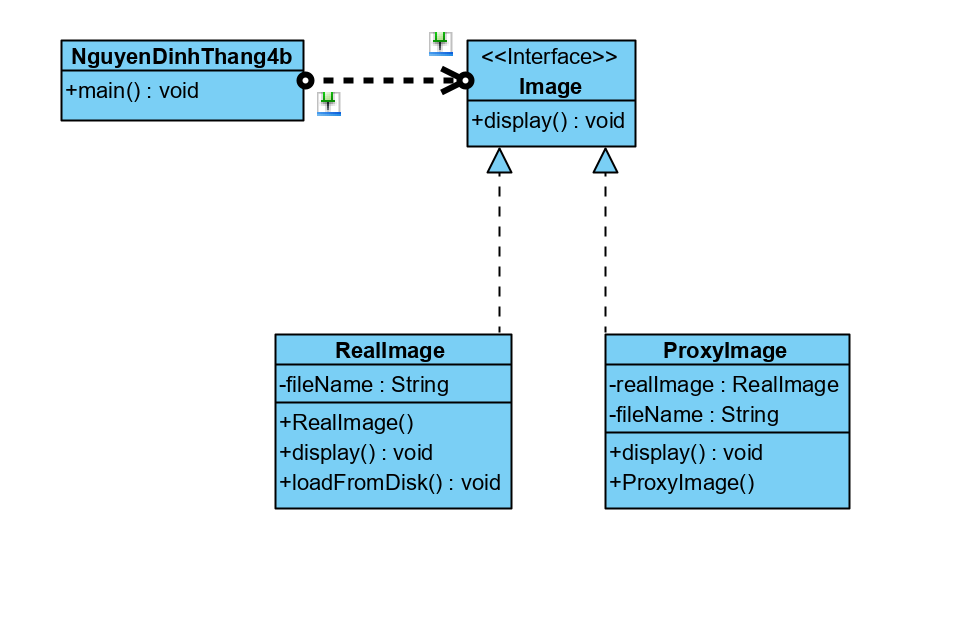
* Code:

|  |  |
| --- | --- |
| **Tên class** | **Code** |
| Class CounterTerrorist |  |
| Interface Player |  |
| Class PlayerFactory |  |
| Class Terrorist |  |
| Class NguyenDinhThang |  |
| Kết quả chạy code |  |

* **Kinh nghiệm**:

Lợi ích:  
▪Giảm số lượng đối tượng được tạo ra bằng cách chia sẻ đối tượng. Vì vậy, tiết  
kiệm bộ nhớ và các thiết bị lưu trữ cần thiết.  
▪Cãi thiện khả năng cache dữ liệu vì thời gian đáp ứng nhanh.  
▪Tăng performance.  
- Sử dụng:  
▪Khi có một số lớn các đối tượng được ứng dụng tạo ra một cách lặp đi lặp lại.  
▪Khi việc tạo ra đối tượng đòi hỏi nhiều bộ nhớ và thời gian.  
▪Khi muốn tái sử dụng đối tượng đã tồn tại thay vì phải tốn thời gian để tạo  
mới.  
▪Khi nhóm đối tượng chứa nhiều đối tượng tương tự và hai đối tượng trong  
nhóm không khác nhau nhiều

* 1. **Proxy**
* Thiết kế:



* Code:

|  |  |
| --- | --- |
| **Tên class** | **Code** |
| Class Image |  |
| Class ProxyImage |  |
| Class RealImage |  |
| Class NguyenDinhThang |  |
| Kết quả chạy code |  |

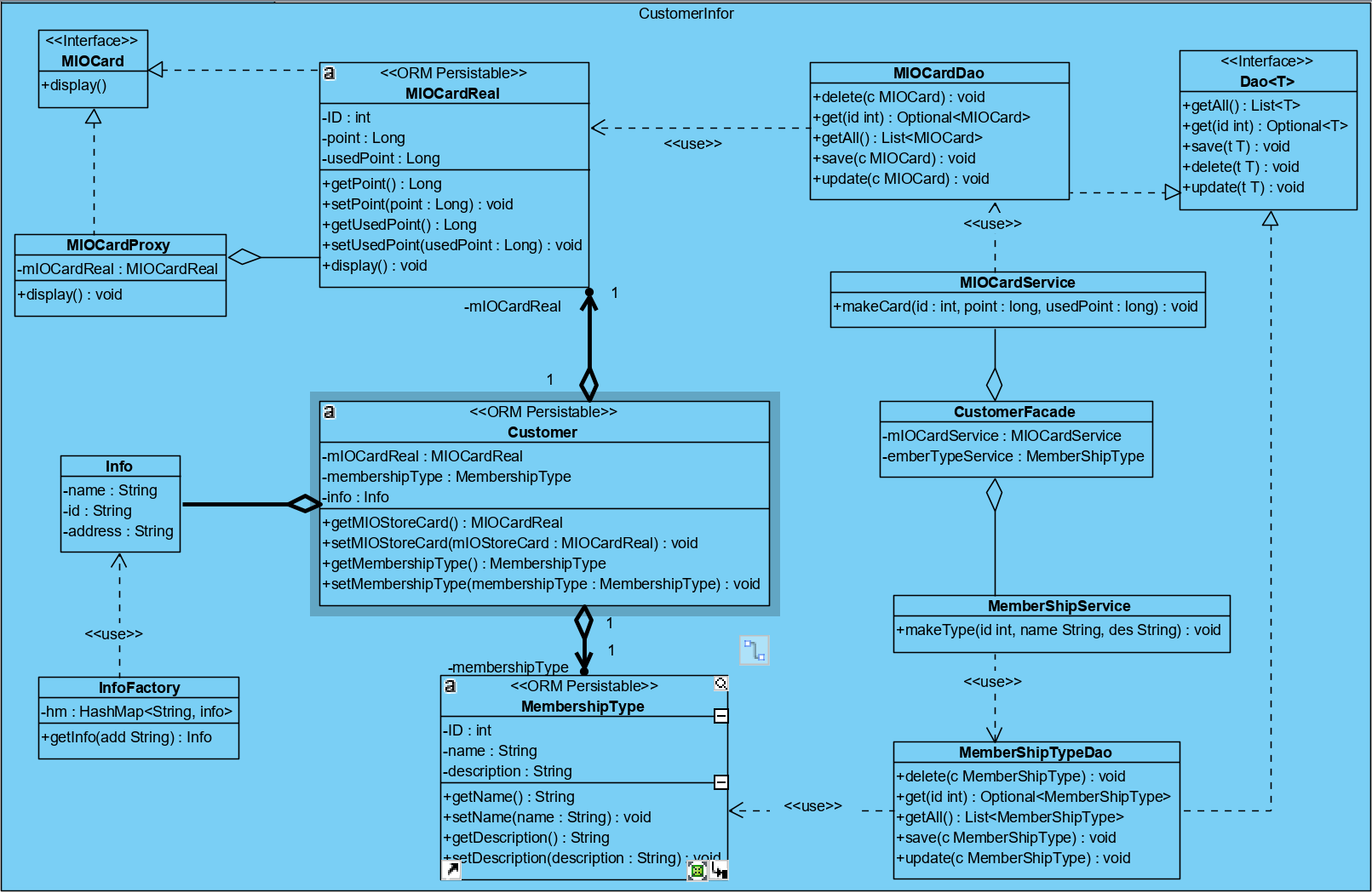
* **Kinh nghiệm**:

Lợi ích:  
▪ Cãi thiện Performance thông qua lazy loading, chỉ tải các tài nguyên khi  
chúng được yêu cầu.  
▪ Nó cung cấp sự bảo vệ cho đối tượng thực từ thế giới bên ngoài.  
▪ Giảm chi phí khi có nhiều truy cập vào đối tượng có chi phí khởi tạo ban đầu  
lớn.  
▪ Dễ nâng cấp, bảo trì.  
- Sử dụng:  
▪ Khi muốn bảo vệ quyền truy xuất vào các phương thức của object thực.  
▪ Khi cần một số thao tác bổ sung trước khi thực hiện phương thức của  
object thực.  
▪ Khi tạo đối tượng ban đầu là theo yêu cầu hoặc hệ thống yêu cầu sự chậm trễ  
khi tải một số tài nguyên nhất định (lazy loading).  
▪ Khi có nhiều truy cập vào đối tượng có chi phí khởi tạo ban đầu lớn.  
▪ Khi đối tượng gốc tồn tại trong môi trường từ xa (remote).  
▪ Khi đối tượng gốc nằm trong một hệ thống cũ hoặc thư viện của bên thứ ba.  
▪ Khi muốn theo dõi trạng thái và vòng đời đối tượng.

1. **Áp dụng pattern vào ứng dụng**

**Áp dụng 4 pattern là façade, dao, flyweight và proxy**

**Thiết kế:**

****

**Code**

|  |  |
| --- | --- |
| **Tên class** | **Code** |
| Class Dao |  |
| Class MIOCardDao |  |
| Class MemberShipDao |  |
| Class MemberShipType |  |
| Interface MIOCard |  |
| Class MIOCardReal |  |
| Class MIOCardProxy |  |
| Class MemberShipService |  |
| Class MIOCardService |  |
| Class Info |  |
| Class InfoFactory |  |
| Class CustomerFacade |  |
| Class Cútomer |  |
| Class NguyenDinhThang |  |
| Kết quả chạy code |  |