

Lời nói đầu

Sau khi học lập trình Hướng đối tượng (OOP – Object-Oriented Programming) bằng một ngôn ngữ Hướng đối tượng nào đó, các bạn sẽ tiếp tục học phân tích thiết kế Hướng đối tượng (OOAD – Object-Oriented Analysis and Design) để tiếp cận với việc xây dựng ứng dụng Hướng đối tượng một cách có phương pháp. Các mẫu thiết kế (Design Patterns) được xem như một kết quả của phần này. Các mẫu thiết kế là kinh nghiệm rút ra từ thực tiễn lập trình, vì vậy mang đến ý nghĩa thực hành rất lớn khi vận dụng lập trình Hướng đối tượng trong thực tế.

Trong tài liệu này, tôi giới thiệu đến các bạn 23 mẫu thiết kế được định nghĩa trong cuốn sách kinh điển "Design Patterns - Elements of Reusable Object-Oriented Software" [Addison-Wesley, 1995] của Erich Gamma, Richard Helm, Ralph Johnson và John Vlissides (GoF – Gang of Four). Giá trị thực tiễn của các mẫu thiết kế đến ngày nay vẫn phát huy mặc dù đã hơn 20 năm kể từ khi cuốn sách được xuất bản.

Tôi cố gắng trình bày một cách đơn giản, trong sáng, giúp bạn đọc nắm bắt được ý tưởng, tinh thần của các mẫu thiết kế. Do việc áp dụng các mẫu thiết kế linh hoạt, tinh tế, nhiều khái niệm dễ mờ hồ, nhầm lẫn. Tôi đã cân nhắc tuyển chọn thật kỹ các ví dụ, các biến thể cài đặt, khoảng 70 bài tập, các case study, ... nhằm làm nổi bật các trường hợp sử dụng và đặc điểm cài đặt của các mẫu thiết kế.

Tôi xin gửi lời tri ân đến các bà đã nuôi dạy tôi, các thầy cô đã tận tâm chỉ dạy tôi. Chỉ có công hiến hết mình cho tri thức tôi mới thấy mình đền đáp được công ơn đó.

Tôi đặc biệt gửi lời cảm ơn chân thành đến anh Huỳnh Văn Đức và anh Lê Gia Minh; tôi may mắn được làm việc chung và học tập từ các anh rất nhiều khi các anh giảng dạy môn Lập trình Hướng đối tượng tại Đại Học Kỹ thuật Công nghệ Thành phố Hồ Chí Minh.

Tôi xin cảm ơn gia đình đã hy sinh rất nhiều để tôi có được khoảng thời gian cần thiết thực hiện được giáo trình này. Tôi cũng gửi lời cảm ơn đến các bạn sinh viên, đã đọc, góp ý và giúp tôi hiệu chỉnh nhiều phần trong giáo trình.

Những trang đầu tiên của tài liệu được bắt đầu viết từ năm 2005, nhưng do phải giải quyết vô số công việc, mãi đến hè năm 2013 tôi mới có thời gian viết xong tài liệu. Mặc dù đã dành rất nhiều thời gian và công sức, viết và vẽ hình minh họa, phải hiệu chỉnh chi tiết và cập nhật nhiều lần, nhưng tài liệu không thể nào tránh được những sai sót và hạn chế. Tôi thực sự mong nhận được các ý kiến đóng góp từ bạn đọc để tài liệu có thể hoàn thiện hơn. Nội dung tài liệu sẽ được cập nhật định kỳ, mở rộng và viết chi tiết hơn; tùy thuộc những phản hồi, những đề nghị, những ý kiến đóng góp từ bạn đọc.

Các bạn đồng nghiệp nếu có sử dụng tài liệu này trong giảng dạy, xin gửi cho tôi ý kiến đóng góp phản hồi, hầu giúp tài liệu được hoàn thiện thêm, phục vụ cho công tác giảng dạy chung.

GoF Design Patterns

Design patterns là tập hợp các mẫu thiết kế, dùng như giải pháp thực tế hoặc dùng như thiết kế chuẩn cho các vấn đề phổ biến khi xây dựng phần mềm hướng đối tượng. Ngoài việc giải quyết các vấn đề phức tạp, các mẫu thiết kế còn được áp dụng trong toàn bộ vòng đời phát triển phần mềm, giúp ứng dụng có được tổ chức tốt, linh hoạt, dễ bảo trì và dễ mở rộng, nâng cấp.

Tài liệu này giới thiệu 23 mẫu thiết kế được định nghĩa trong cuốn sách kinh điển [1] "Design Patterns - Elements of Reusable Object-Oriented Software" [Addison-Wesley, 1995] của nhóm 4 tác giả: Erich Gamma, Richard Helm, Ralph Johnson, và John Vlissides (GoF – Gang of Four)

Các mẫu thiết kế của GoF được xem là nền tảng để hiểu rất nhiều mẫu thiết kế khác trong nhiều lĩnh vực phần mềm. Chúng cũng được xem là kiến thức tối thiểu của các lập trình viên sử dụng ngôn ngữ OOP. Sau gần 25 năm, giá trị của các mẫu thiết kế này vẫn còn quan trọng.

GoF phân loại 23 mẫu thiết kế này theo hai cách:

- Dựa trên mục đích (purpose) của mẫu thiết kế: chia làm ba nhóm.

Creational gồm 5 mẫu thiết kế: Abstract Factory, Builder, Factory Method, Prototype và Singleton. Nhóm này mô tả việc trừu tượng hóa quá trình tạo ra các thể hiện (instance) của đối tượng, ẩn giấu logic tạo đối tượng thay vì khởi tạo trực tiếp và công khai thông qua constructor. Lưu ý, quá trình tạo đối tượng (instantiation) thường dẫn đến các vấn đề kết nối chặt, tính đa hình cũng không làm việc khi chúng ta tạo đối tượng.

Structural gồm 7 mẫu thiết kế: Adapter, Bridge, Composite, Decorator, Facade, Flyweight và Proxy. Nhóm này mô tả cách phối hợp các lớp và các đối tượng để hình thành một cấu trúc phức tạp hơn. Nhóm này được sử dụng khi thiết kế hệ thống mới hoặc khi bảo trì, mở rộng hệ thống có sẵn.

Behavioral gồm 11 mẫu thiết kế: Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method và Visitor. Nhóm này quan tâm đến việc tăng tính linh động khi trao đổi thông tin, phân chia trách nhiệm và tương tác giữa các đối tượng với nhau. Nhóm này được áp dụng khi ứng dụng có luồng điều khiển phức tạp khó theo dõi trong thời gian chạy. Các mẫu thiết kế sẽ chuyển trọng tâm ra khỏi luồng điều khiển để cho phép bạn tập trung vào cách các đối tượng được kết nối với nhau.

- Dựa trên phạm vi (scope) quan hệ: chia làm hai nhóm.

Class dựa vào thừa kế. Bao gồm: Factory Method, Adapter (class), Interpreter và Template Method.

Object dựa vào tổng hợp. Bao gồm: Abstract Factory, Builder, Prototype, Singleton, Adapter (object), Bridge, Composite, Decorator, Facade, Flyweight, Proxy, Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy và Visitor.

Steven John Metsker [8] lại phân loại các mẫu thiết kế thành 5 nhóm:

Interface gồm Adapter, Facade, Composite và Bridge. Giải quyết các vấn đề liên quan đến giao diện.

Responsibility gồm Singleton, Observer, Mediator, Proxy, Chain of Responsibility và Flyweight. Giải quyết các vấn đề liên quan đến phân công trách nhiệm cho đối tượng.

Construction gồm Builder, Factory Method, Abstract Factory, Prototype và Memento. Giải quyết các vấn đề về tạo đối tượng mà không dùng constructor.

Operation gồm Template Method, State, Strategy, Command và Interpreter. Giải quyết các vấn đề điều khiển tác vụ, xử lý khi có nhiều tác vụ tham gia.

Extension gồm Decorator, Iterator và Visitor. Giải quyết vấn đề mở rộng chức năng.

Trong cuộc phỏng vấn năm 2009 (<http://www.informit.com/articles/article.aspx?p=1404056>) nhóm GoF thảo luận ý định phân loại lại các mẫu thiết kế nhằm nhấn mạnh các mẫu quan trọng và tách chúng ra khỏi các mẫu ít được sử dụng hơn. GoF cũng có ý định đưa thêm vào một số mẫu thiết kế mới, được trình bày trong phần "Các mẫu thiết kế mở rộng".

Core Composite, Strategy, State, Command, Iterator, Proxy, Template Method, Facade

Creational Factory (nâng cấp từ Factory Method), Prototype, Builder, Dependency Injection (thêm), đề nghị loại Singleton

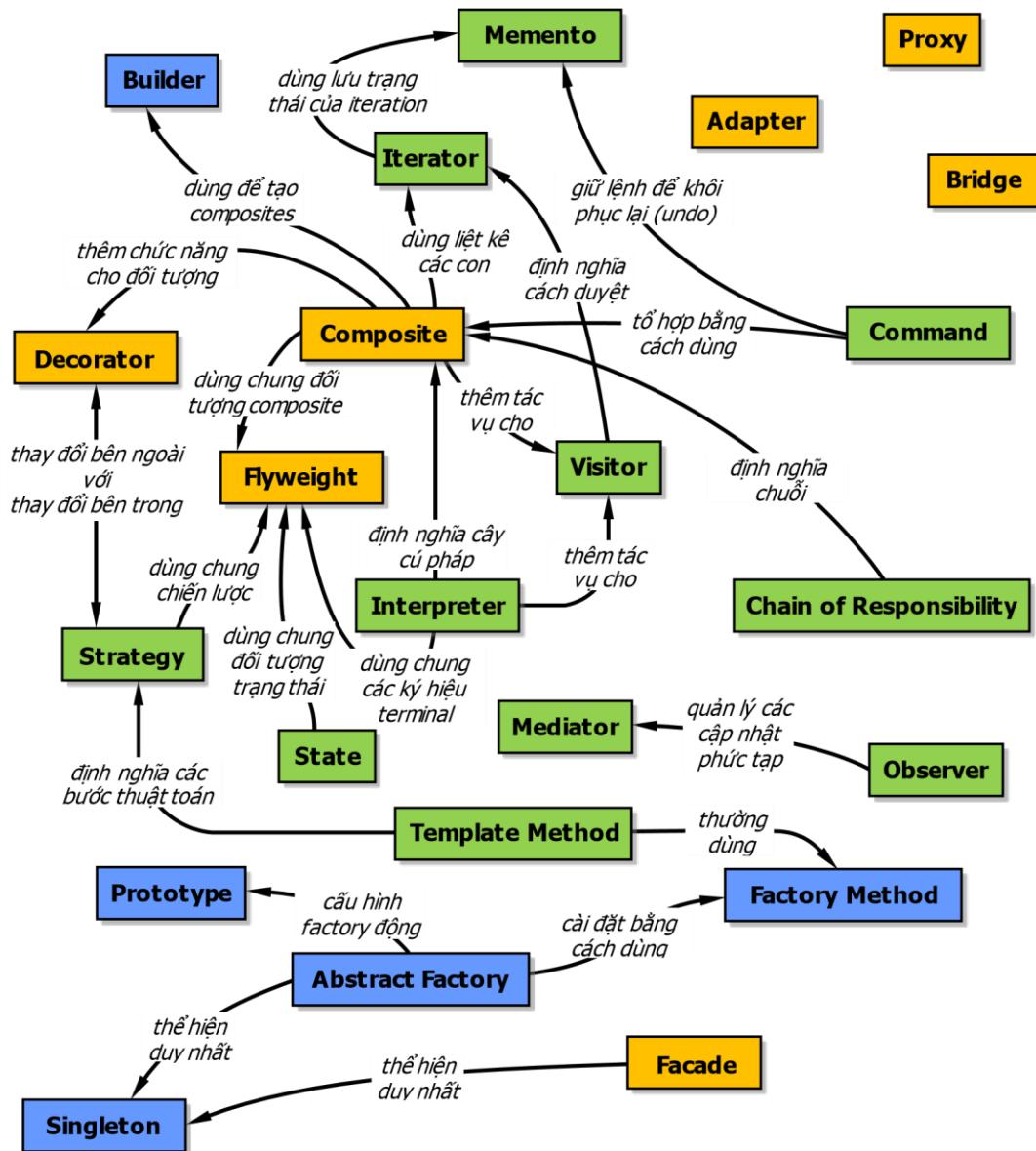
Peripheral Abstract Factory, Visitor, Decorator, Mediator, Type Object (thêm), Null Object (thêm), Extension Object (thêm)

Other Flyweight, Interpreter

Trong thực tế, các mẫu thiết kế có quan hệ bổ sung cho nhau, thường được dùng phối hợp với nhau. GoF trình bày quan hệ giữa các mẫu thiết kế như hình trong trang sau. Trong hình, các mẫu thiết kế Adapter, Bridge, Proxy có quan hệ nhưng không trình bày quan hệ đó.

Một mẫu thiết kế sẽ ánh xạ giữa một vấn đề thường gặp phải khi thiết kế và một giải pháp chung:

- Đã tạo trực tiếp các đối tượng có nhiệm vụ rõ ràng: hệ thống gắn kết chặt với cài đặt quá cụ thể làm mất đi tính linh động, khó mở rộng. Giải pháp: tạo các đối tượng một cách gián tiếp bằng cách dùng Abstract Factory, Factory Method, Prototype.
- Phụ thuộc vào các tác vụ cụ thể: hệ thống chỉ có một cách để đáp ứng yêu cầu. Giải pháp: tránh viết code cố định bằng cách dùng Chain of Responsibility, Command.
- Phụ thuộc vào nền tảng phần cứng hoặc phần mềm: ứng dụng khó chuyển đến các nền tảng khác. Giải pháp: giới hạn sự phụ thuộc hệ nền bằng cách dùng Abstract Factory và Bridge.
- Phụ thuộc vào giao diện người dùng hoặc code của Client: giao diện người dùng và code của Client có thể bị thay đổi nếu các đối tượng trong hệ thống thay đổi. Giải pháp: cách ly với Client, bằng cách dùng Abstract Factory, Bridge, Memento, Proxy.
- Phụ thuộc vào thuật toán: thuật toán sử dụng thay đổi thường xuyên; và khi thuật toán thay đổi, các đối tượng phụ thuộc nó buộc phải thay đổi. Giải pháp: cô lập thuật toán, dùng Builder, Iterator, Strategy, Template Method, Visitor.
- Ràng buộc quá chặt chẽ: các lớp liên kết với nhau quá chặt chẽ sẽ rất khó sử dụng lại, khó kiểm tra, bảo trì. Giải pháp: làm suy yếu liên kết quá chặt chẽ giữa các lớp bằng cách dùng Abstract Factory, Bridge, Chain of Responsibility, Command, Facade, Mediator, Observer.
- Mở rộng chức năng của lớp bằng thừa kế: thừa kế (inheritance) khó sử dụng, khó hiểu hơn tổng hợp (composition). Giải pháp: mở rộng chức năng tránh dùng thừa kế mà dùng tổng hợp, với Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy.
- Không dễ dàng tùy biến các lớp: các lớp không thể tiếp cận, không thể hiểu hoặc khó thay đổi. Giải pháp: không can thiệp vào lớp mà mở rộng bằng cách dùng Adapter, Decorator, Visitor.



Quan hệ giữa các mẫu thiết kế [Gamma et al. 1995]
Tham khảo sơ đồ quan hệ này sau khi đã hiểu rõ 23 mẫu thiết kế

Học các mẫu thiết kế là học kinh nghiệm từ thực tiễn, nâng cao tư duy thiết kế hướng đối tượng, nắm bắt nguyên tắc cấu trúc ứng dụng, biết cách tổ chức lại ứng dụng. Tuy nhiên, do bạn hoàn toàn có thể xây dựng chương trình không dùng các mẫu thiết kế, trước hết bạn cần nhận thức được lợi ích khi học và sử dụng các mẫu thiết kế.

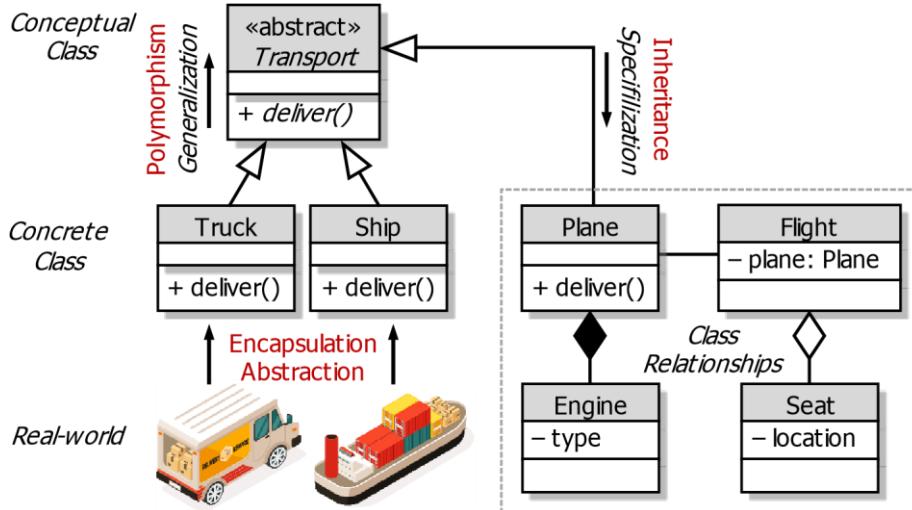
Lộ trình tiếp cận như sau:

- Đầu tiên, bạn tạm chấp nhận giả thuyết cho rằng các mẫu thiết kế là quan trọng trong việc thiết kế hệ thống phần mềm.
- Bạn phải nắm vững các nguyên lý thiết kế OOP và phải thông thạo ngôn ngữ UML để đọc hiểu sơ đồ thiết kế. Bạn có thể dùng UMLet, tại: <http://www.umllet.com> để vẽ sơ đồ lớp. Với các ví dụ trong tài liệu, bạn nên dùng ObjectAid UML Explorer, plugin của Eclipse, tại: <http://www.objectaid.net>. ObjectAid UML Explorer cho phép chuyển ngược các lớp Java trở lại thành sơ đồ lớp.
- Bạn phải nhận ra rằng bạn cần phải đọc về các mẫu thiết kế để biết khi nào bạn có thể sử dụng chúng. Thường thông qua các bài tập bạn thấy được các tình huống sử dụng chúng. Bạn cũng có thể áp dụng các mẫu thiết kế để tái cấu trúc code (refactoring) nhằm quen dần với việc sử dụng các mẫu thiết kế. Tham khảo "Refactoring to Patterns" của Joshua Kerievsky [5].
- Bạn phải hiểu các mẫu thiết kế một cách chi tiết (ý tưởng, kiến trúc, vấn đề giải quyết) đủ để biết loại nào trong chúng có thể giúp bạn giải quyết yêu cầu thiết kế hoặc vấn đề bạn gặp phải khi thiết kế.

Các mẫu thiết kế trong tài liệu được trình bày theo thứ tự như trong sách của GoF [1]. Tuy nhiên, theo chúng tôi, để dễ tiếp cận, bạn nên tìm hiểu theo thứ tự sau: Singleton, Iterator, Adapter, Decorator, Strategy, State, Factory Method, Observer, Facade, Template Method, rồi tự lựa chọn thứ tự tìm hiểu các mẫu thiết kế còn lại. Chú ý xem kỹ các mẫu thiết kế khó hiểu: Abstract Factory, Interpreter, Bridge và Mediator.

Các nguyên tắc thiết kế hướng đối tượng

Các trụ cột của OOP



- **Abstraction** (trừu tượng hóa) khi viết chương trình với OOP, bạn định hình các đối tượng của chương trình dựa trên các đối tượng trong thế giới thực. Các đối tượng trong chương trình không bao gồm mọi thuộc tính và hành vi của đối tượng thực. Thay vào đó, bạn chỉ lựa chọn mô hình hóa các thuộc tính và hành vi liên quan đến ngữ cảnh ứng dụng mà bạn muốn quản lý, lược bỏ các chi tiết không cần thiết khác. Kết quả của Abstraction là lớp (class), chứa các thuộc tính và hành vi cần quản lý.

- **Encapsulation** (đóng gói) là gom dữ liệu và hành vi làm việc trên dữ liệu đó vào trong một lớp, ẩn giấu dữ liệu sao cho chúng được truy cập có kiểm soát. Đối tượng chỉ bộc lộ một cách có giới hạn giao diện của nó với các đối tượng khác trong chương trình. Đóng gói giúp tách rời mã thành các module để chúng có thể phát triển độc lập. Kỹ thuật thay đổi code bên trong module mà không ảnh hưởng đến hành vi bên ngoài của một module gọi là refactoring (tái cấu trúc mã).

- **Inheritance** (thừa kế) là khả năng xây dựng một lớp mới dựa trên thuộc tính và hành vi của một lớp có sẵn. Thừa kế cho phép sử dụng lại mã của lớp có sẵn, mở rộng nó bằng cách đưa vào các chức năng bổ sung. Kết quả của việc sử dụng thừa kế là hệ thống phân cấp các lớp dẫn xuất có giao diện giống lớp cơ sở. Dẫn xuất một lớp con cũng được xem như quá trình đặc tả từ lớp cha mang tính khái niệm thành lớp con mang tính cụ thể. Ngoài thừa kế, có thể dùng cách tổng hợp (composition) để tạo lớp mới từ lớp có sẵn, sử dụng mối quan hệ giữa các lớp (association, dependency, aggregation, composition).

- **Polymorphism** (đa hình) để dễ quản lý các lớp cụ thể có nhiều đặc điểm giống nhau, bạn tổng quát hóa chúng thành lớp cha chung mang tính khái niệm, đưa các thuộc tính và hành vi giống nhau lên lớp cha trừu tượng này. Do lớp cha mang tính khái niệm nên hành vi chung cũng trừu tượng, không được cài đặt ở lớp cha mà chỉ cài đặt ở lớp con cụ thể. Hành vi này trở nên đa hình, nghĩa là có nhiều hình thức thực thi khác nhau tùy theo cài đặt của lớp con cụ thể. Ngôn ngữ OOP hỗ trợ việc xác định lớp cụ thể của đối tượng để thực thi hành vi cài đặt tương ứng với nó.

Coupling và Cohesion

Một module được sử dụng để thực hiện một nhiệm vụ (task) đơn. Một phần mềm thực hiện nhiều nhiệm vụ, nghĩa là nó dựa trên nhiều module. Một chức năng là một thành phần của một module. Thiết kế hướng đối tượng thường quan tâm đến mức độ kết nối (coupling) và gắn kết (cohesion) giữa các thành phần. **Cohesion** (gắn kết)

- Thể hiện mối quan hệ về chức năng giữa các thành phần trong một module.

- Cho thấy mức độ kết nối giữa các thành phần trong một module (intra-module).

Thường yêu cầu thiết kế đạt được gắn kết cao, nghĩa là các thành phần trong một module nên phân nhóm rõ ràng, hỗ trợ tốt với nhau để thực hiện nhiệm vụ của module đó. Điều này giúp cho module tập trung vào nhiệm vụ của nó.

Coupling (kết nối)

- Thể hiện mối quan hệ giữa một thành phần của module này với các thành phần của các module khác.

- Cho thấy mức độ kết nối giữa các module (inter-module).

Thường yêu cầu thiết kế đạt được kết nối lỏng, nghĩa là giảm thiểu việc module của bạn tham chiếu trực tiếp đến module khác. Vì các module ảnh hưởng ít đến nhau, việc thay đổi và bảo trì code trở nên dễ dàng, linh động.

Luật Demeter (LoD - Law of Demeter) hoặc nguyên tắc "kiến thức ít nhất" bảo đảm kết nối lỏng: Một phương thức của một đối tượng chỉ có thể gọi các phương thức của:

+ chính đối tượng đó.

+ các đối tượng là tham số của phương thức đó.

+ các đối tượng được tạo trong phương thức.

+ các đối tượng là trường/thuộc tính của chính đối tượng đó.

Ngoài cohesion và coupling, có nhiều hướng dẫn giúp bạn tạo nên thiết kế hướng đối tượng tốt:

- DRY (Don't Repeat Yourself), không nên viết những đoạn code trùng lặp trong cùng một chương trình (WET - Write Everything Twice) mà nên dùng lớp trừu tượng hoặc viết thành phương thức để dùng chung.

- Chương trình nên được thiết kế sao cho mọi thay đổi trên nó sẽ chỉ ảnh hưởng đến một phần nhỏ, có thể dự đoán được. - Đóng gói những gì dễ thay đổi (thuật toán, trạng thái, lệnh, ...).

- Nên dùng tổng hợp (composition) hơn là dùng thừa kế (inheritance).

- Client luôn làm việc với phần trừu tượng (interface, abstract class), không trực tiếp làm việc với cài đặt cụ thể. - Thay vì thực hiện tất cả hành vi, ủy nhiệm (delegation) cho các lớp phù hợp thực hiện hành vi mong muốn.

SOLID

Các hướng dẫn trên nói chung dựa trên năm nguyên tắc thiết kế hướng đối tượng, gọi tắt là các nguyên tắc SOLID: -

Single Responsibility

- Open-Closed
- Liskov's Substitution
- Interface Segregation
- Dependency Inversion

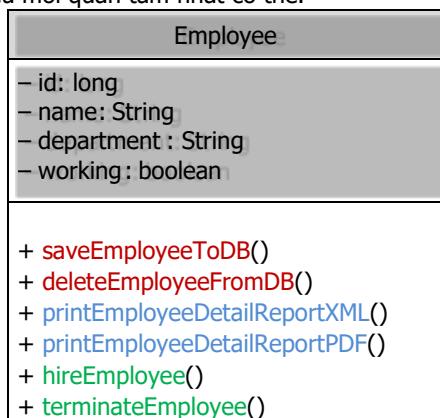
Năm nguyên tắc này có mối quan hệ tương hỗ với nhau. Chúng được áp dụng khi xây dựng các mẫu thiết kế.

1. Single Responsibility (SRP)

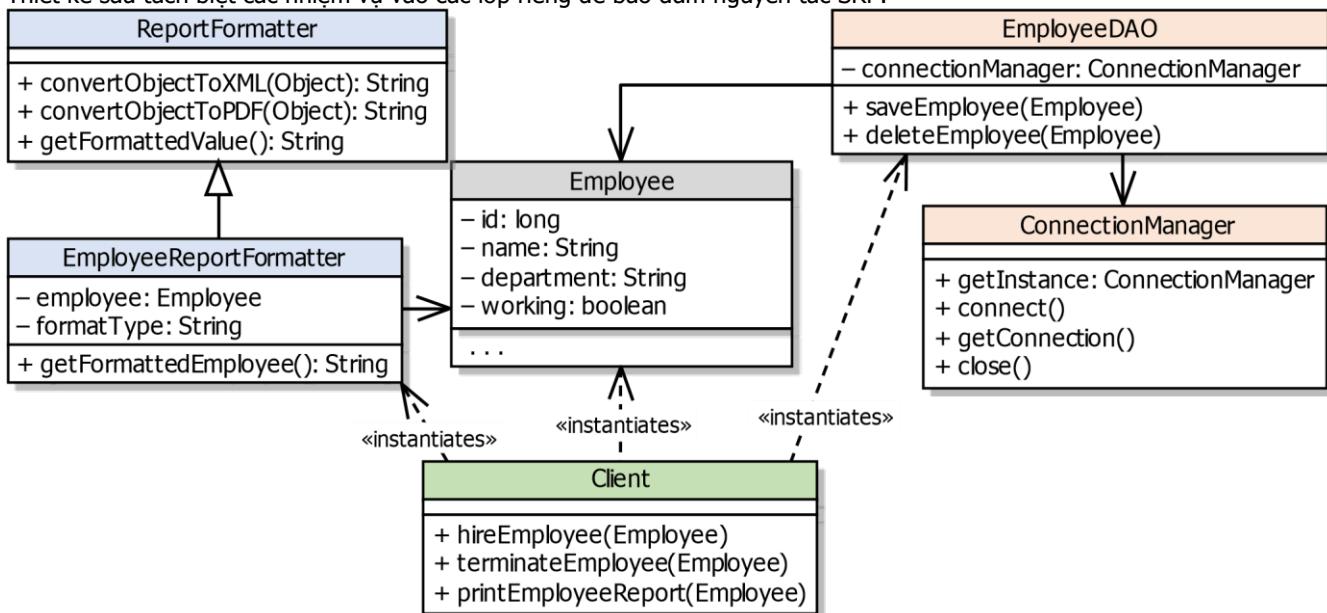
[Tom DeMarco và Meilir Page-Jones] Mỗi lớp chỉ nên đảm đương một nhiệm vụ và tất cả các phương thức của nó phải liên quan đến nhiệm vụ đó. Một nhiệm vụ là một trục thay đổi, nhiều nhiệm vụ sẽ gây nên nhiều lý do thay đổi lớp, làm cho lớp trở nên khó bảo trì. Vì vậy, một lớp chỉ có một lý do để thay đổi.

Lưu ý, một nhiệm vụ không phải là một phương thức, mà là nhóm phương thức làm cho đối tượng thay đổi với lý do tương tự nhau. Ví dụ: nhiệm vụ kết nối (connect(), disconnect()), nhiệm vụ truyền thông (send(), receive()).

Nói cách khác, hãy nhóm các đối tượng mà chúng thay đổi theo lý do giống nhau, và tách biệt các đối tượng mà chúng thay đổi theo các lý do khác nhau. Ví dụ, lớp Employee sau đảm nhiệm cùng lúc ba nhiệm vụ nên vi phạm nguyên tắc SRP, lớp như vậy gọi là "God Object", do cố gắng xử lý nhiều mối quan tâm nhất có thể.



Thiết kế sau tách biệt các nhiệm vụ vào các lớp riêng để bảo đảm nguyên tắc SRP:



Vi phạm nguyên tắc SRP	Đảm bảo nguyên tắc SRP
<pre> class Employee { -name +getName() +printTimeSheetReport() } </pre>	<pre> class TimeSheetReport { +print(Employee) } class Employee { -name +getName() } </pre>

2. Open-Closed (OCP)

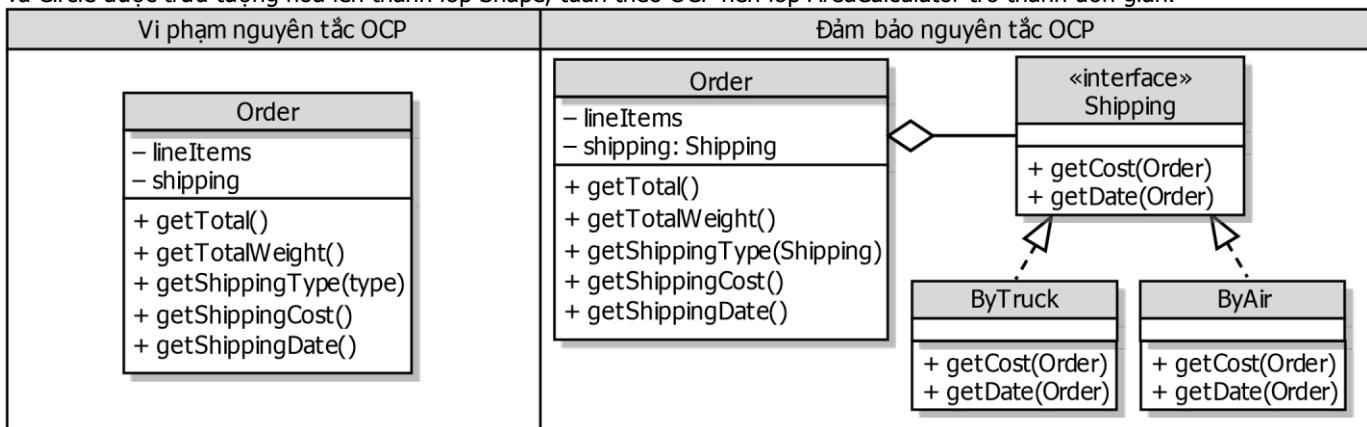
[Bertrand Meyer] Module được thiết kế sao cho một khi đưa vào ứng dụng, nó có thể mở rộng được, dễ dàng thêm hành vi mới, nhưng phải đóng để tránh làm thay đổi module hiện có. Nguyên tắc:

- Open for extensibility: module được sử dụng như một lớp cơ bản (base) để dễ tạo lớp mới khi cần mở rộng hành vi của module.
- Closed for modification: tránh thay đổi với module hiện tại, liên quan đến nguyên tắc SRP.

Để thực hiện nguyên tắc OCP, trước tiên phải cài đặt module theo nguyên tắc SRP. Sau đó, thực hiện nguyên tắc OCP bằng cách trừu tượng hóa. Do dẫn xuất từ một lớp trừu tượng là đóng, không cho phép thay đổi, bởi vì lớp trừu tượng đã định nghĩa cố định hành vi; nhưng hành vi đó có thể được mở rộng bằng cách cài đặt mới tại các lớp dẫn xuất.

Vi phạm nguyên tắc OCP	Đảm bảo nguyên tắc OCP
<pre>class Square { double side; } class Circle { double radius; } class AreaCalculator { public static double Area(Object[] shapes) { double area = 0; for (Object object : shapes) { if (object instanceof Square) { Square square = (Square) object; area += square.side * square.side; } else if (object instanceof Circle) { Circle circle = (Circle) object; area += circle.radius * circle.radius * Math.PI; } } return area; } }</pre>	<pre>abstract class Shape { public abstract double area(); } class Square extends Shape { double side; @Override public double area() { return side * side; } } class Circle extends Shape { double radius; @Override public double area() { return radius * radius * Math.PI; } } class AreaCalculator { public static double Area(Shape[] shapes) { return Stream.of(shapes) .mapToDouble(Shape::area) .sum(); } }</pre>

Thiết kế bên trái, lớp AreaCalculator sử dụng trực tiếp các lớp Square và Circle, nên vi phạm nguyên tắc OCP. Nếu mở rộng, ví dụ thêm lớp Triangle, buộc phải thay đổi nhiều trong lớp AreaCalculator. Thiết kế bên phải theo nguyên tắc SRP, các lớp Square và Circle được trừu tượng hóa lên thành lớp Shape, tuân theo OCP nên lớp AreaCalculator trở thành đơn giản.



3. Liskov's Substitution (LSP)

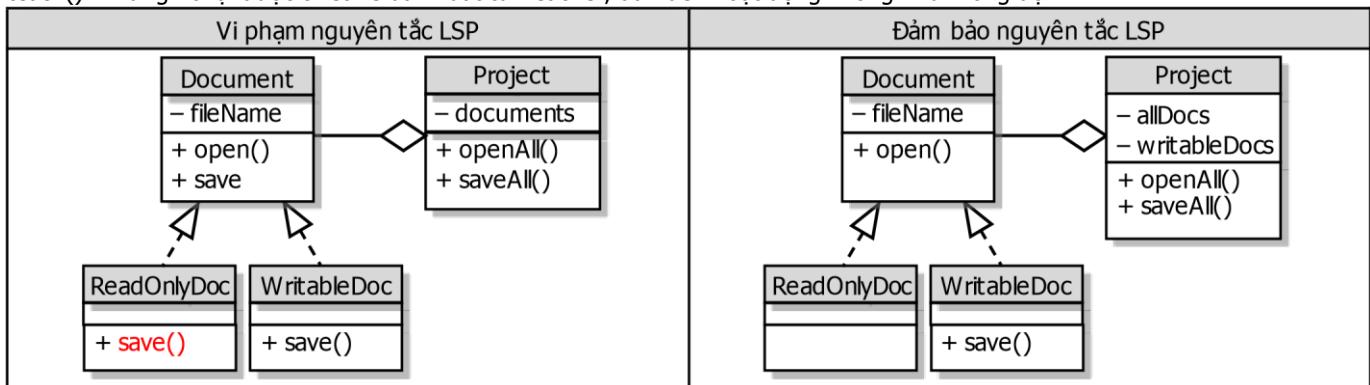
[Barbara Liskov] Kiểu dẫn xuất phải có khả năng thay thế được kiểu cơ sở của nó. Nói một cách đơn giản, lớp dẫn xuất phải được cài đặt sao cho nó không phá vỡ chức năng của lớp cha dưới góc nhìn của người dùng. Người dùng sử dụng một lớp, và họ mong tiếp tục làm việc đúng như thế với lớp dẫn xuất từ lớp đó.

LSP thường bị vi phạm khi bạn loại bỏ một số tính năng từ lớp cha. Ví dụ, từ lớp cha tạo lớp con, trong đó lớp con sử dụng lớp cha để cài đặt các tính năng. Cân nhắc sử dụng tổng hợp (composition) thay cho thừa kế khi tạo lớp để tránh vi phạm LSP.

Vi phạm nguyên tắc LSP	Đảm bảo nguyên tắc LSP
------------------------	------------------------

<pre>abstract class Teacher { String name; public abstract void teach(); private void takeAttendance() { System.out.println("Take attendance"); } private void collectPaper() { System.out.println("Collect papers"); } public void performOtherTasks() {</pre>	<pre>class Staff { String name; private void takeAttendance() { System.out.println("Take attendance"); } private void collectPaper() { System.out.println("Collect papers"); } public void performOtherTasks() { takeAttendance(); collectPaper(); } }</pre>
<pre> takeAttendance(); collectPaper(); } } class MathTeacher extends Teacher { @Override public void teach() { System.out.println("Calculate math"); } } class SubstitutionTeacher extends Teacher { @Override public void teach() { // cannot teach? throw exception? } }</pre>	<pre>interface Instructor { void teach(); } class MathTeacher extends Staff implements Instructor { @Override public void teach() { System.out.println("Calculate math"); } } class SubstitutionTeacher extends Staff { }</pre>

Thiết kế bên trái vi phạm nguyên tắc LSP. Lớp SubstitutionTeacher không phải là Teacher, nó không cần đến phương thức teach(). Nhưng nó lại được thiết kế dẫn xuất từ Teacher, dẫn đến hoạt động không như mong đợi.



4. Interface Segregation (ISP)

[Robert C. Martin] Giao diện lớn nên tách thành nhóm các giao diện (interface) có chức năng đặc thù hơn, mỗi giao diện thu hẹp như vậy gọi là giao diện vai trò (role interface), phục vụ cho một tập khách hàng riêng.

Thiết kế theo ISP giữ cho hệ thống tách biệt, dễ dàng tổ chức lại, thay đổi hoặc tái bố trí.

Vi phạm nguyên tắc ISP	Đảm bảo nguyên tắc ISP
------------------------	------------------------

```

interface IWorker {
void work(); void
eat();
} class Worker implements
IWorker {
@Override public void work() {
System.out.println("worker working");
}

@Override public void eat() {
System.out.println("eating");
}
}
class Robot implements IWorker {
@Override public void work() {
System.out.println("robot working");
}

@Override public void eat() {
System.out.println("no need eat");
}
}
public class Manager_noISP {
public static void main(String[] args) {
IWorker[] list = {new Worker(), new Robot()};
Stream.of(list).forEach(IWorker::work);
}
}

```

```

interface IWorkable { void work(); }

interface IFeedable { void eat(); }

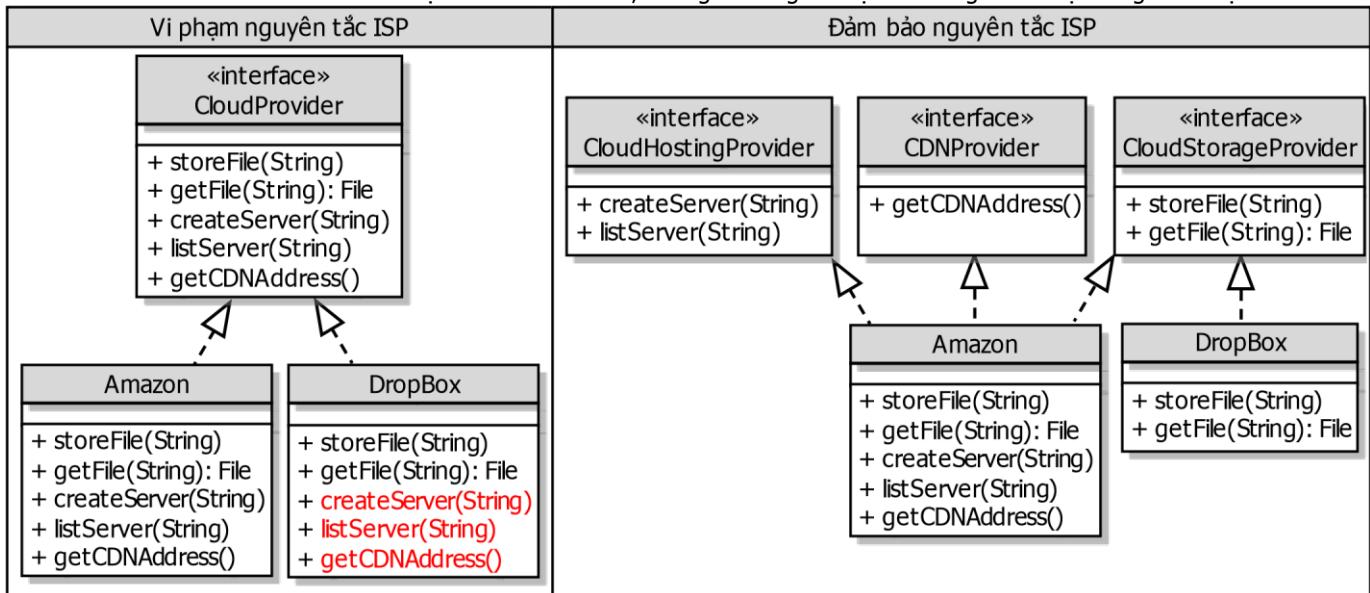
interface IWorker extends IFeedable, IWorkable {
}
class Worker implements IWorker {
{
@Override public void work() {
System.out.println("worker working");
}

@Override public void eat() {
System.out.println("eating");
}
}
class Robot implements IWorkable {
@Override public void work() {
System.out.println("robot working, not
eating");
}
}

public class Manager_ISP { public static void
main(String[] args) { IWorkable[] list =
{new Worker(), new Robot()};
Stream.of(list).forEach(IWorkable::work);
}
}

```

Các interface IWorkable và IFeedable đặc thù hơn IWorker, chúng là các giao diện vai trò giữ cho hệ thống tách biệt.



5. Dependency Inversion (DIP)

[Robert C. Martin] Ngược với kiến trúc truyền thống, những module ở mức cao (nơi phối hợp hoạt động của nhiều module ở mức thấp) không nên phụ thuộc trực tiếp vào những module mức thấp (nơi thực hiện chức năng cơ bản), cả hai nên phụ thuộc thông qua lớp trừu tượng (hoặc interface) để tránh kết nối chặt.

Lớp trừu tượng (hoặc interface) không thường xuyên thay đổi nên ít gây ảnh hưởng đến các module phụ thuộc nó.

Kiến trúc: thay thế (high-module □ low-module) bằng (high-module □ [interface □□ low-module])

```

interface Writer { void writeOn(); }

interface Reader { void readIn(); }

class Keyboard implements Reader {
    @Override public void readIn() {
        System.out.println("Read in keyboard");
    }
}

class Printer implements Writer {
    @Override public void writeOn() {
        System.out.println("Write on paper");
    }
}

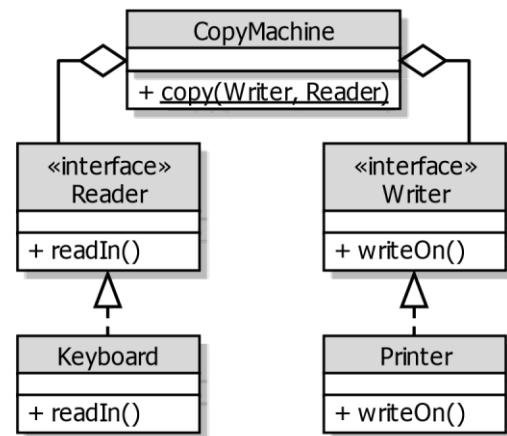
public class CopyMachine {
    public static void copy(Writer w, Reader r) {
        r.readIn();
        w.writeOn();
    }
}

```

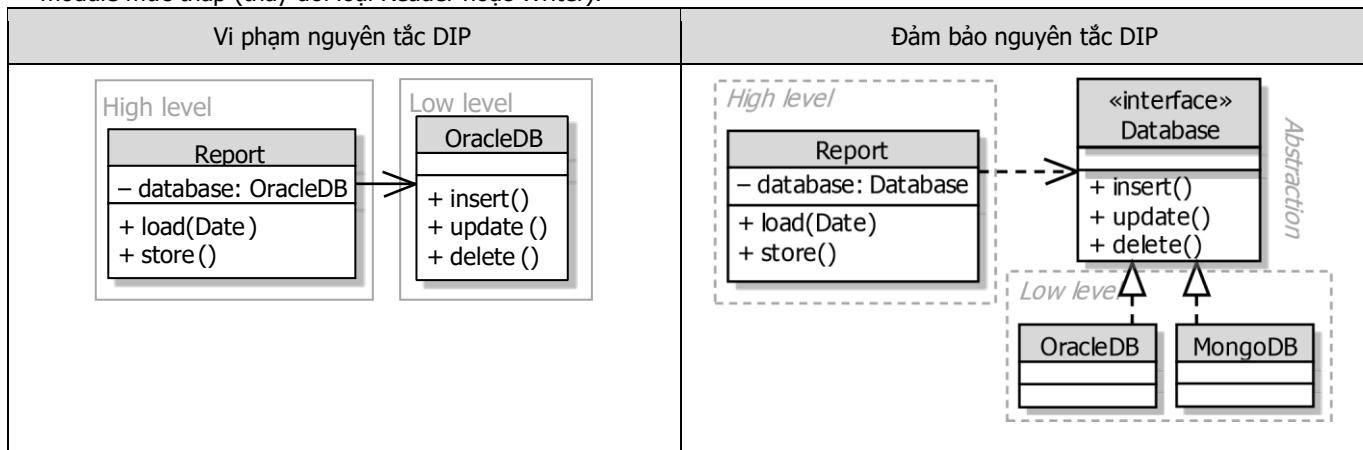
```

    public static void main(String[] args) {
        Writer w = new Printer();
        Reader r = new Keyboard();
        copy(w, r);
    }
}

```



Lớp CopyMachine không phụ thuộc trực tiếp vào lớp Keyboard hoặc lớp Printer. Nó phụ thuộc gián tiếp thông qua các lớp trung gian Reader và Writer. Kiến trúc này giúp module mức cao (copy() của CopyMachine) không bị ảnh hưởng khi thay đổi các module mức thấp (thay đổi loại Reader hoặc Writer).



GRASP

Tổng quan

Mục tiêu của thiết kế hướng đối tượng là: định danh các lớp (danh từ), gán trách nhiệm (động từ) cho các lớp và xác định mối quan hệ giữa các lớp. GRASP (General Responsibility Assignment Software Principles) được Craig Larman mô tả trong cuốn sách "Applying UML and Patterns" [17]. GRASP giúp bạn hiểu các nguyên lý thiết kế hướng đối tượng cơ bản để áp dụng thiết kế một cách có phương pháp, hợp lý và có thể giải thích được. Cách tiếp cận của GRASP dựa trên việc phân công trách nhiệm, chỉ rõ đối tượng hoặc lớp (WHO) nào chịu trách nhiệm về hành động hoặc vai trò (WHAT) nào. Trách nhiệm được hiểu là nhiệm vụ của một đối tượng về hành vi của nó:

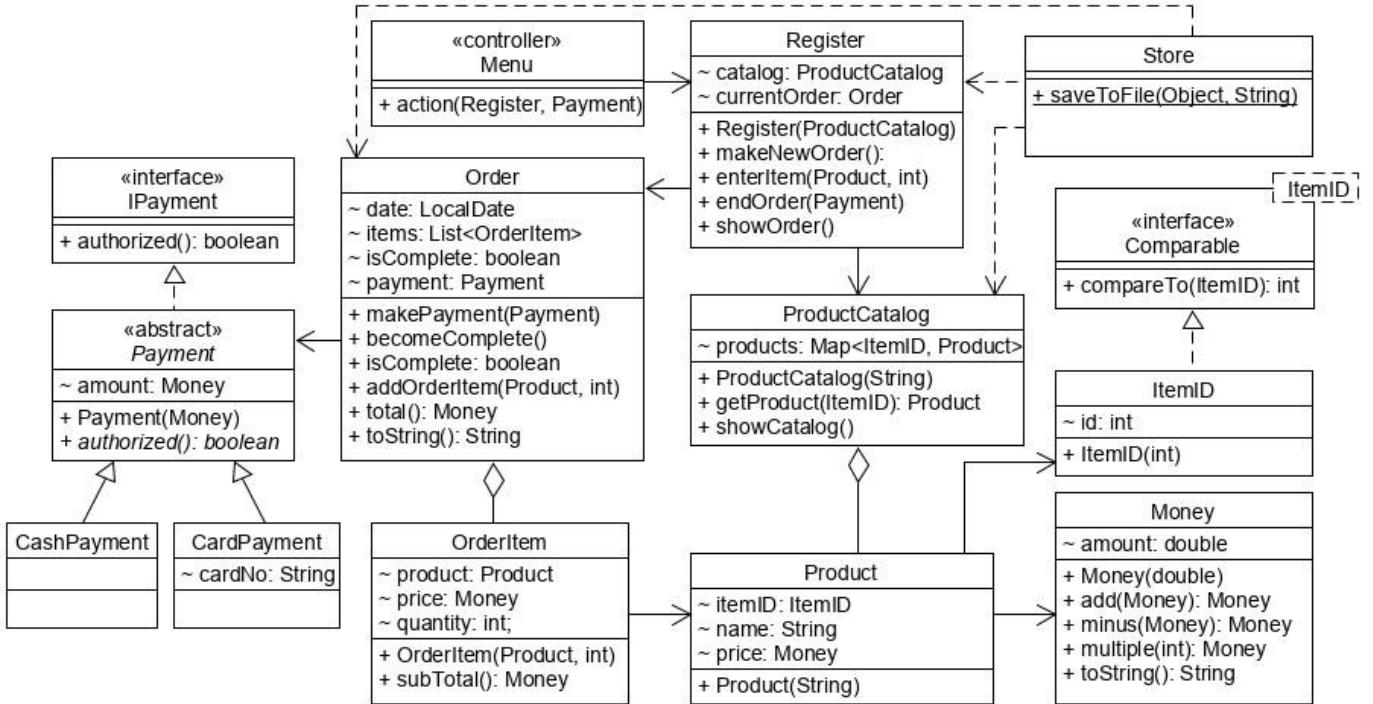
- **biết:** biết về dữ liệu được đóng gói riêng tư của nó, biết về các đối tượng nó liên quan và biết về những thứ mà nó có thể truy cập được hoặc tính toán được.
- **làm:** tự làm một điều gì đó, chẳng hạn như tạo một đối tượng hoặc thực hiện các tính toán, khởi tạo hành động của các đối tượng khác, kiểm soát và điều phối các hoạt động của các đối tượng khác.

Ví dụ

- Order có trách nhiệm tạo OrderItem (làm).
- Order có trách nhiệm cho biết tổng giá trị đơn hàng của nó (biết).

Trách nhiệm được thực hiện bằng cách sử dụng các phương thức hoạt động độc lập hoặc cộng tác với các phương thức và các đối tượng khác.

Các nguyên tắc của GRASP được biểu diễn dưới dạng các mẫu thiết kế, bao gồm 9 mẫu: Information Expert, Creator, Controller, Low Coupling, High Cohesion, Polymorphism, Pure Fabrication, Indirection và Protected Variations. Chúng ta dùng ví dụ có sơ đồ lớp sau để minh họa và giải thích các nguyên tắc GRASP:



1. Information Expert

Vấn đề

Nguyên tắc chung về phân công trách nhiệm cho các lớp là gì?

Giải pháp

Gán trách nhiệm cho "chuyên gia thông tin" (expert information), nghĩa là lớp có đủ thông tin cần thiết để hoàn thành trách nhiệm. **Tiếp cận**

- Nêu rõ trách nhiệm.
- Tìm kiếm các lớp trong mô hình thiết kế¹ có thông tin cần thiết đủ để hoàn thành trách nhiệm (biết cách tốt nhất để hoàn thành trách nhiệm). Nếu không, quan sát mô hình miền và cố gắng sử dụng hoặc mở rộng để tạo ra các lớp thiết kế đáp ứng yêu cầu. - Phân công trách nhiệm.

Việc thực hiện trách nhiệm thường đòi hỏi thông tin được lan truyền qua các lớp đối tượng khác nhau, nghĩa là có sự cộng tác của các chuyên gia thông tin "một phần" để thực hiện trách nhiệm.

Lớp	Thông tin "biết"	Trách nhiệm (phương thức)
Product	price	Lấy đơn giá (getPrice())
OrderItem	quantity subTotal = quantity × price	Tổng giá trị mục hàng (subTotal())
Order	List<OrderItem>, orderDate, tax, total = tổng các subTotal + tax	Tổng giá trị đơn hàng (total())

Lợi ích

- Bảo đảm tính đóng gói vì các đối tượng sử dụng dữ liệu của riêng chúng để hoàn thành nhiệm vụ.
- Hỗ trợ kết nối lồng.
- Trách nhiệm được phân phối trên các lớp có thông tin cần thiết.

2. Creator

Vấn đề

Ai nên được giao trách nhiệm tạo ra đối tượng mới của một lớp?

Giải pháp

Tạo đối tượng là một trách nhiệm đặc biệt, cần tìm "chuyên gia thông tin" có thể tạo đối tượng. Gán cho lớp A trách nhiệm tạo đối tượng lớp B nếu:

- A tự hợp (aggregate) toàn bộ hoặc một phần B.
- A chứa các đối tượng B.
- A ghi nhận các đối tượng B.
- A sử dụng các đối tượng B.
- A khởi tạo dữ liệu cần thiết để tạo các đối tượng B.

Tiếp cận

- Quan sát kỹ mô hình miền và trả lời câu hỏi: ai nên tạo các lớp này?
- Tìm các lớp học có quan hệ tạo thành, tự hợp, ... - Phân công trách nhiệm tạo.

¹ Mô hình thiết kế (design model) là mô hình minh họa các lớp trong phần mềm. Mô hình miền (domain model) là mô hình minh họa các lớp khái niệm của miền (lĩnh vực) thực tế.

Lớp	Mỗi quan hệ với các lớp khác	Trách nhiệm (phương thức)
Register	Chứa List<Order>	Tạo Order (createOrder())
Order	Quan hệ tự hợp với OrderItem	Tạo OrderItem(addOrderItem)

Lợi ích

- Hỗ trợ kết nối lỏng.
- Tách biệt logic tạo đối tượng với các logic nghiệp vụ khác.

3. Controller

Vấn đề

Ai chịu trách nhiệm xử lý sự kiện hệ thống? Các sự kiện này đến từ các Actor bên ngoài, từ giao diện người dùng, ...

Giải pháp

Gán trách nhiệm xử lý thông điệp sự kiện hệ thống cho lớp thể hiện một trong các điều sau:

- Lớp đại diện cho toàn bộ hệ thống, thiết bị hoặc hệ thống con.
- Lớp đại diện cho một trường hợp sử dụng (use case) trong đó sự kiện hệ thống xảy ra.
- Lớp đại diện cho nghiệp vụ tổng thể (facade).
- Đại diện cho một cái gì đó trong thế giới thực đang hoạt động. Các lớp trên gọi là bộ điều khiển (controller):
- Nhìn thấy các thông điệp được gửi đến các "chuyên gia" trong mô hình, ví dụ gửi đến Order.
- Nó tách biệt phần View (Presentation) và phần Model (Business logic).

Lưu ý là bộ điều khiển không nên làm nhiều việc, nó úy nhiệm công việc cần được thực hiện cho các đối tượng khác. Khi có ít sự kiện, dùng bộ điều khiển "mặt tiền" (facade). Với hệ thống có nhiều use case, thường có các bộ điều khiển cho từng use case. Các lớp này thường không tự làm việc, mà úy quyền nó cho lớp khác. **Tiếp cận**

- Quan sát kỹ mô hình miền, tìm các lớp làm công việc liên quan đến một số chức năng cụ thể và cần có điều hành.
- Thêm lớp mới chịu trách nhiệm điều khiển hoặc điều hành, tất cả mã từ client sẽ truy cập chức năng thông qua lớp này.

Lớp	Lý do chọn
Register	Lớp đại diện cho nghiệp vụ tổng thể. Lớp Menu là phần View.

Phần Controller trong mô hình MVC thiết kế theo nguyên tắc này.

Lợi ích

- Hỗ trợ kết nối lỏng.
- Dễ hiểu, dễ bảo trì.

4. Low Coupling

Vấn đề

Phân công trách nhiệm sao cho kết nối lỏng, nghĩa là hỗ trợ sự phụ thuộc thấp và tăng khả năng sử dụng lại.

Với thiết kế kết nối lỏng, các thay đổi trong một thành phần (hệ thống con, hệ thống lớp,...) sẽ hạn chế các thay đổi "đổ xuống" (cascade) lan đến các thành phần khác. Ngoài ra từng thành phần dễ cô lập để hiểu.

Giải pháp

Gán trách nhiệm sao cho các lớp kết nối tối thiểu với nhau.

Tiếp cận

- Tìm kiếm các lớp có nhiều liên kết đến các lớp khác.
- Tìm các phương thức dựa vào nhiều phương thức khác, hoặc dựa vào các phương thức của các lớp khác (các dependency). Mục tiêu là giảm thiểu các lớp phụ thuộc.
- Thiết kế lại để gán trách nhiệm cho các lớp khác nhau theo cách mà chúng có sự liên kết và sự phụ thuộc thấp hơn. Các hình thức kết nối phổ biến từ A đến B bao gồm:
 - A có một dữ liệu thành viên tham chiếu đến B.
 - A có một phương thức nhận B là tham số.
 - A là lớp con trực tiếp hoặc gián tiếp của B.
 - A cài đặt interface B.

Lớp	Quan hệ với các lớp khác	Trách nhiệm (Phương thức)	Nhận xét
Register (version 1)	Register là creator của Order và Payment	Register gọi makePayment() của Order tạo Payment trong Order, rồi gọi addPayment(Payment) của Order	Nếu Order hoặc Payment thay đổi sẽ ảnh hưởng đến Register (kết nối chặt)
Register/Order (version II)	Register là creator của Order Order là creator của Payment	Register gọi makePayment() của Order, makePayment() đầu tiên tạo Payment.	Trách nhiệm đã được phân chia, Register không cần biết Payment (kết nối lỏng).

Lợi ích

- Dễ bảo trì, thay đổi ảnh hưởng ít hoặc không ảnh hưởng đến các thành phần khác.
- Dễ hiểu từng thành phần nếu cách ly chúng.
- Khả năng sử dụng lại cao.

5. High Cohesion **Vấn đề**

Phân công sao cho các trách nhiệm có liên quan chặt chẽ với nhau. Độ gắn kết liên quan đến quan hệ cộng tác giữa phương thức trong lớp (hoặc lớp trong gói) khi thực thi trách nhiệm.

Giải pháp

Gán trách nhiệm để sự gắn kết vẫn cao.

Tiếp cận

Các ví dụ về gắn kết thấp:

- Chức năng, phương thức không liên quan với người dùng.
 - Phương thức có nhiều AND/OR trong tên, ngụ ý phương thức sẽ thực thi nhiều hàm khiến việc sử dụng lại khó khăn.
 - Lớp xử lý công việc ngoài hệ thống con (package, namespace).
- Craig Larman đã mô tả mức độ gắn kết từ thấp đến cao
- Gắn kết rất thấp: lớp chịu trách nhiệm cho nhiều thứ trong các lĩnh vực không liên quan.
 - Gắn kết thấp: lớp chịu trách nhiệm cho nhiều thứ trong các lĩnh vực liên quan.
 - Gắn kết vừa phải: lớp có ít trách nhiệm trong các lĩnh vực khác nhau, các lĩnh vực này liên quan đến lớp nhưng không liên quan đến nhau.
 - Gắn kết cao: lớp có ít trách nhiệm trong một lĩnh vực và cộng tác với các lớp khác để hoàn thành nhiệm vụ.

Lớp	Quan hệ với các lớp khác	Trách nhiệm (Phương thức)	Nhận xét
Register (version 1)	Register là creator của Payment, Order, OrderItem	Register thực hiện cả makePayment(), makeOrder() và addOrderItem()	Các trách nhiệm này không liên quan, dẫn đến gắn kết thấp.
Register/Order (version II)	Register là creator của Order Order là creator của Payment, OrderItem	Register gọi makePayment() của Order, makePayment() đầu tiên tạo Payment.	Phân chia trách nhiệm theo quan hệ giữa chúng làm giảm độ phức tạp dẫn đến gắn kết cao.

Lợi ích

- Dễ hiểu, thiết kế rõ ràng.
- Dễ bảo trì, cải tiến được đơn giản hóa.
- Hỗ trợ kết nối lỏng.
- Các lớp có kích thước nhỏ và các chức năng liên quan cao làm tăng khả năng sử dụng lại.

6. Polymorphism **Vấn đề**

Gán trách nhiệm như thế nào nếu có các lựa chọn thay thế hoặc hành vi thay đổi theo kiểu? Làm thế nào để tạo ra các thành phần mềm có thể "cắm" vào được²?

Giải pháp

Gán trách nhiệm là hành vi đa hình.

Tiếp cận

- Xác định các lớp (loại) sẽ thay đổi hành vi
- Gán trách nhiệm cho hành vi đa hình

Lớp	Quan hệ với các lớp khác	Trách nhiệm (Phương thức)	Nhận xét
Payment	Có nhiều cách kiểm tra thanh toán tùy theo kiểu Payment.	Hành vi authorized() thay đổi theo kiểu Payment.	Payment là lớp mang tính khái niệm do có phương thức đa hình authorized().
CashPayment, CreditPayment	Các lớp con của Payment: CashPayment, CreditPayment	Cài đặt hành vi authorized() cụ thể cho các lớp con của Payment.	Nếu có kiểu thanh toán mới (CheckPayment), không gây nhiều thay đổi cho lớp sử dụng Payment.

Lợi ích

- Dễ thêm các lớp không được dự kiến do chưa xác định được nhu cầu.
- Dễ bảo trì và đơn giản hóa cải tiến.

7. Pure Fabrication

Vấn đề

Ai được gán trách nhiệm khi có một lớp "chuyên gia" vi phạm gắn kết cao và kết nối lỏng.

Giải pháp

Gán trách nhiệm xử lý thông điệp sự kiện hệ thống cho một lớp "giả" không thể hiện cho một khái niệm trong domain. Chỉ định tập trách nhiệm gắn kết cho lớp đó để hỗ trợ sự gắn kết cao, kết nối thấp và sử dụng lại. Vì lớp này là lớp "giả", nên nó có thể được gọi là kết quả của trí tưởng tượng hoặc "thuần túy chế tạo" ra.

Tiếp cận

- Quan sát kỹ mô hình miền, định vị các lớp có gắn kết thấp và kết nối cao.
- "Chế tạo" lớp mới chịu trách nhiệm về chức năng gây ra sự gắn kết thấp và kết nối cao.

Bạn muốn lưu Order vào cơ sở dữ liệu. Theo nguyên tắc "chuyên gia thông tin" phương thức saveToDatabase() được gán cho Order. Tuy nhiên, phương thức này dẫn đến một số lượng lớn các hoạt động hướng cơ sở dữ liệu, không liên quan gì đến Order. Do chịu trách nhiệm về nhiều thứ trong lĩnh vực không liên quan nên lớp Order có gắn kết thấp. Ngoài ra, khi thực hiện

² pluggable, thành phần có thể thay thế bằng thành phần khác mà không ảnh hưởng đến mã của client.

các hoạt động cơ sở dữ liệu, Order cần sử dụng đến giao diện cơ sở dữ liệu dẫn đến kết nối chặt. Ta cũng nhận thấy trách nhiệm lưu trữ tượng vào một cơ sở dữ liệu là một trách nhiệm chung chung.

Giải pháp là tạo một lớp mới Store, tương tác với giao diện cơ sở dữ liệu và tham chiếu đến Object. Object có thể là Order, Payment, Register. Lớp Store có sự gắn kết cao (thực hiện trách nhiệm duy nhất là lưu Order) và kết nối lỏng (lớp sử dụng chung, có thể sử dụng lại).

Theo Craig Larman, có 2 cách tiếp cận thiết kế cho hành vi "chế tạo thuận túy".

- Phân rã phần presentation: Thiết kế các đối tượng theo cách chúng thể hiện trong domain.
- Phân rã hành vi: Thiết kế các đối tượng theo cách của chúng hành động. Thường là chức năng trung tâm hoặc đóng gói thuật toán.

Lớp	Lý do chọn
Order	saveToDatabase() làm lớp Order gắn kết thấp và kết nối chặt.
Store	saveToDatabase(Object) làm lớp Store gắn kết cao và kết nối lỏng. Tham số Object làm lớp Store có trách nhiệm chung.

Lợi ích

- Hỗ trợ kết nối lỏng.
- Kết quả gắn kết cao. - Tăng khả năng sử dụng lại.

8. Indirection **Vấn đề**

Làm thế nào kết hợp kết nối lỏng (tách rời các đối tượng) với khả năng sử dụng lại cao?

Giải pháp

Gán trách nhiệm cho một đối tượng trung gian để dàn xếp giữa các thành phần hoặc dịch vụ để chúng không kết nối trực tiếp.

Tiếp cận

- Quan sát kỹ mô hình miền và định vị các lớp có kết nối lỏng hoặc kết nối trực tiếp.
- Gán trách nhiệm cho một lớp trung gian lần lượt cộng tác với hai lớp để tránh kết nối trực tiếp.

Lưu ý là Pure Fabrication tập trung vào xử lý kết nối lỏng hơn là khả năng sử dụng lại.

Để tránh kết nối chặt, trực tiếp giữa Order và lớp chứa các dịch vụ cơ sở dữ liệu cụ thể, cần lớp trung gian tách quan hệ giữa lớp Order và giao diện cơ sở dữ liệu. Lớp trung gian PersistentStorageBroker sẽ tương tác với giao diện cơ sở dữ liệu và lưu thể hiện của lớp Order. Lớp Order gọi PersistentStorageBroker ủy nhiệm trách nhiệm thực hiện các hoạt động cơ sở dữ liệu. Lớp Order không bị ảnh hưởng từ việc lưu thể hiện Order riêng vào cơ sở dữ liệu, do đó dẫn đến sự gắn kết cao và kết nối lỏng. Lớp PersistentStorageBroker cũng gắn kết cao bằng cách thực hiện trách nhiệm duy nhất là lưu đối tượng.

Mẫu thiết kế Adapter, Mediator là các ví dụ tốt cho nguyên tắc Indirection.

Lợi ích

- Hỗ trợ kết nối lỏng.
- Kết quả tăng khả năng sử dụng lại.

9. Protected Variations **Vấn đề**

Làm thế nào chỉ định trách nhiệm sao cho những thay đổi (biến thể) của thành phần này mà không gây ra tác động không mong muốn đến các thành phần khác?

Giải pháp

- Dự đoán các điểm biến đổi (các biến thể) có thể.
- Gán trách nhiệm để tạo giao diện ổn định bao quát các biến thể hiện tại và tương lai. Điều này liên quan đến việc tạo ra các lớp hoàn toàn mới để đóng gói các biến thể.

Nguyên tắc Open-Closed trong SOLID có cùng ý tưởng với Protected Variations.

Tiếp cận

- Tìm kiếm các thành phần có kết nối chặt và tương đối dễ bị thay đổi.
- Xác định các đối tượng của biến thể được dự đoán.
- Chỉ định các trách nhiệm bằng cách tạo ra giao diện ổn định xung quanh các biến thể. Tóm lại, tìm những biến thể và đóng gói nó.

Lợi ích

Nếu bạn đoán chính xác các điểm thay đổi:

- Dễ mở rộng để triển khai thêm các pluggable.
- Kết nối lỏng hơn.
- Chi phí cho thay đổi thấp hơn.

Ví dụ minh họa

```
import java.io.*;
import java.nio.file.*;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.util.*; import java.util.stream.*;
```

```
class ItemID implements Comparable<ItemID> {
    int id;
```

```

    ItemID(int id) { this.id = id; }
    @Override public int compareTo(ItemID o) { return id - o.id; }
} class Money {
double amount;
Money(double amount) { this.amount = amount; }

Money add(Money money) {
    this.amount += Math.max(0, money.amount);
return this;
}

Money minus(Money money) {
    this.amount -= Math.max(0, money.amount);
return this;
}

Money multiple(int times) { return new Money(amount * times); }
@Override public String toString() { return String.format("%.1f", amount); }
}
interface IPayment {
    public boolean authorized();
}
abstract class Payment implements IPayment {
Money amount;
protected Payment(Money amount) { this.amount = amount; }
}
class CardPayment extends Payment {
String cardNo;
public CardPayment(double amount, String cardNo) {
super(new Money(amount));      this.cardNo = cardNo;
}

// Luhn Algorithm
@Override public boolean authorized() {
return IntStream.range(0, cardNo.length())
    .map(i -> (i % 2 != 0 ? 2 : 1) * (cardNo.charAt(i) - '0'))
    .map(i -> i > 9 ? i - 9 : i)
    .sum() % 10 == 0;
}
} class Product
{
    ItemID
itemID;
    String name;
    Money price;
    Product(String info) { // Line format: id|name|price
Scanner scanner = new Scanner(info);
scanner.useDelimiter("\\|");
    this.itemID = new ItemID(Integer.parseInt(scanner.next()));
this.name = scanner.next();
    this.price = new Money(Double.parseDouble(scanner.next()));
}
}
class ProductCatalog {
    Map<ItemID, Product> products;
ProductCatalog(String fileName) {
try {
    this.products = Files.lines(Paths.get(fileName))
        .map(Product::new)
        .collect(Collectors.toMap(p -> p.itemID, p -> p,
            (u, v) -> { throw new IllegalStateException("Duplicate: " + u); }, TreeMap::new));
} catch (IOException e) {
    System.out.println("[ERROR]: " + e.getMessage());
}
}

Product getProduct(ItemID itemID) { return products.get(itemID); }
void showCatalog() {
    System.out.println("--- Product List ---");
}

```

```

        products.entrySet().forEach(e -> System.out.printf("[%d] %s $%s%n",
            e.getKey().id, e.getValue().name, e.getValue().price));
    }
}
class OrderItem {
Product product;
Money price; int
quantity;
    public OrderItem(Product product, int quantity) {
this.product = product; this.price =
this.product.price; this.quantity = quantity;
    }

    Money subTotal() { return price.multiple(quantity); }
}
class Order {
    LocalDate date = LocalDate.now();
boolean isComplete = false;
List<OrderItem> items = new ArrayList<>();
Payment payment = null;

    void becomeComplete() {
payment.amount.minus(total());
isComplete = true;
    Store.saveToFile(this, "orders.txt");
    }
    public boolean isComplete() { return isComplete; }

    void makePayment(Payment payment) {
this.payment = payment;
    }

    Money total() {
        return items.stream().map(OrderItem::subTotal).reduce(new Money(0), Money::add);
    }
    void addOrderItem(Product p, int q) { items.add(new OrderItem(p, q)); }
@Override public String toString() {
    String m = String.format("Order date: %s%n",
date.format(DateTimeFormatter.ofPattern("dd/MM/yyyy")));      m += "---- Item list [" + items.size() +
"] ---\n";      return String.format("%sTotal: $%", items.stream()
.map(i ->
String.format("%-10s\t%d\t$%s\t%$s%n",
i.product.name, i.quantity, i.price, i.subTotal())))
.reduce(m, String::concat), total());
    }
}
class Register {
    ProductCatalog catalog;
Order currentOrder;
    Register(ProductCatalog catalog) {
        this.catalog = catalog;
catalog.showCatalog();
    } void endOrder(Payment
payment) {
currentOrder.makePayment(payment);
currentOrder.becomeComplete();
    } void makeNewOrder() {
currentOrder = new Order();
    } void enterItem(Product product, int
quantity) {
currentOrder.addOrderItem(product, quantity);
    }

    void showOrder() {
        System.out.println(currentOrder);
        System.out.println("Balance: " + currentOrder.payment.amount);
    }
}
class
Store {
    static void saveToFile(Object o, String fileName) {

```

```

        try (BufferedWriter out = Files.newBufferedWriter(Paths.get(fileName))) {
    out.write(o.toString());
} catch (IOException e) {
    System.out.println("[ERROR]: " + e.getMessage());
}
}

class Menu {
    void action(Register register,
Payment payment) {
        Scanner scanner = new
Scanner(System.in);
        register.makeNewOrder();
while (true) {
    System.out.print("Enter product ID: ");
    String id = scanner.nextLine();
    System.out.print("Enter quantity: ");
String quantity = scanner.nextLine();
    if (!id.matches("\\d+")) System.out.println("Product not found!");
else {
    Product p = register.catalog.getProduct(new ItemID(Integer.parseInt(id)));
if (p != null) {
        if (!quantity.matches("\\d+")) System.out.println("Quantity incorrect!");
else register.enterItem(p, Integer.parseInt(quantity));
    }
}
System.out.print("Next? ");
String option = scanner.nextLine();
if (option.matches("[Nn]")) {
register.endOrder(payment);
        Store.saveToFile(register.currentOrder, "orders.txt");
break;
}
}
register.showOrder();
}
}

public class Main {
    public static void main(String[] args) {
        new Menu().action(new Register(new ProductCatalog("products.txt")),
new CardPayment(100, "194774915"));
    }
}

```

Creational Design Patterns

Abstract Factory

Create object families

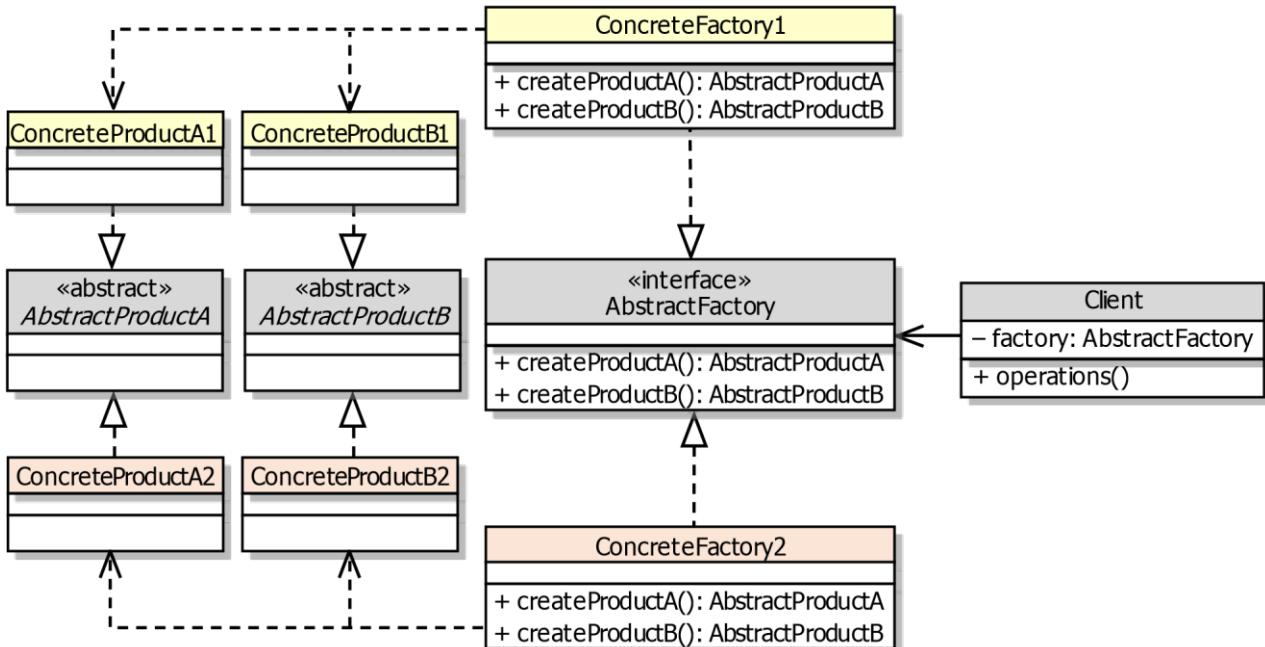
Mẫu thiết kế Abstract Factory cung cấp một giao diện dùng để tạo ra một họ các đối tượng có quan hệ (hoặc phụ thuộc) với nhau mà không cần chỉ định rõ lớp cụ thể của chúng. Họ đối tượng (object family) không mang ý nghĩa chung cùng cây thừa kế, mà chúng là một dòng các đối tượng có quan hệ (hoặc phụ thuộc) với nhau trong một ngữ cảnh. Một số tài liệu gọi "họ" là kit, set.

Ví dụ về "họ" đối tượng theo ngữ cảnh sử dụng:

- Ngữ cảnh là theme sử dụng: window và scrollbar có theme màu vàng thuộc họ đối tượng YellowThemeWidget; họ này được tạo ra từ factory cụ thể, YellowThemeWidgetFactory.
- Ngữ cảnh là hệ nền chạy chương trình: window và scrollbar chạy trên Unix thuộc họ đối tượng XWindowsComponent; họ này được tạo ra từ factory cụ thể, XWindowsFactory.

Tập hợp các phương thức dùng tạo đối tượng được đóng gói trong một đối tượng tạo, gọi là đối tượng factory.

1. Cài đặt



- `AbstractFactory`: khai báo giao diện chứa một tập hợp các phương thức tạo các loại đối tượng.
- `ConcreteFactory`: cài đặt cụ thể các phương thức tạo đối tượng để tạo ra họ đối tượng. Ví dụ, "họ" màu vàng và "họ" màu hồng.
- `AbstractProduct`: khai báo giao diện chung cho loại đối tượng sẽ được tạo. Ví dụ, window và scrollbar là hai loại đối tượng.
- `ConcreteProduct`: các đối tượng thực sự được tạo ra bởi factory. Ví dụ, window "họ" màu vàng và scrollbar "họ" màu hồng.
- `Client`: sử dụng các đối tượng được tạo ra thông qua giao diện. Do Client không xác định được ngữ cảnh tạo đối tượng nên phải dùng `Abstract Factory`.

Các đặc tính của mẫu thiết kế này:

- Được gọi là "trùu tượng" (abstract) vì Client truy cập đến việc tạo (factory) sản phẩm một cách trùu tượng thông qua interface (`AbstractFactory`) thay vì làm việc trực tiếp với lớp factory cụ thể.
 - Sử dụng tổng hợp (composition) thay cho thừa kế, Client "có một" (HAS-A) tham chiếu kiểu `AbstractFactory`. Tham chiếu này có thể chỉ đến một trong các đối tượng thuộc lớp factory cụ thể bất kỳ, thừa kế/cài đặt `AbstractFactory`, mỗi lớp factory cụ thể có khả năng tạo ra một họ các sản phẩm có liên quan.
 - Khi factory tạo ra các sản phẩm cụ thể, Client sẽ truy cập đến các sản phẩm này thông qua interface (`AbstractProduct`), điều này cho phép Client làm việc như nhau với các sản phẩm được tạo ra cho dù chúng được tạo từ lớp factory cụ thể nào. **Các bước thực hiện**
 - Trước tiên, cần hiểu rõ khái niệm "họ" đối tượng, phân biệt các "họ" đối tượng của các `Abstract Product` cho dù chúng cùng loại.
 - Tạo `AbstractFactory` là lớp abstract hoặc interface, định nghĩa các phương thức abstract để tạo các loại `AbstractProduct`.
 - Cung cấp các `ConcreteFactory`, chứa các phương thức tạo ra các `ConcreteProduct` thuộc các họ `AbstractProduct` khác nhau nhưng cùng "họ" với nhau.
- ```

// AbstractFactory
abstract class AbstractFactory {
 + createProductA(): AbstractProductA
 + createProductB(): AbstractProductB
}

// ConcreteFactory
class ConcreteFactory1 : AbstractFactory {
 + createProductA(): AbstractProductA {
 return new ConcreteProductA1();
 }
 + createProductB(): AbstractProductB {
 return new ConcreteProductB1();
 }
}

class ConcreteFactory2 : AbstractFactory {
 + createProductA(): AbstractProductA {
 return new ConcreteProductA2();
 }
 + createProductB(): AbstractProductB {
 return new ConcreteProductB2();
 }
}

// AbstractProduct
abstract class AbstractProductA {
 <<abstract>>
}

abstract class AbstractProductB {
 <<abstract>>
}

// ConcreteProduct
class ConcreteProductA1 : AbstractProductA {
 ...
}

class ConcreteProductB1 : AbstractProductB {
 ...
}

class ConcreteProductA2 : AbstractProductA {
 ...
}

class ConcreteProductB2 : AbstractProductB {
 ...
}

```

```

// ConcreteProduct
class YellowThemeWindow extends Window {
 @Override public void draw() {
 System.out.println(getClass().getName());
 }
}
class PinkThemeWindow extends Window {
 @Override public void draw() {
 System.out.println(getClass().getName());
 }
}
class YellowThemeScrollbar extends Scrollbar {
 @Override public void paint() {
 System.out.println(getClass().getName());
 }
}
class PinkThemeScrollbar extends Scrollbar {
 @Override public void paint() {
 System.out.println(getClass().getName());
 }
}

```

```

// AbstractFactory interface
WidgetFactory {
 Scrollbar
 createScrollbar();
 Window createWindow();
}

// ConcreteFactory
class YellowThemeWidgetFactory implements WidgetFactory {
 @Override public Scrollbar createScrollbar() {
 return new YellowThemeScrollbar();
 }

 @Override public Window createWindow() {
 return new YellowThemeWindow();
 }
}
class PinkThemeWidgetFactory implements WidgetFactory {
@Override public Scrollbar createScrollbar() {
 return new PinkThemeScrollbar();
}

@Override public Window createWindow() {
 return new PinkThemeWindow();
}
}

public class Client {
 private static void initGUI(WidgetFactory factory) {
 factory.createScrollbar().paint();
 factory.createWindow().draw();
 }
 public static void main(String[] args) {
 System.out.println("--- Abstract Factory Pattern ---");
 initGUI(new PinkThemeWidgetFactory());
 }
}

```

Họ đối tượng ở đây là các widget (window, scrollbar) thiết kế đồ họa theo cùng một chủ đề (theme). Bằng cách thay đổi factory tạo họ đối tượng, bạn có thể chuyển sang sử dụng từ họ đối tượng này sang họ đối tượng khác một cách dễ dàng.

## 2. Liên quan

- Factory Method: Abstract Factory thường được cài đặt như một factory các Factory Method.
- Prototype: lớp AbstractFactory cũng được cài đặt bằng Prototype.
- Singleton: lớp ConcreteFactory thường là một Singleton.

## 3. Java API

Nhiều lớp của JAXP (DocumentBuilderFactory, SAXParserFactory, TransformFactory, XPathFactory, ...) là AbstractFactory, chúng có phương thức tạo trả về một factory, factory này được dùng để tạo các kiểu abstract/interface khác.

Lớp java.awt.Toolkit là một Abstract Factory được dùng để tạo các đối tượng làm việc với hệ thống cửa sổ thuộc các hệ nền khác nhau.

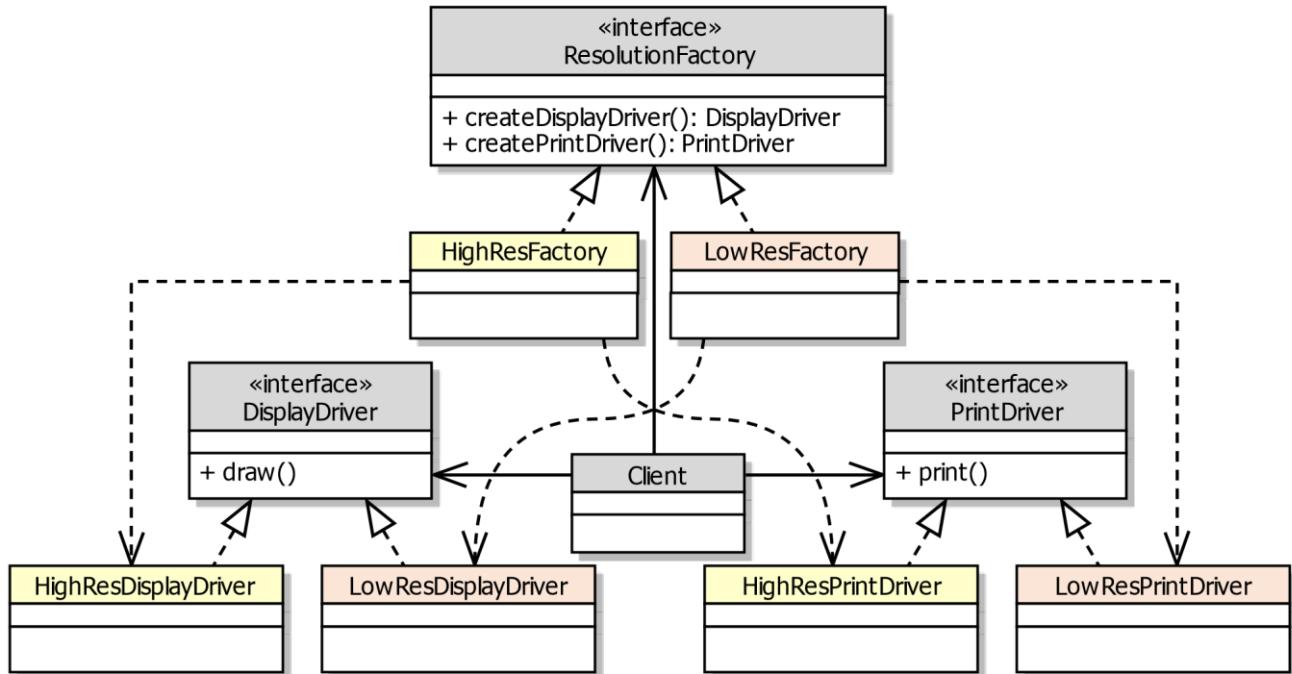
Trong JavaEE, các mẫu thiết kế Data Access Object (DAO, lớp Integration) và Transfer Object Assembler (lớp Business) cũng áp dụng mẫu thiết kế Abstract Factory.

## 4. Sử dụng Ta muốn:

- Một hệ thống độc lập với cách mà các sản phẩm của nó được tạo ra, được tích hợp và được thể hiện.
- Một hệ thống được cấu hình để tạo ra một hoặc nhiều họ sản phẩm, hoạt động trong nhiều ngữ cảnh.
- Yêu cầu bắt buộc các sản phẩm trong một họ phải làm việc với nhau.
- Cung cấp một thư viện lớp các sản phẩm, nhưng chỉ muốn đề cập đến phần giao diện, chưa chú ý đến phần cài đặt.

## 5. Bài tập

- a) Device driver cho màn hình (Display) và máy in (Printer) trong thuộc hai họ driver: driver có độ phân giải thấp (low-resolution) và driver có độ phân giải cao (high-resolution). Bạn hãy áp dụng mẫu thiết kế Abstract Factory để tạo ra các họ driver thích hợp.

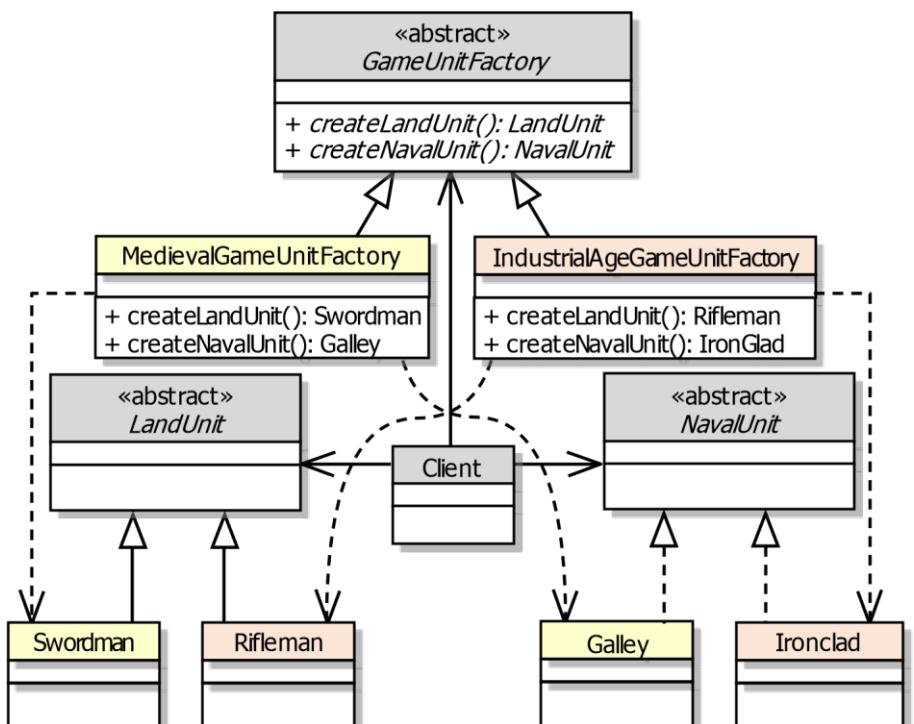


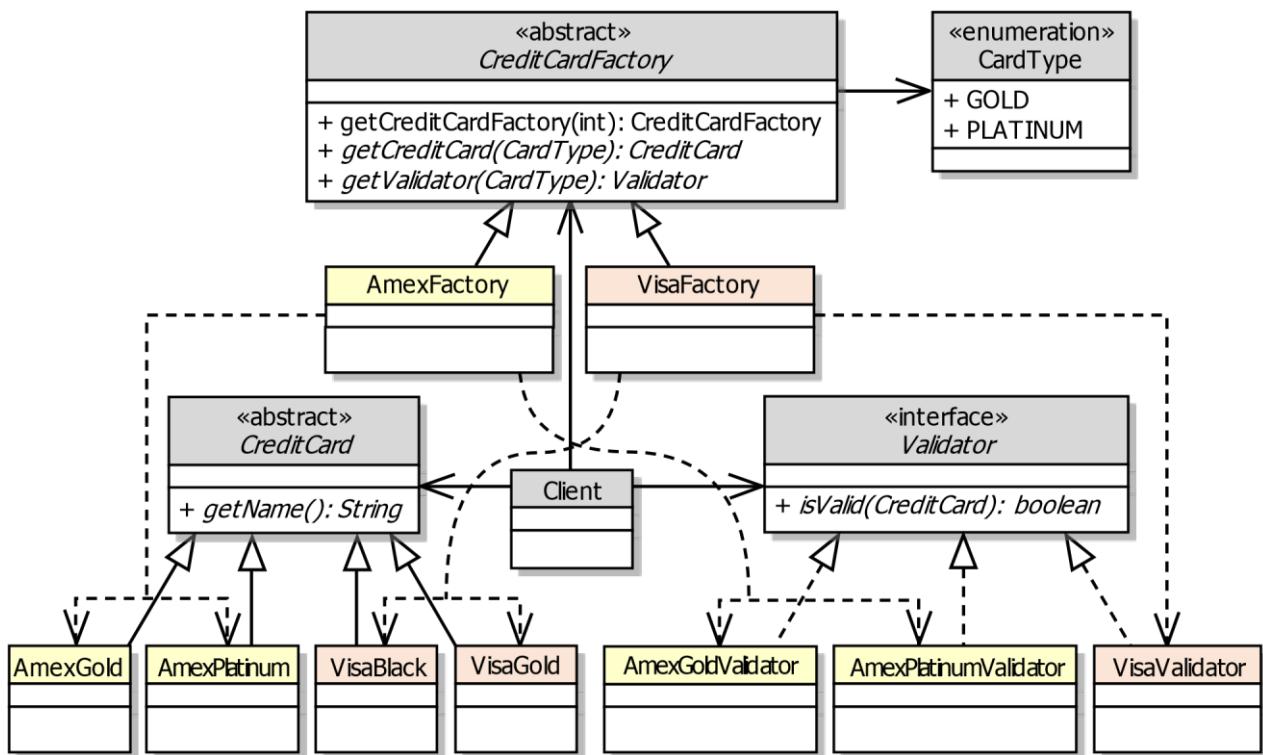
b) Trong một game chiến thuật, mỗi thời đại có một họ các đơn vị game. Đơn vị bộ binh (LandUnit) có 2 loại: Swordman (thời trung cổ) và Rifleman (kỷ nguyên công nghiệp). Đơn vị hải quân (NavalUnit) cũng có 2 loại: Galley (thời trung cổ) và IronGlad (kỷ nguyên công nghiệp). Chúng được tạo từ hai factory: dùng tạo các đơn vị thời trung cổ (MedievalGameUnitFactory) và dùng tạo các đơn vị kỷ nguyên công nghiệp (IndustrialAgeGameUnitFactory).

Bạn hãy áp dụng mẫu thiết kế Abstract Factory để tạo ra các họ đơn vị game cho từng ngã cảnh thời đại.

c) CreditCard có 2 loại: AmexCreditCard (gồm AmexGold và AmexPlatinum) và VisaCreditCard (VisaBlack và VisaGold). Hai loại AmexCreditCard được xác minh bằng các Validator AmexGoldValidator và AmexPlatinumValidator, trong lúc hai loại VisaCreditCard đều được xác minh bằng một Validator chung (VisaValidator). Loại CreditCard được tạo tùy theo creditScore

(creditScore > 500 cho AmexCreditCard và ngược lại cho VisaCreditCard) và CardType. Bạn hãy áp dụng mẫu thiết kế Abstract Factory để tạo ra các họ CreditCard và áp dụng loại Validator thích hợp.





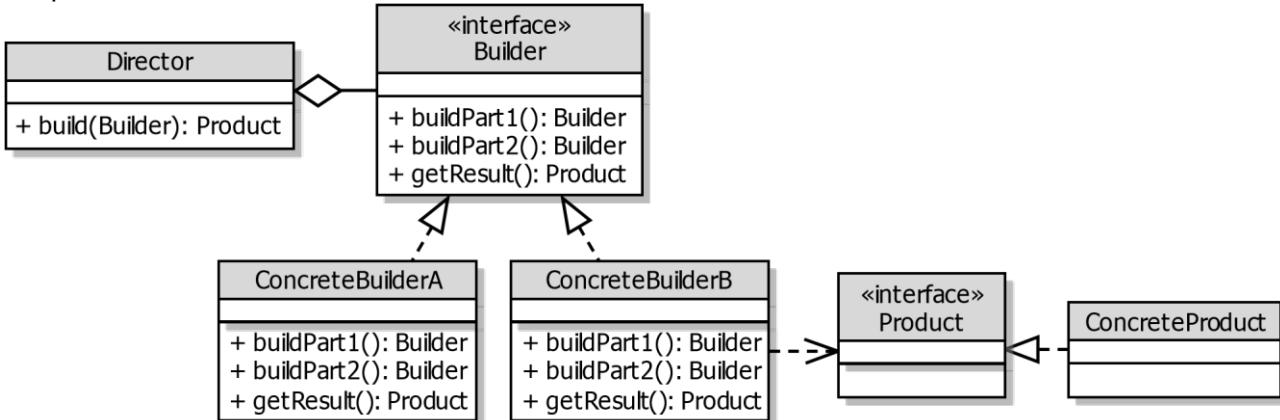
## Builder

Building complex objects

Mẫu thiết kế Builder tách việc khởi tạo nhiều bước của một đối tượng phức tạp (Product) thành tập hợp các thao tác khởi tạo từng phần của đối tượng đó. Lớp Director quyết định các bước khởi tạo một đối tượng phức tạp từ nhiều thành phần, trong lúc các lớp Builder thực sự xây dựng các thành phần đó.

Sự tách biệt trách nhiệm này cho phép cùng một quá trình khởi tạo có thể tạo ra các thể hiện khác nhau của đối tượng phức tạp. Ngoài ra, đối tượng có thể khởi tạo từng phần trong trường hợp chưa có đầy đủ dữ liệu cần cho việc khởi tạo.

### 1. Cài đặt



- Builder: cung cấp giao diện để Director tạo ra các thành phần của đối tượng phức tạp Product.
- ConcreteBuilder: các cài đặt khác nhau cho các phương thức tạo ra các thành phần của một Product cụ thể.
- Director: Client gọi Director để ráp nối và tạo ra Product từ các thành phần do Builder cung cấp. Director được dùng để quyết định số lượng và thứ tự các bước xây dựng Product, nó gọi các Builder khi cần các thành phần để hình thành nên Product.
- Product: đối tượng có quá trình khởi tạo phức tạp.
- Client phải liên kết một trong các Builder với Director, thường bằng cách truyền tham số cho constructor của Director. Director sẽ dùng Builder này cho việc xây dựng tương lai. Cách khác là truyền Builder cho phương thức build() của Director. **Các bước thực hiện**
- Bắt đầu bằng việc tạo interface Builder:
  - + Nhận diện các "phần" của Product và cung cấp các phương thức buildPart() để tạo ra các phần này.
  - + Các phương thức buildPart() thường viết theo fluent syntax, tạo thành dây chuyền phương thức, để "lắp ráp" các "phần" Product liên tiếp với nhau.
  - + Cung cấp một phương thức getResult (hoặc build()) để tạo ra Product cuối đầy đủ.
  - + Builder có thể được tạo như một lớp nội static của Product.
- Director hiếm khi cài đặt như một lớp riêng mà thường tích hợp trong Client.

```
// Builder
interface QueryBuilder {
 QueryBuilder setFrom(String from);
 QueryBuilder setWhere(String where);
 Query getQuery();
}

// ConcreteBuilder class SQLQueryBuilder
implements QueryBuilder { protected Query
query = new SQLQuery();
 @Override public QueryBuilder setFrom(String from) {
query.add(from); return this;
}

 @Override public QueryBuilder setWhere(String where) {
query.add(where); return this;
}

 @Override public Query getQuery() {
return query;
}
}

// Product interface
Query {
 void add(String s); void
execute();
}
```

```

}

// ConcreteProduct
class SQLQuery implements Query {
 StringBuilder query = new StringBuilder();
 @Override public void add(String part) {
 query.append(" ").append(part);
 }

 @Override public void execute() {
 System.out.printf("Execute \'%s\'%n", query.toString().trim());
 }
}

// Director
class QueryBuildDirector {
 public Query build(QueryBuilder builder, String from, String where) {
 return builder.setFrom(from).setWhere(where).getQuery();
 }
}

public class Client {
 public static void main(String[] args) {
 System.out.println("--- Builder Pattern ---");
 QueryBuildDirector director = new QueryBuildDirector();
 QueryBuilder builder = new SQLQueryBuilder();
 Query query = director.build(builder, "FROM Student", "WHERE code=1234");
 query.execute();
 }
}

```

Director quyết định cách dùng các thành phần xây dựng "cấu hình" đối tượng Product, trong lúc loại ConcreteBuilder được lựa chọn mới thực sự tạo ra các thành phần này. Bạn có thể tạo thêm nhiều loại ConcreteBuilder khác, ví dụ NoSQLQueryBuilder. Do Builder thường viết theo fluent syntax, nên việc thừa kế Builder có thể phát sinh vấn đề, do this được trả về tham chiếu đến Builder lớp cha, không thể gọi tiếp phương thức của lớp con nữa. Vấn đề này được giải quyết bằng cách dùng kiểu generic ràng buộc đệ quy (recursive bounded). Xem ví dụ sau:

```

class Person {
 String name, position;
 @Override public String toString() {
 return String.format("%s:%s", name, position);
 }
}

class PersonBuilder<SELF extends PersonBuilder<SELF>> {
 protected Person person = new Person(); public SELF
 withName(String name) { person.name = name;
 return self();
 }
 protected SELF self() { return (SELF) this; }
 public Person build() { return person; }
}

class EmployeeBuilder extends PersonBuilder<EmployeeBuilder> {
 public EmployeeBuilder worksAs(String position) {
 person.position = position; return self();
 }

 @Override protected EmployeeBuilder self() {
 return this;
 }
}

class Client {
 public static void main(String[] args) {
 EmployeeBuilder builder = new EmployeeBuilder()
 .withName("Karl Marx")
 .worksAs("Philosopher");
 System.out.println(builder.build());
 }
}

```

## 2. Liên quan

- Abstract Factory: Builder tập trung vào các bước xây dựng đối tượng phức tạp, trong khi Abstract Factory tập trung vào việc tạo một họ các đối tượng chỉ trong một bước, giữ vai trò trung tâm khi khởi tạo đối tượng. Director có thể dùng Abstract Factory để tạo các đối tượng Builder.
- Template Method: Builder thường được cài đặt bằng cách dùng Template Method.
- Composite: Builder thường được dùng để xây dựng các đối tượng Composite.
- Facade: nếu quá trình xây dựng đối tượng chia thành nhiều bộ phận phức tạp, có thể tạo một Facade Builder, xem các Builder cho các bộ phận như các subsystem. Xem ví dụ sau:

```

class Person {
 // address
 String streetAddress, postcode, city;
 // employment
 String companyName, position;
 int annualIncome;
 @Override public String toString() {
 return String.format("%s:%d", streetAddress, annualIncome);
 }
}

// Facade Builder class
PersonBuilder {
 protected Person person = new Person();
 PersonAddressBuilder lives() {
 return new PersonAddressBuilder(person);
 }

 PersonJobBuilder works() {
 return new PersonJobBuilder(person);
 }

 Person build() { return person; }
}

// Subsystems
class PersonAddressBuilder extends PersonBuilder {
 PersonAddressBuilder(Person person) {
 this.person = person;
 }

 PersonAddressBuilder at(String streetAddress) {
 person.streetAddress = streetAddress; return
 this;
 }

 PersonAddressBuilder withPostcode(String postcode) {
 person.postcode = postcode; return this;
 }

 PersonAddressBuilder in(String city) {
 person.city = city; return this;
 }
}
class PersonJobBuilder extends PersonBuilder {
 PersonJobBuilder(Person person) {
 this.person = person;
 }

 PersonJobBuilder at(String companyName) {
 person.companyName = companyName;
 return this;
 }

 PersonJobBuilder asA(String position) {
 person.position = position; return
 this;
 }

 PersonJobBuilder earning(int annualIncome) {

```

```

 person.annualIncome = annualIncome;
 return this;
}
}
class Client {
 public static void main(String[] args) {
 System.out.println("--- Façade Builder Pattern ---");
 Person trump = new PersonBuilder()
 .lives()
 .at("The White House")
 .in("1600 Pennsylvania Avenue NW")
 .withPostcode("DC 20500")
 .works()
 .at("United States Government")
 .asA("President")
 .earning(150000)
 .build();
 System.out.println(trump);
 }
}

```

### 3. Java API

Như một áp dụng của mẫu thiết kế Builder, đoạn code sau dùng SAXParser (Director) của JAXP để phân tích tập tin XML với lớp xử lý MySAXHandler do ta viết (ConcreteBuilder) dẫn xuất từ org.xml.sax.helpers.DefaultHandler (Builder).

```

SAXParser parser = factory.newSAXParser(); // Director
MySAXHandler handler = new MySAXHandler(); // ConcreteBuilder
parser.parse("books.xml", handler); // thiết lập Builder cho Director và xử lý
ArrayList<Book> books = handler.getBooks(); // lấy Product được tạo

```

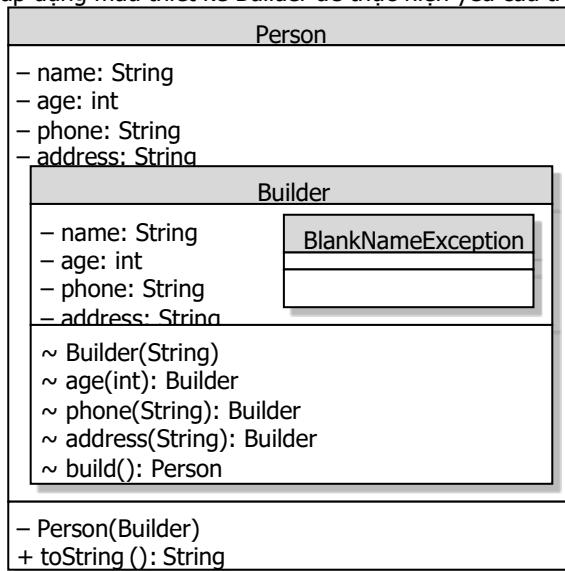
Lớp java.lang.StringBuilder và lớp java.lang.StringBuffer cũng là các Builder, có phương thức tạo (append) trả về chính thể hiện của nó, giúp xây dựng đối tượng bằng cách khởi tạo từng phần. Lớp java.util.Calendar.Builder (Java 8) cũng là một Builder.

### 4. Sử dụng Ta muốn:

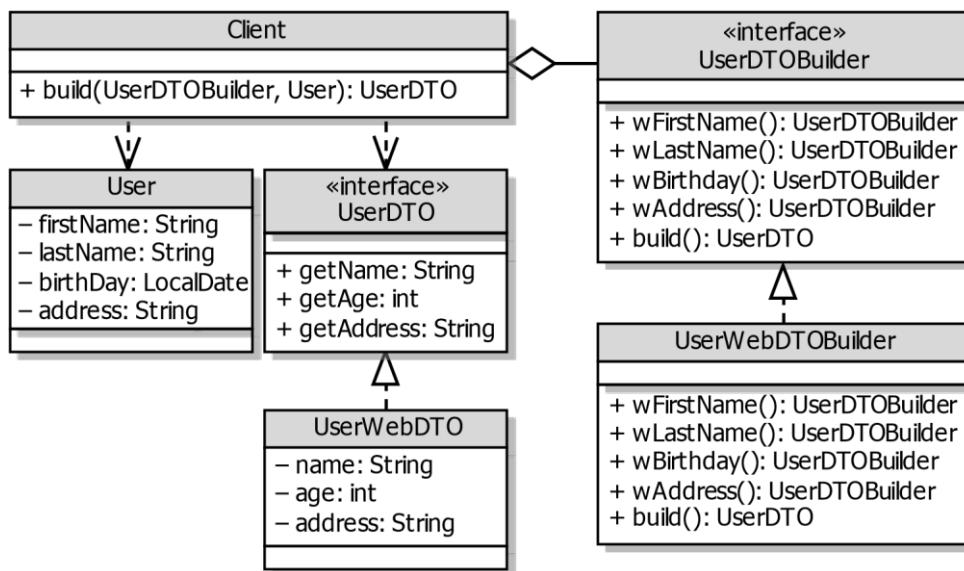
- Quá trình xây dựng nên đối tượng độc lập với các thành phần tạo nên đối tượng.
- Dễ dàng thêm các cài đặt mới, tức các Builder mới.
- Kiểm soát linh hoạt hơn quá trình xây dựng đối tượng: vấn đề nhiều constructor và thứ tự tạo thành các phần của đối tượng.
- Giải quyết vấn đề constructor cần nhiều thông tin để khởi tạo, constructor có quá nhiều tham số.

### 5. Bài tập

- a) Thông tin đăng ký của một cá nhân có phần bắt buộc (name) và phần tùy chọn (age, phone, address). Phần tùy chọn có thể được bổ sung sau. Để tránh viết nhiều constructor với số lượng tham số tăng dần (telescoping constructors) và có thể xây dựng đối tượng từng phần, hãy áp dụng mẫu thiết kế Builder để thực hiện yêu cầu trên.



- b) Lớp Client (đóng vai trò Director) cần trích lấy thông tin từ lớp User để tạo nên thực thể UserDTO (Data Transfer Object) cần sử dụng. Có nhiều cách trích thông tin nên yêu cầu nhiều Builder (UserWebDTOBuilder, UserRESTDTOBuilder, ...) để tạo ra loại UserDTO tương ứng. Hãy dùng mẫu thiết kế Builder thực hiện yêu cầu trên.  
Trong trường hợp chỉ cần một loại UserDTO, nghĩa là chỉ cần một loại Builder, không cần các interface UserDTO và UserDTOBuilder nữa. Hãy tổ chức lớp Builder như lớp nội của UserDTO, thường gọi là "nested builder".



## Factory Method

Delegate object creation

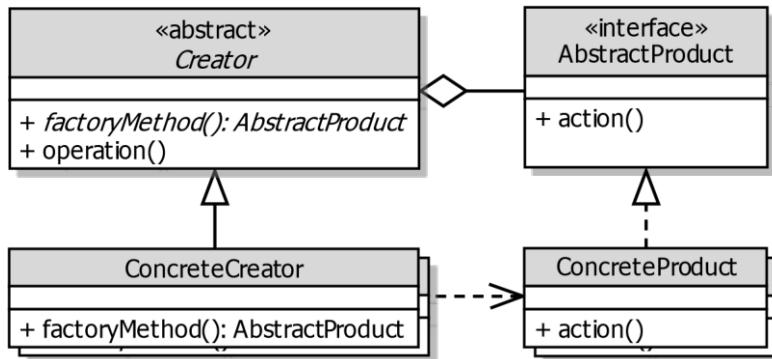
Mẫu thiết kế Factory Method định nghĩa một giao diện trừu tượng dùng tạo một đối tượng, nhưng nó ủy nhiệm cho lớp dẫn xuất quyết định đối tượng thuộc lớp cụ thể nào sẽ được tạo. Do đặc điểm này, Factory Method còn gọi là Virtual Constructor. Đối tượng tạo Creator, là một đối tượng trừu tượng chứa phương thức (factory method) dùng tạo ra một đối tượng trừu tượng khác, gọi là AbstractProduct. AbstractProduct có nhiều kiểu dẫn xuất khác nhau (ConcreteProduct). Vì Creator không biết kiểu

ConcreteProduct cụ thể nào được tạo ra nên nó trì hoãn nhiệm vụ tạo đối tượng cho lớp con của nó thực hiện.

Việc trì hoãn tạo đối tượng có ý nghĩa, vì khi dùng toán tử new để tạo đối tượng, phải xác định ngay kiểu của đối tượng được tạo. Khi dùng mẫu thiết kế Factory Method, ta tạo đối tượng nhưng ủy nhiệm cho lớp khác quyết định kiểu của đối tượng đó.

Trong OOP, thừa kế và viết lại phương thức là để thay đổi hành vi của lớp. Với mẫu thiết kế Factory Method, hành vi được thay đổi là tạo đối tượng, thừa kế Creator và viết lại phương thức factory để thay đổi cách tạo đối tượng.

### 1. Cài đặt



- Creator: lớp trừu tượng chứa phương thức trừu tượng factoryMethod() dùng tạo ra AbstractProduct. Phương thức này cũng có thể được tham số hóa để tạo ra nhiều kiểu AbstractProduct hoặc cài đặt sẵn để tạo ra đối tượng mặc định.

Creator cũng có thể chứa phương thức operation() có sử dụng AbstractProduct được tạo ra.

- ConcreteCreator: lớp con của Creator, cài đặt cụ thể phương thức factoryMethod() để sinh ra loại ConcreteProduct được chọn.

- AbstractProduct: giao diện của đối tượng được tạo ra từ phương thức factoryMethod(), có nhiều kiểu dẫn xuất.

- ConcreteProduct: cài đặt giao diện AbstractProduct, là "sản phẩm" cụ thể được tạo ra từ phương thức factoryMethod() của ConcreteCreator. **Các bước thực hiện**

- Tạo lớp Creator, có thể là lớp cụ thể nếu phương thức factoryMethod() cài đặt sẵn để tạo đối tượng mặc định.

- Nhánh Creator ánh xạ nhánh AbstractProduct, phương thức factoryMethod() của mỗi ConcreteCreator tạo ra ConcreteProduct tương ứng.

```
import java.util.Arrays;
abstract class Creator { public static
final int INSERTION_SORT = 0; public static
final int SELECTION_SORT = 1;

 public abstract SortAlgorithm createAlgorithms(int type);

 public void sortArray(int type, Comparable... array) {
createAlgorithms(type).sort(array);
 }
}

// ConcreteCreator
class SortAlgorithmCreator extends Creator {
 @Override public SortAlgorithm createAlgorithms(int type) {
switch (type) { case INSERTION_SORT: return new
InsertionSort(); case SELECTION_SORT: return new
SelectionSort(); default: return null;
 }
}
}

// AbstractProduct
interface SortAlgorithm<T extends Comparable<? super T>> {
void sort(T... array);
}

// ConcreteProduct
class InsertionSort<T extends Comparable<? super T>> implements SortAlgorithm<T> {
```

```

@Override public void sort(Comparable... array) {
 Comparable temp;
 int i, j;
 for (i = 1; i < array.length; i++) {
 temp = array[i];
 for (j = i; j > 0 && temp.compareTo(array[j - 1]) < 0; j--) array[j] = array[j - 1];
 array[j] = temp;
 }
}
class SelectionSort<T extends Comparable<? super T>> implements SortAlgorithm<T> {
 @Override public void sort(Comparable... array) {
 int i, j, least;
 for (i = 0; i < array.length - 1; ++i) {
 for (j = i + 1, least = i; j < array.length; ++j)
 if (array[j].compareTo(array[least]) < 0) least = j;
 if (least != i) {
 Comparable temp = array[least];
 array[least] = array[i];
 array[i] = temp;
 }
 }
 }
}
public class Client { public static void
main(String[] args) { System.out.println("----"
Factory Method ---"); int[] a = { 1, 8, 3, 6,
5, 4, 7, 2, 9 };
 Integer[] o = Arrays.stream(a).boxed().toArray(Integer[]::new);
 Creator creator = new SortAlgorithmCreator();
 creator.sortArray(Creator.SELECTION_SORT, o);
 System.out.println(Arrays.toString(o));
 Arrays.stream(o).forEach(System.out::println);
 }
}

```

Client gọi phương thức sortArray() được tham số hóa để chỉ định thuật toán sắp xếp. Phương thức này gọi phương thức factory đa hình createAlgorithm() của Creator để tạo đối tượng SortAlgorithm yêu cầu. Lớp con của Creator, SortAlgorithmCreator, mới là lớp thực sự tạo ra đối tượng SortAlgorithm cụ thể: InsertionSort hoặc SelectionSort.

## 2. Liên quan

- Template Method: Factory Method đôi khi được cài đặt như một bước của Template Method.
- Prototype: các Prototype không cần dẫn xuất Creator, chúng cần một tác vụ khởi tạo trong lớp Product, Creator dùng tác vụ khởi tạo này để khởi tạo đối tượng.
- Abstract Factory: thường dễ nhầm lẫn giữa Abstract Factory và Factory Method. Khác biệt chủ yếu giữa chúng:
  - + Factory Method: quan tâm đến phương thức factory, tức phương thức sinh đối tượng. Ý tưởng chính là khai báo phương thức factory ở giao diện trừu tượng và cài đặt nó ở lớp dẫn xuất; để lớp dẫn xuất quyết định lớp đối tượng cụ thể được tạo từ phương thức factory.
  - + Abstract Factory: quan tâm đến đối tượng factory. Ý tưởng chính là tạo một lớp Factory đóng gói nhiều phương thức tạo (creational method, phân biệt với phương thức factory); Client ủy nhiệm cho phương thức tạo cụ thể tạo đối tượng. Các phương thức tạo trong Abstract Factory thường được cài đặt với Factory Method.
- Iterator: thường được tạo từ một Factory Method.

## 3. Java API

Giao diện Collection của Java có khai báo một phương thức factory tạo ra một Iterator, dùng để duyệt các phần tử của nó:

Iterator iterator();

Mỗi lớp dẫn xuất từ Collection cài đặt phương thức iterator() theo cách khác nhau, do đối tượng Iterator của chúng có cơ chế hoạt động hoàn toàn khác nhau. Khả năng đa hình cho phép lời gọi: Iterator iterator = collection.iterator(); gọi phương thức iterator() của lớp dẫn xuất cụ thể để tạo ra Iterator cụ thể phù hợp với loại collection dẫn xuất đó. Điều này linh động hơn việc tạo Iterator từ constructor.

Trong trường hợp này: Creator là Collection, ConcreteCreator là các lớp dẫn xuất của Collection (LinkedList, ArrayList, Queue), AbstractProduct là Iterator, ConcreteProduct là lớp dẫn xuất từ Iterator (thường vô danh), phương thức factoryMethod() là phương thức iterator().

Ngoài ra, còn nhiều ví dụ về phương thức factory trong Java API: java.util.Calendar.getInstance(),  
 java.nio.charset.Charset.forName(), java.util.EnumSet.of(),  
 java.lang.Class.forName(), javax.xml.bind.JAXBContext.createMarshaller(),  
 java.net.URLStreamHandlerFactory.createURLStreamHandler(), ...

## 4. Sử dụng Ta có:

- Biết rõ khi nào đối tượng được tạo (trong thời gian chạy) nhưng chưa biết rõ thông tin về đối tượng được tạo. Ví dụ kiểu đối tượng được tạo phụ thuộc vào tham số được nhập vào, phụ thuộc hành vi chỉ định.

- Việc tạo đối tượng đang trở nên phức tạp. Ví dụ việc khởi tạo đối tượng phụ thuộc vào nhiều đối tượng khác, phải xây dựng thêm các đối tượng liên quan.

Ta muốn:

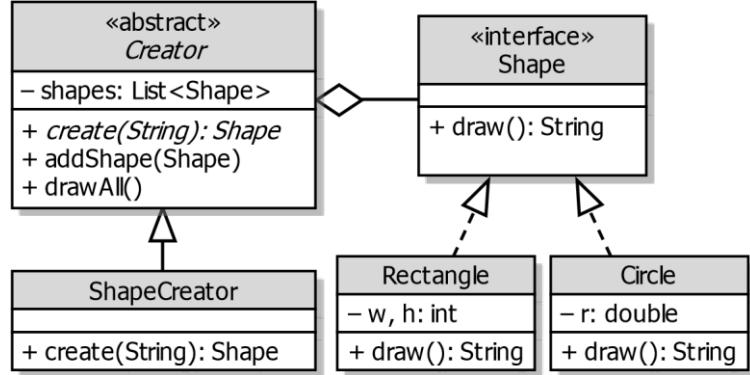
- Linh hoạt trong việc tạo đối tượng. Hệ thống các lớp được tạo có thể thay đổi tự động. Xem xét sử dụng:
- Abstract Factory, Prototype, hoặc Builder. Chúng phức tạp hơn, nhưng linh hoạt hơn.
- Prototype, lưu tập các đối tượng được nhân bản từ Abstract Factory.

## 5. Bài tập

a) Chương trình vẽ dùng lớp Creator để sinh và quản lý các hình vẽ (Shape). Có hai loại hình vẽ: Rectangle (với thông tin là chiều dài w và chiều rộng h) và Circle (với thông tin là bán kính r). Chúng dùng phương thức draw() để vẽ. Lớp Creator có phương thức create() tạo loại Shape từ chuỗi chứa thông tin hình vẽ có định dạng:

#rectan w, h      Ví dụ, #rectan 8, 5  
#circle r            Ví dụ, #circle 3.7

Lớp Creator cũng có phương thức drawAll() dùng vẽ tất cả hình nó lưu trữ.



b) Cho lớp PaymentService, constructor của nó tạo một thể hiện của lớp FinancialTrustCCP, là một dịch vụ xử lý thẻ (Credit Card Processing) do bên thứ ba cung cấp. `public class PaymentService {`

```

private final static String recipientID = "123-456-789";
private FinancialTrustCCP ccp; // dịch vụ xử lý thẻ của bên thứ ba
public PaymentService() { ccp = new FinancialTrustCCP();
}
public void pay(String senderID, String amount) { // chuyển tiền từ sender đến recipient
boolean approved = ccp.post(senderID, recipientID, amount); if (approved) // ... else //
...
}

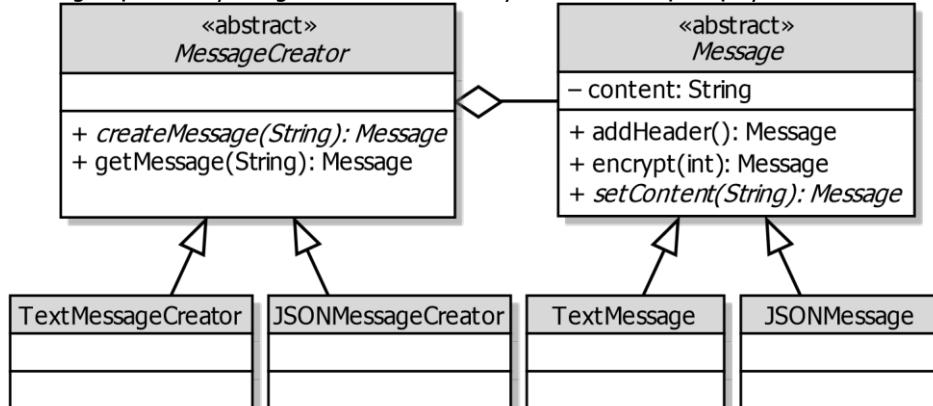
```

Đối tượng được tạo FinancialTrustCCP liên kết quá chặt với đối tượng dùng nó là PaymentService nên vi phạm nguyên tắc OCP. Không thể nào mở rộng giải pháp hiện tại, ví dụ làm việc với một dịch vụ xử lý thẻ khác, hoặc kiểm thử với mock object, mà không phải thay đổi code.

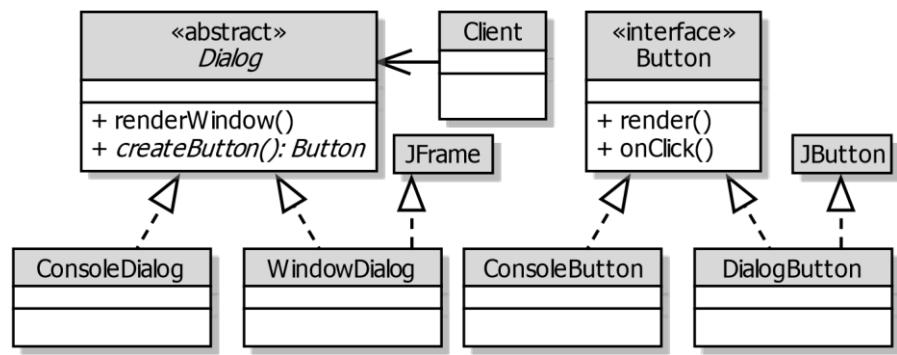
Bạn hãy giải quyết vấn đề này bằng cách áp dụng mẫu thiết kế Factory Method. Ý tưởng là thay thế việc tạo trực tiếp đối tượng FinancialTrustCCP bằng một Factory Method như sau: CCPService createCCPService();

CCPService do phương thức trả về là interface mà FinancialTrustCCP, các dịch vụ xử lý thẻ khác hoặc mock object, phải cài đặt.

c) Chương trình dùng hai loại Message có định dạng ban đầu khác nhau: TextMessage và JSONMessage. Từ định dạng ban đầu, thông điệp được mã hóa (Caesar) và gắn thêm header để tạo ra thông điệp cuối. Lớp MessageCreator dẫn xuất ra các Factory sinh ra loại Message cụ thể. Hãy dùng mẫu thiết kế Factory Method để thực hiện yêu cầu.



d) Dialog là đối tượng tạo các Button, các loại Dialog khác nhau yêu cầu kiểu Button phù hợp với chúng. Vì vậy, bạn dẫn xuất các lớp con của Dialog và cài đặt các phương thức factory phù hợp để tạo loại Button tương ứng. Gọi phương thức onClick() trên từng loại Button để thấy chúng hoạt động đúng.



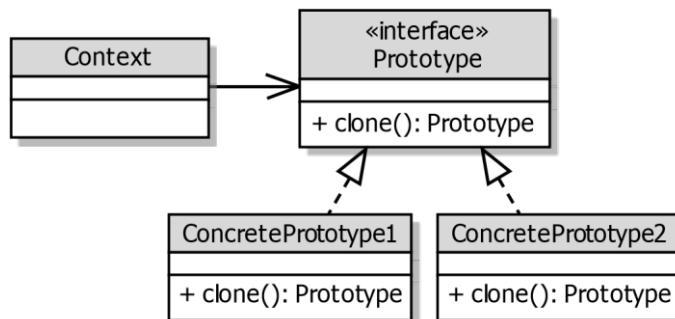
## Prototype

Create object on prototype

Mẫu thiết kế Prototype dùng một thể hiện nguyên mẫu (prototypical instance) để chỉ định loại đối tượng sẽ được tạo. Nói cách khác, mẫu thiết kế Prototype tạo các đối tượng mới bằng cách nhân bản một thể hiện nguyên mẫu là chính nó.

Nếu chi phí khởi tạo đối tượng cao, ví dụ khởi tạo có truy xuất cơ sở dữ liệu, ta không tạo ra các đối tượng trực tiếp từ constructor (dùng toán tử new) mà sao chép chúng từ đối tượng nguyên mẫu đang tồn tại và thay đổi thuộc tính của chúng nếu cần.

### 1. Cài đặt



- Prototype: khai báo giao diện để nhân bản chính nó. Mỗi Prototype là một đối tượng factory, có thể tạo đối tượng giống chính nó, code dùng tạo đối tượng đặt trong phương thức clone() của nó.

- ConcretePrototype: cài đặt cụ thể tác vụ clone() để nhân bản với dữ liệu lựa chọn.

Đối tượng nhân bản (cloned object) là một đối tượng mới, được tạo ra bằng cách sao chép sâu (deep copy) nên nó thực sự tách rời khỏi đối tượng gốc ban đầu. Sau đó, tùy biến đối tượng được nhân bản.

### Các bước thực hiện

Bắt đầu bằng việc tạo lớp sẽ trở thành nguyên mẫu dùng nhân bản (Prototype):

- Lớp phải cài đặt interface Clonable.
- Lớp phải cài đặt (override) phương thức clone() và trả về bản sao chính nó, trong đó lưu ý đến sao chép sâu (deep copy) hay bề mặt (shallow copy) tùy trường hợp.
- Phương thức phải ném CloneNotSupportedException để cho phép lớp con quyết định có hỗ trợ nhân bản hay không.

```
import java.util.ArrayList;
import java.util.Arrays; import
java.util.HashMap; import
java.util.List;

abstract class ColorPrototype implements Cloneable {
protected List<Integer> list = new ArrayList<>(); protected
ColorPrototype(Integer... colors) {
list.addAll(Arrays.asList(colors));
}

@Override public String toString() {
return list.toString();
}
}

// ConcretePrototype
class Color extends ColorPrototype {
public Color(Integer... colors) {
super(colors);
} public Color setColor(Integer...
colors) { for (int i = 0; i <
colors.length; i++) list.set(i,
colors[i]); return this;
}

@Override public Color clone() throws CloneNotSupportedException {
System.out.println("[LOG]: Cloning");
Color clone = (Color) (ColorPrototype) super.clone();
// deep copy
clone.list = new ArrayList<>(list);
return clone;
}
}
```

```

// Context
class ColorPalette {
 HashMap<String, ColorPrototype> palette = new HashMap<>();
 public HashMap<String, ColorPrototype> getPalette() {
 return palette;
 }
 public ColorPalette addColor(String name, ColorPrototype color) {
 palette.put(name, color);
 return this;
 }
 public void showPalette() {
 palette.keySet().forEach(key -> System.out.printf("%s: %s%n", palette.get(key), key));
 }
}
public class Client {
 public static void main(String[] args) throws CloneNotSupportedException {
 System.out.println("--- Prototype Pattern ---");
 Color base = new Color(0, 0, 0);
 Color red = base.clone().setColor(255, 0, 0);
 Color green = base.clone().setColor(0, 255, 0);
 Color blue = base.clone().setColor(0, 0, 255);
 new ColorPalette()
 .addColor("red", red)
 .addColor("green", green)
 .addColor("blue", blue)
 .showPalette();
 }
}

```

ColorPalette lưu trữ ba đối tượng Color: "red", "green" và "blue"; chúng được "nhân bản" từ Color gốc (base) rồi thiết đặt lại. Chỉ có thực thể gốc "base" này được tạo từ constructor, dùng toán tử new.

## 2. Liên quan

- Abstract Factory: Prototype chính là một đối tượng factory. Vì vậy có thể lưu một họ Prototype để nhân bản và trả về đối tượng được tạo.
- Composite và Decorator: tạo Prototype rồi "phức hợp" thêm bằng Composite hoặc "trang trí" thêm bằng Decorator.

## 3. Java API

Phương thức clone() của java.lang.Object là phương thức tạo, trả về một thể hiện khác của chính đối tượng, có cùng thuộc tính của đối tượng gọi. Lưu ý phương thức clone() chỉ sao chép bề mặt (shallow copy).

Clone thường thực hiện trong "copy constructor", constructor dùng tạo đối tượng từ một đối tượng khác cùng lớp.

Deep clone có thể thực hiện bằng cách serialize rồi deserialize đối tượng. Lớp org.apache.commons.lang3.SerializationUtils có phương thức static roundtrip() thực hiện deep clone theo cơ chế này.

```

import java.util.Arrays;
import org.apache.commons.lang3.SerializationUtils;
class Thing {
 String name;
 int[] list = { 1, 2, 3 };

 public Thing(String name) {
 this.name = name;
 }
 public Thing(Thing other) {
 this.name = other.name;
 this.list = SerializationUtils.roundtrip(list);
 }
}
class Client { public static void
main(String[] args) {
 Thing origin = new Thing("First"); Thing
clone = new Thing(origin); for (int i = 0; i
< origin.list.length; i++) origin.list[i]
*= 2;
 System.out.println(Arrays.toString(origin.list)); // [2, 4, 6]
 System.out.println(Arrays.toString(clone.list)); // [1, 2, 3]
}
}

```

## 4. Sử dụng

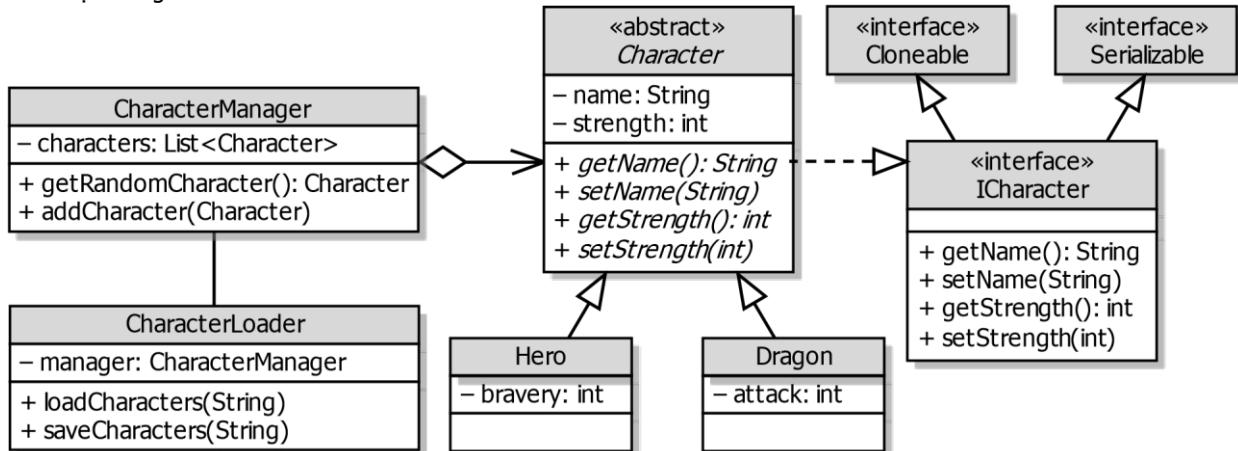
Ta muốn:

- Che giấu quá trình tạo các lớp cụ thể với Client.

- Dùng Prototype để thêm và loại các lớp mới trong thời gian chạy.
- Giữ số lượng các lớp trong hệ thống ở mức tối thiểu. - Thích ứng với thay đổi cấu trúc dữ liệu trong thời gian chạy.

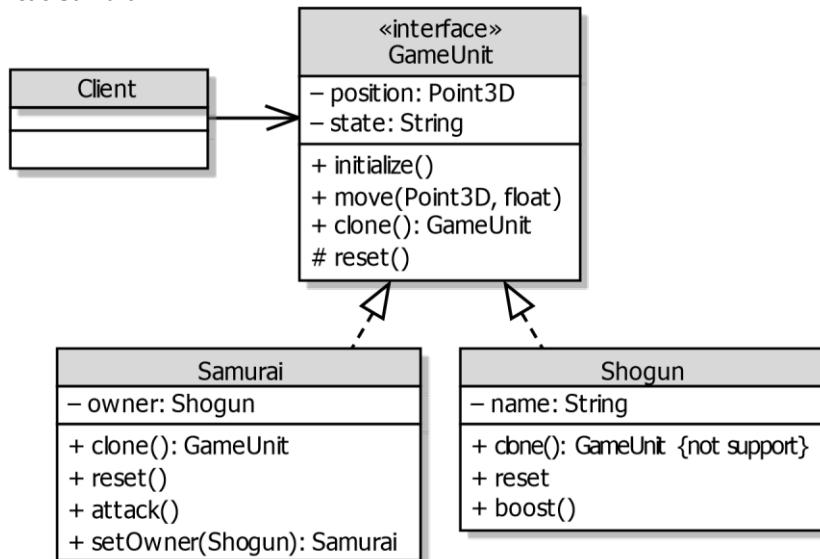
## 5. Bài tập

- a) Trong một trò chơi, mẫu (prototype) của các nhân vật (Character) sẽ xuất hiện trong trò chơi được lưu trong tập tin, CharacterLoader sẽ đọc chúng từ tập tin bằng ObjectStream và lưu vào danh sách characters của CharacterManager bằng phương thức addCharacter() của lớp này. Khi muốn tạo thêm nhân vật, thay đổi lại tập tin, CharacterManager gọi phương thức getRandomCharacter(), phương thức này lấy một prototype ngẫu nhiên trong danh sách characters để nhân bản (clone) ra nhân vật mong muốn.



- b) Nhân vật trong một game (GameUnit) chia làm hai loại:

- Shogun: có tên cụ thể, số lượng phụ thuộc vào số người chơi, mỗi người chơi đóng vai một Shogun.
- Samurai: có số lượng lớn, mỗi nhóm nhiều Samurai thuộc về một Shogun. Hãy sử dụng mẫu thiết kế Prototype để tạo nhanh các Samurai.



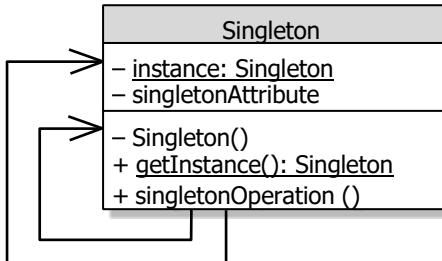
## Singleton

Single object instances

Mẫu thiết kế Singleton đảm bảo rằng một lớp chỉ có một thể hiện (instance) duy nhất. Do thể hiện này có tiềm năng sử dụng trong suốt chương trình, nên mẫu thiết kế Singleton cũng cung cấp một điểm truy cập toàn cục đến nó. Lý do phổ biến cho việc chỉ tạo một thể hiện là để kiểm soát truy cập vào một tài nguyên dùng chung (cơ sở dữ liệu hoặc tập tin).

Hộp thoại Find là một ví dụ điển hình cho mẫu thiết kế Singleton. Dù bạn chọn menu hoặc nhấn Ctrl+F nhiều lần, tại bất kỳ nơi nào trong ứng dụng, chỉ một hộp thoại duy nhất xuất hiện.

### 1. Cài đặt



Mẫu thiết kế Singleton đơn giản và dễ áp dụng, chỉ cần bổ sung vài dòng lệnh trong lớp muốn chuyển thành Singleton.

- Dữ liệu thành viên instance (private và static) là đối tượng duy nhất của lớp Singleton.
- Constructor của lớp Singleton được định nghĩa thành protected hoặc private để ngăn người dùng tạo thực thể trực tiếp từ bên ngoài lớp.
- Phương thức getInstance() dùng khởi tạo đối tượng duy nhất, định nghĩa thành public và static. Client chỉ dùng getInstance() để tạo đối tượng cho lớp Singleton.
- Thực hiện khởi tạo chậm (lazy initialization) trong getInstance(): chỉ khi gọi phương thức getInstance() mới khởi tạo đối tượng. Phương thức này trả về một thể hiện mới hay một null tùy thuộc vào một biến boolean dùng như cờ hiệu báo xem lớp Singleton đã tạo thể hiện hay chưa.

Trong chế độ đa luồng (multithreading), mẫu thiết kế Singleton có thể làm việc không tốt: do getInstance() không an toàn thread, hai thread có thể gọi phương thức sinh đối tượng cùng một thời điểm và hơn một thể hiện sẽ được tạo ra. Giải quyết bằng đồng bộ (synchronized) phương thức getInstance() để an toàn thread sẽ dẫn đến giảm hiệu suất chương trình. Có nhiều giải pháp cho vấn đề này:

#### - double-checked locking (lazy initialization)

Kiểm tra sự tồn tại của lớp, với sự hỗ trợ của đồng bộ hóa, hai lần trước khi khởi tạo. Phải khai báo volatile cho instance để tránh lớp làm việc không chính xác do quá trình tối ưu hóa của trình biên dịch.

```

// Singleton class
Singleton {
 private static Singleton instance;

 private Singleton() { }

 private synchronized static void createInstance() {
 if (instance == null) instance = new
Singleton();
 }

 public static Singleton
getInstance() { if (instance == null)
createInstance(); return instance;
 }
}

public class Client {
 public static void main(String[] args) {
 System.out.println("--- Singleton Pattern ---");
 Singleton single1 = Singleton.getInstance();
 Singleton single2 = Singleton.getInstance(); if
(single1.equals(single2)) {
 System.out.println("Unique Instance");
 }
 }
}

```

Một cách cài đặt khác: class

```

Singleton {
 private static volatile Singleton instance;

 private Singleton() { }

 public static Singleton getInstance() {
 Singleton singleton = Singleton.instance;
 if (singleton == null) { synchronized
(Singleton.class) { singleton =
Singleton.instance;
 if (singleton == null) Singleton.instance = singleton = new Singleton();
 }
 }
 return singleton;
}

```

#### - eager initialization (Bill Pugh singleton)

Thực thể Singleton là biến static và final, được khởi tạo sớm khi lớp lần đầu được nạp vào JVM.

```

import java.util.function.Supplier;
class Singleton { // eager
initialization
 private static final Singleton INSTANCE = new Singleton();

 private Singleton() { }

 public static Supplier<Singleton> getInstance() {
 return () -> INSTANCE;
 }
}
public class Client {
 public static void main(String[] args) {
System.out.println("--- Singleton Pattern ---");
 Singleton single1 = Singleton.getInstance().get();
 Singleton single2 = Singleton.getInstance().get();
 if (single1.equals(single2)) {
System.out.println("Unique Instance");
 }
 }
}
- class loader (static block initialization) class
Singleton {
 private static class SingletonHolder {
 static final Singleton instance = new Singleton();
 }
 private Singleton() { }
 public static Singleton getInstance()
 { return SingletonHolder.instance;
 }
}
public class Client {
 public static void main(String[] args) {
System.out.println("--- Singleton Pattern ---");
 Singleton single1 = Singleton.getInstance();
 Singleton single2 = Singleton.getInstance(); if
(single1.equals(single2)) {
System.out.println("Unique Instance");
 }
 }
}
- enum-based singleton enum
Singleton{
 SINGLETON;
}
public class Client {
 public static void main(String[] args) {
System.out.println("--- Singleton Pattern ---");
 Singleton single1 = Singleton.SINGLETON;
 Singleton single2 = Singleton.SINGLETON; if
(single1.equals(single2)) {
System.out.println("Unique Instance");
 }
 }
}

```

Erich Gamma lưu ý mẫu thiết kế Singleton mang đến những vấn đề về thiết kế. Xem các vấn đề sau: -  
Dùng Java Reflection có thể thiết lập constructor từ private thành truy cập được:

```

try {
 Class c = Class.forName("samples.Singleton");
 Constructor<Singleton> ctor = c.getDeclaredConstructor();
 ctor.setAccessible(true); Singleton s =
 ctor.newInstance();
} catch (ClassNotFoundException | NoSuchMethodException | SecurityException | InstantiationException |
IllegalAccessException | IllegalArgumentException | InvocationTargetException e) {
 System.out.println(e.getMessage());
}

```

- Nếu thể hiện của lớp Singleton được tạo ra với hashCode() khác thể hiện đầu tiên, Singleton đã bị vi phạm.
- + Nếu lớp Singleton cài đặt Cloneable và phương thức clone(), có thể nhân bản đối tượng, hash code khác đối tượng gốc.
- + Nếu serialize rồi deserialize một đối tượng, kết quả nhận được là đối tượng với hash code khác đối tượng gốc.

- Ta thấy các thể hiện của một lớp chỉ khác nhau ở thuộc tính (state). Mẫu thiết kế Monostate dựa trên ý tưởng này, các trạng thái của lớp Monostate đều được thiết lập static, như vậy tất cả các thể hiện của Monostate đều sử dụng cùng một dữ liệu (static).

## 2. Liên quan

- Facade: lớp Façade có thể chuyển thành Singleton nếu đổi tượng Façade duy nhất đủ bao quát các trường hợp sử dụng.
- Abstract Factory: thường là Singleton để trả về các đối tượng factory duy nhất.
- Builder: dùng xây dựng một đối tượng phức tạp, trong đó Singleton được dùng để tạo một đối tượng truy cập tổng quát (Director).
- Prototype: dùng để sao chép một đối tượng, hoặc tạo ra một đối tượng khác từ Prototype của nó, trong đó Singleton được dùng để chắc chắn chỉ có một Prototype.

## 3. Java API

- Lớp java.lang.Runtime là lớp Singleton, để lấy được đối tượng duy nhất của nó, ta gọi phương thức getRuntime().
- Lớp java.awt.Desktop cũng là lớp Singleton, tạo đối tượng duy nhất bằng phương thức getDesktop().
- java.lang.System.getSecurityManager() trả về một đối tượng SecurityManager duy nhất.
- Lớp java.util.logging.Logger, với phương thức getLogger().

Tuy nhiên, lớp Singleton không phổ biến như ta nghĩ, không nên lạm dụng nó mà chỉ áp dụng với lớp cần bảo đảm có duy nhất một thể hiện. Lớp java.lang.Math và lớp java.util.Calendar chẳng hạn, không phải là lớp Singleton.

Erich Gamma cho rằng nếu phải loại bỏ mẫu thiết kế nào đó, ông sẽ chọn Singleton vì nó gây ra những vấn đề trong thiết kế.

## 4. Sử dụng Ta

muốn:

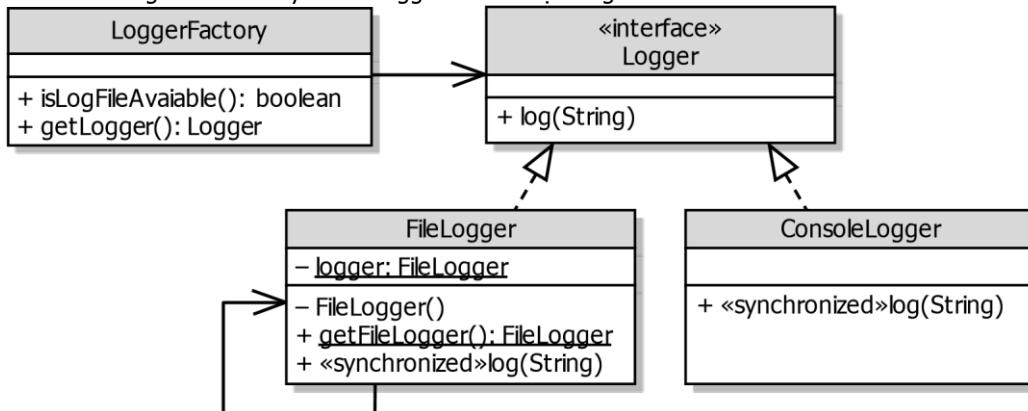
- Đảm bảo rằng chỉ có một thể hiện của lớp.
- Quản lý việc truy cập tốt hơn vì chỉ có một thể hiện duy nhất.
- Quản lý số lượng thể hiện của một lớp. Trường hợp này không nhất thiết chỉ có một thể hiện, bạn cần kiểm soát số lượng thể hiện trong giới hạn chỉ định.

## 5. Bài tập

a) Để xây dựng hệ thống ghi nhận thông tin, người ta dùng giao diện Logger với phương thức log(). Có hai loại Logger: ConsoleLogger xuất thông tin ghi nhận ra màn hình, FileLogger lưu thông tin ghi nhận vào tập tin log.txt, chèn dòng thông tin mới vào phía đầu tập tin.

LoggerFactory chịu trách nhiệm quyết định loại Logger sẽ sử dụng. Nó xem xét mục FileLogging trong tập tin logger.properties, nếu mục này là ON, FileLogger sẽ được tạo và sử dụng; ngược lại, ConsoleLogger sẽ được sử dụng.

Vì chỉ có một tập tin nhật ký vật lý được tham chiếu bởi FileLogger nên FileLogger phải là một Singleton. Viết các lớp cần thiết và áp dụng mẫu thiết kế Singleton để chuyển FileLogger thành một Singleton.



## Structural Design Patterns

### Adapter

Adapt from one to another

Mẫu thiết kế Adapter giữ vai trò trung gian giữa hai lớp, chuyển đổi giao diện của một hay nhiều lớp có sẵn thành một giao diện khác, tương thích với lớp đang viết. Adapter giải quyết vấn đề không tương thích của hai giao diện, điều này cho phép các lớp có các giao diện khác nhau có thể dễ dàng giao tiếp tốt với nhau thông qua giao diện chuyển tiếp trung gian, không cần thay đổi code của lớp có sẵn cũng như lớp đang viết.

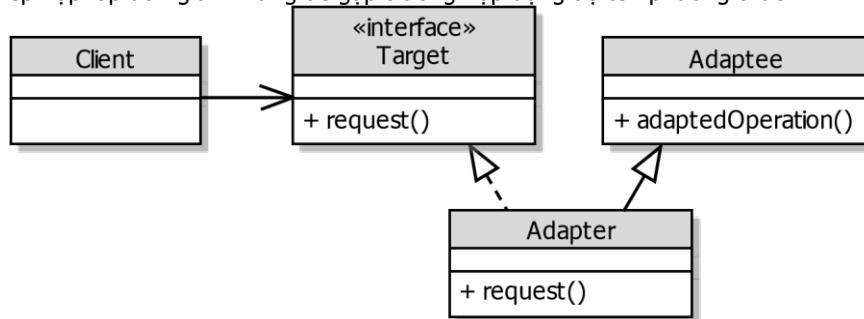
Mẫu thiết kế Adapter còn gọi là Wrapper do cung cấp một giao diện "bọc ngoài" tương thích cho một hệ thống có sẵn. Hệ thống có sẵn thường có dữ liệu và hành vi phù hợp nhưng có giao diện không tương thích với lớp đang viết.

Wrapper tương ứng với nguyên tắc thiết kế Open-Closed: nếu bạn cần một lớp hoạt động hơi khác một chút thì đừng sửa đổi nó; thay vào đó, tạo một lớp mới "bao bọc" lấy nó.

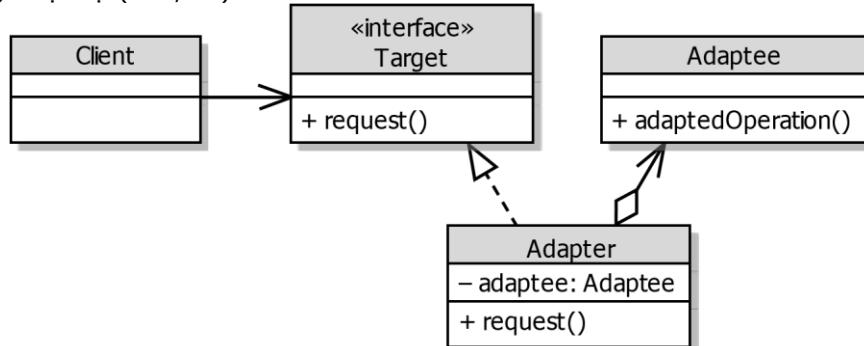
#### 1. Cài đặt

Có hai cách để cài đặt mẫu thiết kế Adapter:

- Tiếp hợp lớp (dùng thừa kế – inheritance): một lớp mới (Adapter) sẽ kế thừa lớp có sẵn với giao diện không tương thích (Adaptee), đồng thời cài đặt giao diện mà người dùng mong muốn (Target). Trong lớp mới, khi cài đặt các phương thức của giao diện người dùng mong muốn, phương thức này sẽ gọi các phương thức cần thiết mà nó thừa kế được từ lớp có giao diện không tương thích. Tiếp hợp lớp đơn giản nhưng dễ gặp trường hợp đụng độ tên phương thức.



- Tiếp hợp đối tượng (dùng tích hợp – composition): một lớp mới (Adapter) sẽ tham chiếu đến một (hoặc nhiều) đối tượng của lớp có sẵn với giao diện không tương thích (Adaptee), đồng thời cài đặt giao diện mà người dùng mong muốn (Target). Trong lớp mới này, khi cài đặt các phương thức của giao diện người dùng mong muốn, sẽ gọi phương thức cần thiết bằng cách ủy nhiệm đối tượng<sup>3</sup> thuộc lớp có giao diện không tương thích. Tiếp hợp đối tượng tránh được vấn đề đa thừa kế, không có trong các ngôn ngữ hiện đại (Java, C#).



Các thành phần tham gia:

- Target: định nghĩa giao diện Client đang làm việc (phương thức **request()**).
- Adaptee: thực hiện được công việc **request()** cần nhưng có giao diện không tương thích, cần được tiếp hợp để sử dụng được.
- Adapter: lớp tiếp hợp, cài đặt giao diện cho Target. Nó ủy nhiệm việc thực hiện phương thức **request()** cho Adaptee, bằng cách gọi **adaptedOperation()** của Adaptee. Adapter có thể chứa mã bổ sung để phù hợp với nhu cầu của Client. Ngoài ra, có thể cung cấp nhiều Adapter.
- Tạo lớp Adapter
- + Cài đặt interface Target mà Client mong đợi.
- + Tiếp hợp lớp: thử thừa kế lớp có sẵn Adaptee, khi cài đặt các phương thức của Target, gọi các phương thức thừa kế từ Adaptee để thực hiện.
- + Tiếp hợp đối tượng: Adapter nhận Adaptee từ constructor, khi cài đặt các phương thức của Target, ủy nhiệm cho Adaptee thực hiện.

```

// Target interface
Target {
 double getArea(Point topleft, Point rightbottom);
}

class Point {
 double x, y;
 public Point (double x, double y) {
 this.x = x; this.y = y;
 }
}

// Adaptee class
Adaptee {
 public double getArea(double x1, double y1, double x2, double y2) {
 return Math.abs((x2 - x1) * (y2 - y1));
 }
}

// Adapter
class Adapter extends Adaptee implements Target {
}

```

<sup>3</sup> Nếu Adapter tham chiếu đến nhiều đối tượng Adaptee, có thể thực hiện được tính đa hình (polymorphism) trong Adapter.

```

@Override public double getArea(Point topleft, Point rightbottom) {
 return getArea(topleft.x, topleft.y, rightbottom.x, rightbottom.y);
}
public class Client {
 public static void main(String[] args) {
 System.out.println("--- Adapter Pattern ---");
 Target target = new Adapter();
 System.out.printf("Area = %.1f%n", target.getArea(new Point(1, 4), new Point(5, 1)));
 }
}

```

Client định gọi phương thức getArea() của Adaptee để tính diện tích hình chữ nhật nhưng không thực hiện được do khác danh sách tham số. Adapter giải quyết vấn đề này: phương thức getArea() của Adapter có signature phù hợp với lời gọi của Client, và khi thực thi nó chuyển tiếp lời gọi đến phương thức getArea() của Adaptee. Adapter chỉ đóng vai trò chuyển đổi giao diện, nó ủy nhiệm cho Adaptee thực hiện lời gọi getArea().

## 2. Liên quan

- Bridge: có cấu trúc tương tự nhưng mục tiêu khác (tách một giao diện khỏi phần cài đặt).
- Decorator: bổ sung thêm chức năng nhưng không làm thay đổi giao diện, trong mẫu thiết kế Decorator, một Adapter sẽ dùng để phối hợp hai đối tượng khác nhau.
- Proxy: cung cấp cùng một giao diện với đối tượng được đại diện. Adapter cung cấp một giao diện khác cho một đối tượng.

## 3. Java API

Mẫu thiết kế Adapter dùng phổ biến trong Java AWT (WindowAdapter, ComponentAdapter, ContainerAdapter, FocusAdapter, KeyAdapter, MouseAdapter và MouseMotionAdapter), ...

Ví dụ: giao diện WindowListener có 7 phương thức. Khi lớp lắng nghe sự kiện (listener) của ta cài đặt giao diện này, cần phải cài đặt tất cả 7 phương thức xử lý sự kiện, dù một số phương thức chỉ cài đặt rỗng do không dùng đến loại sự kiện đó. Lớp WindowAdapter cài đặt giao diện WindowListener, cài đặt sẵn và rỗng cả 7 phương thức. Như vậy nếu lớp lắng nghe sự kiện của ta thừa kế lớp WindowAdapter, chỉ viết lại (override) chỉ những phương thức ta muốn thay đổi.

Các lớp InputStream, OutputStream và các lớp con của chúng cung cấp các cách để đọc và ghi byte giữa các loại đầu vào và đầu ra khác nhau. Nhưng các hoạt động ở mức byte thường quá thấp là không thực tế, vì vậy

Java có các lớp hỗ trợ các hoạt động cấp cao hơn. Các lớp Reader, Writer và các lớp con của chúng cài đặt các hoạt động cấp cao bằng cách sử dụng mẫu thiết kế Adapter, chúng bao bọc các InputStream và OutputStream tương ứng.

## 4. Sử dụng Ta muốn:

- Sử dụng một lớp đã tồn tại trước đó nhưng giao diện sử dụng không phù hợp như mong muốn, ta lại không có mã nguồn để sửa đổi giao diện đó.
- Sử dụng một lớp, nhưng lớp này được tạo ra với mục đích chung, nên không thích hợp cho việc tạo một giao diện đặc thù.
- Có sự chuyển đổi giao diện từ nhiều nguồn khác nhau.
- Tiếp hợp nhiều đối tượng cùng một lúc, nhưng giao diện mong muốn không phải là interface mà là một lớp trừu tượng.

## 5. Bài tập

a) Công ty Liberty Bell Tours điều hành tuyến du lịch xe buýt ở Philadelphia, Pennsylvania. Xe buýt được trang bị cơ chế điều hướng hiệu Aquarius, đã hoạt động tốt trong một vài năm. Các chủ sở hữu của công ty gần đây biết đến cơ chế điều hướng mới hiệu Capricorn, mang lại độ tin cậy cao hơn và ít bảo trì định kỳ hơn Aquarius. Họ quyết định thay thế đơn vị Aquarius bằng đơn vị Capricorn trong lần bảo trì xe buýt tiếp.

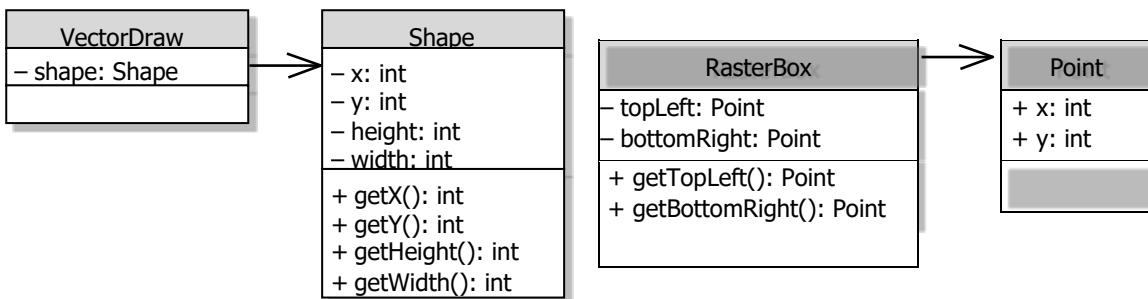
Aquarius hỗ trợ giao diện NavigationUnit, cho phép theo dõi hướng xe buýt dựa trên bốn hướng la bàn chính: Bắc, Nam, Đông và Tây. Theo đó, phần mềm trên xe buýt đã được cấu hình để sử dụng thiết bị Aquarius thông qua giao diện NavigationUnit.

Đơn vị Capricorn mới không dựa trên hướng mà trên góc phương vị (bearing), dùng góc giữa  $0^\circ$  và  $360^\circ$  để biểu thị hướng đi. Capricorn không hỗ trợ giao diện NavigationUnit và cũng không hỗ trợ hướng la bàn. Vì vậy, để nó sử dụng được với phần mềm trên xe buýt, cần phải chuyển đổi góc phương vị và hướng: - Hướng Bắc tương đương với góc phương vị là  $0^\circ$

- Hướng Đông tương đương với góc phương vị là  $90^\circ$
- Hướng Nam tương đương với góc phương vị là  $180^\circ$
- Hướng Tây tương đương với góc phương vị là  $270^\circ$

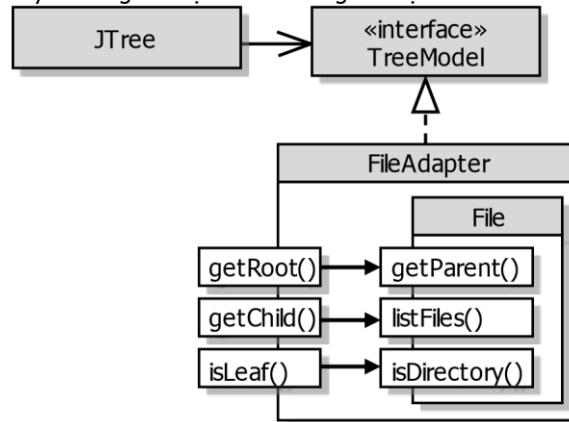
Thay vì viết lại tất cả phần mềm hiện có cho xe buýt để sử dụng được đơn vị Capricorn mới, nhà phát triển quyết định áp dụng mẫu thiết kế Adapter. Khi có một yêu cầu hướng từ phần mềm xe buýt, lớp tiếp hợp sẽ yêu cầu từ Capricorn góc phương vị hiện tại và sau đó chuyển đổi giá trị này thành hướng la bàn tương ứng để gửi lại phần mềm xe buýt. Hãy thực hiện lớp tiếp hợp này.

b) Lớp VectorDraw của khách hàng đã sử dụng lớp Shape. Áp dụng mẫu thiết kế Adapter để cho lớp VectorDraw sử dụng được thêm lớp RasterBox (và lớp Point mà RasterBox sử dụng).

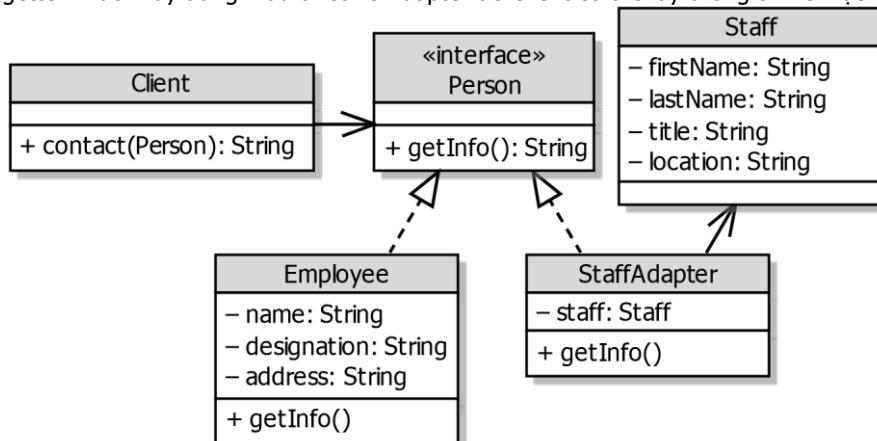


c) Duyệt hệ thống tập tin và hiển thị lên JTree theo sơ đồ sau:

JTree (View) □ TreeModel (Model) □ **FileAdapter** □ File Bạn  
hãy sử dụng mẫu thiết kế Adapter, chuyển đổi giao diện File thành giao diện TreeModel.



d) Trước đây, lớp Client gọi phương thức contact(Person) để lấy thông tin liên lạc của một Person. Phương thức này gọi các getter của Employee, lớp cài đặt giao diện Person. Khi mở rộng hệ thống, có thêm lớp Staff chứa các thông tin tương tự nhưng truy cập bằng getter khác. Hãy dùng mẫu thiết kế Adapter để Client có thể lấy thông tin liên lạc từ lớp Staff.



e) Dữ liệu chứng khoán được lấy từ RSS có định dạng XML. Tuy nhiên thư viện phân tích dữ liệu lại làm việc với dữ liệu đầu vào có định dạng JSON. Hãy áp dụng mẫu thiết kế Adapter, viết lớp tiếp hợp XMLtoJSON để giải quyết vấn đề không tương thích giữa hai bên cung cấp và tiêu thụ dữ liệu trên.

## Bridge

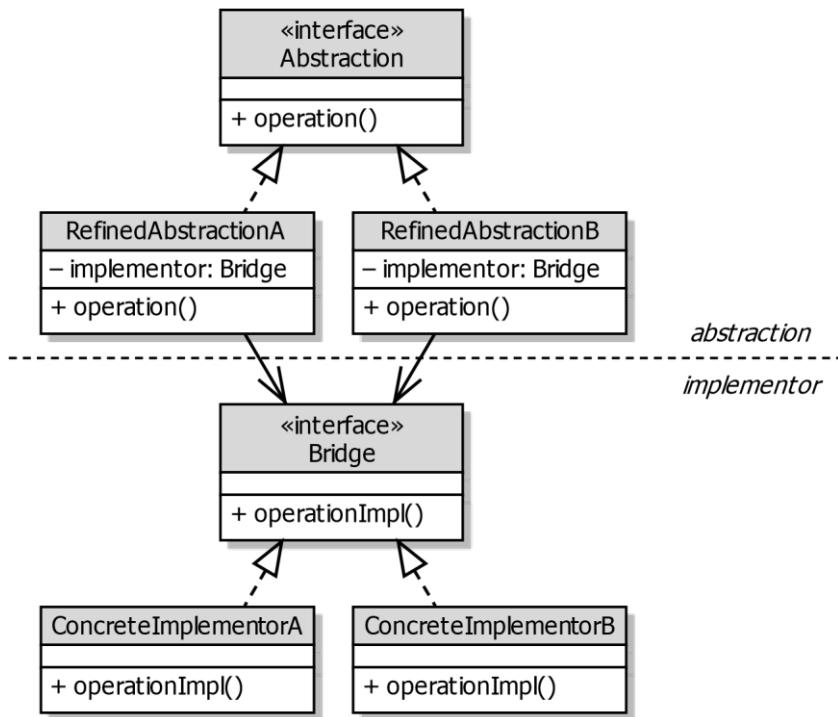
Decouple abstraction from implementation

Phần trừu tượng (Abstraction) và phần cài đặt (Implementor) thường có quan hệ chặt chẽ dựa trên thừa kế. Mẫu thiết kế Bridge dùng tách chúng thành hai thành phần: phần trừu tượng (Abstraction) và phần cài đặt (Implementor), giúp bạn có thể thay đổi hai thành phần này (gọi là "hai đầu cầu") một cách độc lập. Mẫu thiết kế Bridge cung cấp một cầu nối (bridge) giữa phần trừu

tương và phần cài đặt. Do đặc điểm này, Bridge còn gọi là Handle/Body.

Mẫu thiết kế Bridge cần khi một phiên bản mới của ứng dụng được dùng để thay thế phiên bản cũ, nhưng phiên bản cũ vẫn còn chạy trên cơ sở Client cũ. Code của Client sẽ không thay đổi do tương thích với trừu tượng cũ.

### 1. Cài đặt



- Abstraction: giao diện trừu tượng mà Client nhìn thấy.
- RefinedAbstraction: dẫn xuất của Abstraction để cải tiến phần trừu tượng, ví dụ thêm phương thức. Không phải cài đặt cụ thể cho Abstraction.
- Bridge: còn gọi là Implementor, giao diện cầu nối giữa phần trừu tượng và phần cài đặt. Các phương thức của nó không cần giống với Abstraction.
- ConcreteImplementor: cài đặt cụ thể cho phần trừu tượng Abstraction, đã được Bridge tách ra để tùy biến riêng. Nói cách khác, cài đặt thì cho Abstraction nhưng giao diện thì giống Bridge. Client thông qua Abstraction vẫn dùng được phần cài đặt này do gọi các phương thức thông qua Bridge.
- Client chỉ làm việc với phần trừu tượng, công việc của Client là liên kết đối tượng "phần trừu tượng" với một trong các đối tượng thuộc "phần cài đặt". **Các bước thực hiện**
- Xác định các thành phần tạo thành cặp khái niệm trừu tượng/cài đặt. Ví dụ, domain/infrastructure, front-end/back-end, ...
- Định nghĩa "phần trừu tượng" trong Abstraction, xác định các tác vụ cơ bản Client cần.
- Phát triển thêm "phần trừu tượng" nếu cần, ví dụ cài đặt thêm tác vụ cho các lớp RefinedAbstraction.
- Định nghĩa "phần cài đặt" trong Implementor, các phương thức của Implementor không trùng với Abstraction. Tuy nhiên bạn có thể thực hiện công việc của Abstraction bằng cách dùng các phương thức của Implementor, do đó gọi Implementor là "cầu nối" Bridge.
- Cài đặt các lớp ConcreteImplementor.

`import java.util.Arrays;`

```

// Abstraction abstract class
Shape { protected Drawing
implementor; abstract public
void draw();
protected Shape(Drawing dp)
{ this.implementor = dp;
}
}

// RefinedAbstraction class
Rectangle extends Shape {
double x1, y1, x2, y2;
public Rectangle(Drawing dp, double x1, double y1, double x2, double y2) {
super(dp);
this.x1 = x1;
this.x2 = x2; this.y1
= y1; this.y2 = y2;
}

@Override public void draw() {
implementor.drawLine(x1, y1, x2, y1);
implementor.drawLine(x2, y1, x2, y2);
implementor.drawLine(x2, y2, x1, y2);
implementor.drawLine(x1, y2, x1, y1);
}
}

```

```

}
class Circle extends Shape {
double x, y, r;
public Circle(Drawing dp, double x, double y, double r) {
super(dp); this.x = x; this.y = y; this.r = r;
}

@Override public void draw() {
implementor.drawCircle(x, y, r);
}
}

// Bridge
abstract class Drawing {
abstract public void drawLine(double x1, double y1, double x2, double y2);
abstract public void drawCircle(double x, double y, double r); }

// ConcreteImplementor
class V1Drawing extends Drawing {
@Override public void drawLine(double x1, double y1, double x2, double y2) {
System.out.printf("%.1f, %.1f)----(%1f, %1f)%n", x1, y1, x2, y2); }

@Override public void drawCircle(double x, double y, double r) {
System.out.printf("%.1f, %.1f)---[%1f]--->%n", x, y, r); }
}
class V2Drawing extends Drawing {
@Override public void drawLine(double x1, double y1, double x2, double y2) {
System.out.printf("%.1f, %.1f).....(%1f, %1f)%n", x1, y1, x2, y2); }

@Override public void drawCircle(double x, double y, double r) {
System.out.printf("<---[%1f]---(%1f, %1f)%n", r, x, y); }
}

public class Client {
public static void main(String argv[]) {
System.out.println("----");
Bridge Pattern --");
Drawing dp1 = new V1Drawing();
Drawing dp2 = new V2Drawing();
Shape[] shapes = { new Rectangle(dp1, 1, 1, 2, 2), new Circle(dp1, 2, 2, 3),
new Rectangle(dp2, 1, 1, 2, 2), new Circle(dp2, 2, 2, 3) };
Arrays.stream(shapes).forEach(Shape::draw);
} }

```

Client chỉ làm việc với phần trừu tượng, trong lúc cài đặt cụ thể của phần trừu tượng được thay đổi, phát triển độc lập ở "đầu cầu bên kia". Bridge giữ vai trò cầu nối giữa phần trừu tượng và phần cài đặt cụ thể.

## 2. Liên quan

- Abstract Factory: Abstract Factory có thể tạo và cấu hình một Bridge cụ thể.
- Adapter: Adapter được áp dụng nếu hệ thống đã thiết kế xong, giúp cho các lớp không liên quan làm việc được với nhau. Trái ngược với Adapter, Bridge được thiết kế ngay từ đầu, cho phép phần trừu tượng và phần hiện thực được tùy biến độc lập, để có thể mở rộng hệ thống sau khi thiết kế.

## 3. Java API

Java AWT 1.1.x dùng mẫu thiết kế Bridge để tách biệt các component trừu tượng với phần cài đặt phụ thuộc hệ nền của chúng. Ví dụ, java.awt.Button hoàn toàn bằng Java trong lúc sun.awt.windows.WButtonPeer được cài đặt bằng code Windows gốc (native).

JDBC API, chúng ta làm việc với DriverManager ở một "đầu cầu" và các Driver làm việc với cơ sở dữ liệu ở "đầu cầu" bên kia.

## 4. Sử dụng Ta có:

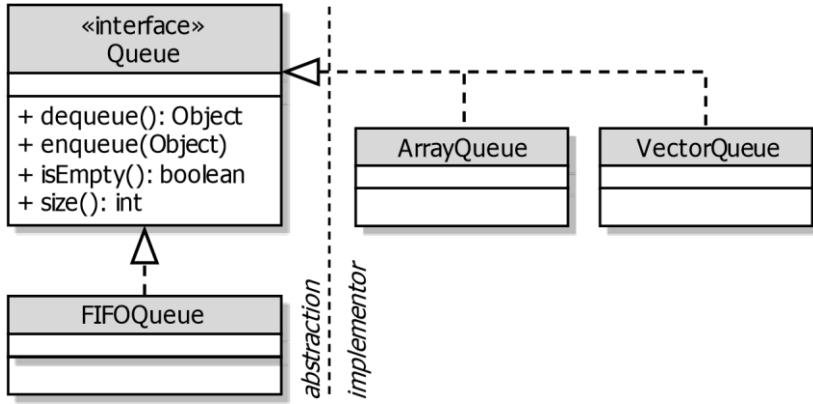
- Phần cài đặt có khả năng thay đổi nhiều phiên bản, trong lúc không muốn thay đổi code của Client làm việc với phần trừu tượng.

Ta muốn:

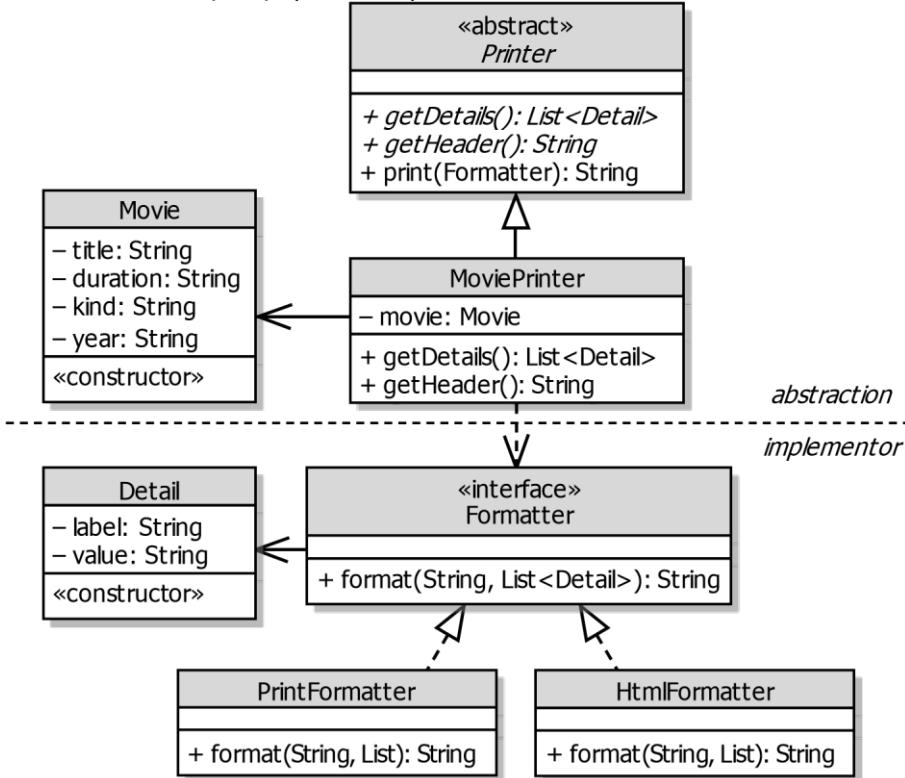
- Che giấu hoàn toàn phần cài đặt với Client.
- Tránh ràng buộc trực tiếp giữa phần trừu tượng và phần cài đặt.
- Thay đổi phần cài đặt mà không cần biên dịch lại phần trừu tượng.
- Kết hợp các phần khác nhau của một hệ thống trong thời gian chạy.

## 5. Bài tập

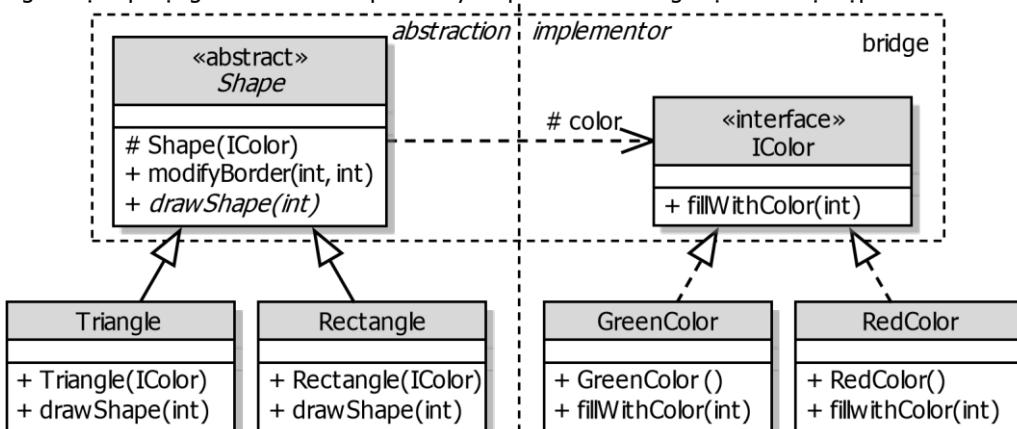
a) Chương trình cần mở rộng khái niệm trừu tượng Queue bằng cách thêm một phân lớp FIFOQueue. Điều này không ảnh hưởng đến các phần cài đặt cụ thể cho Queue như ArrayQueue, VectorQueue. Hãy dùng mẫu thiết kế Bridge để thực hiện yêu cầu này.



b) Phần trừu tượng Printer được phát triển độc lập với phần cài đặt thực hiện các định dạng khi in thông tin. Áp dụng mẫu thiết kế Bridge như hình sau để thực hiện yêu cầu này:

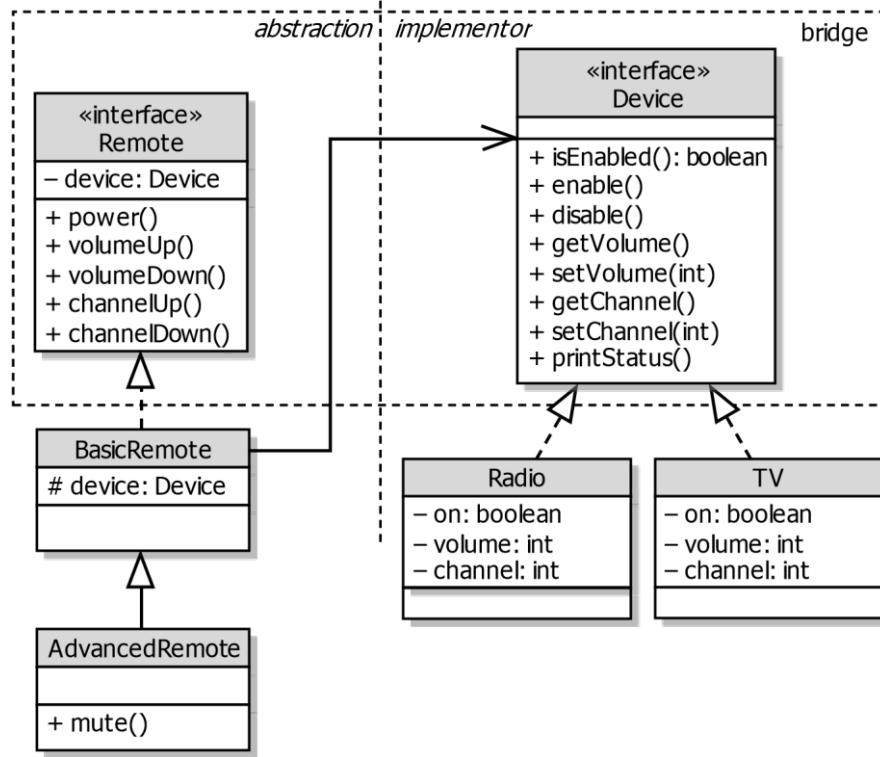


c) Phần mềm đồ họa phát triển "phần trừu tượng" là các lớp mở rộng từ lớp cơ sở trừu tượng Shape: Triangle, Rectangle, ... Việc phát triển này phải không ảnh hưởng đến "phần cài đặt" liên quan đến tô màu và thay đổi kích thước biên của Shape. Mẫu thiết kế Bridge được áp dụng để tách rời hai phần này và phát triển chúng một cách độc lập.



d) Interface Device đóng vai trò như "phần cài đặt", trong lúc lớp Remote đóng vai trò như "phần trừu tượng". Remote làm việc với các thiết bị thông qua interface Device cho phép một Remote hỗ trợ nhiều thiết bị (cài đặt interface Device). Bạn

có thể phát triển "phần trừu tượng" Remote độc lập với các lớp Device cụ thể, ví dụ dẫn xuất q thêm lớp AdvancedRemote với nút mới mute(). Áp dụng mẫu thiết kế Bridge để thực hiện điều này.



## Composite

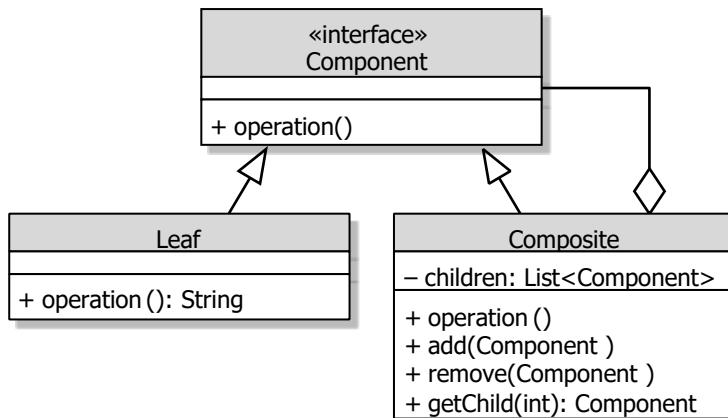
Uniformly treat tree objects

Mẫu thiết kế Composite cung cấp một giao diện chung để xử lý với một đối tượng đơn theo cách tương tự như xử lý với một đối tượng phức hợp. Hai loại đối tượng (Component) này được tổ chức như một cấu trúc cây:

- Đối tượng đơn (primitive object) xem như node lá (Leaf, hoặc terminal) của cây.
- Đối tượng phức hợp (Composite) xem như các node trong (non-terminal) của cây, có thể chứa các đối tượng đơn (node lá) hoặc các đối tượng phức hợp khác. Nói cách khác, đối tượng phức hợp chứa một nhóm các đối tượng Component.

Ngoài tác vụ xử lý chung, đối tượng phức hợp còn có các tác vụ điển hình để xử lý các đối tượng con của nó: thêm đối tượng (add), loại bỏ đối tượng (remove), lấy nhóm đối tượng con (getChild), hiển thị các đối tượng (print), tìm kiếm (find).

### 1. Cài đặt



- Component: giao diện khai báo tác vụ xử lý chung cho Leaf và Composite, tác vụ này là hành vi đa hình.
- Leaf: đối tượng đơn, không chứa các đối tượng con khác, tương tự như node lá của cây. Vì thế nó chỉ chứa tác vụ xử lý chung, không chứa các tác vụ dùng truy cập đối tượng con.
- Composite: đối tượng phức hợp, tương tự node trong của cây, chứa danh sách các đối tượng con (node lá hoặc node phức hợp). Ngoài tác vụ xử lý chung, còn có các phương thức cho phép truy cập các đối tượng con.
- Client làm việc với tất cả các thành phần thông qua interface Component. Do đó Client có thể làm việc với thành phần đơn cũng như với thành phần phức hợp theo cách giống nhau. **Các bước thực hiện**
- Định nghĩa Component, cung cấp tác vụ `operation()`, tác vụ này không cần quan tâm nó chạy với Leaf hoặc với Composite.
- Cài đặt cho Composite. Composite có tác vụ `add/remove` các con của nó. Các tác vụ này có thể chuyển lên Component và không cài đặt trong Leaf. Tác vụ `operation()` trên Composite sẽ thực hiện trên tất cả các con của nó. Vì vậy, khi truy cập đến node gốc sẽ cho phép duyệt toàn bộ cây.
- Cài đặt tác vụ `operation()` cho Leaf. `import java.util.ArrayList;` `import java.util.List;`

```

// Component
interface Graphic {
void paint();
}

// Composite
class Shape implements Graphic {
List<Graphic> sides = new ArrayList<>();
public Shape addSide(Graphic side) {
sides.add(side); return this;
}
public void removeSide(Graphic side) {
sides.remove(side);
}
public Graphic getSide(int i) {
return sides.get(i);
}

@Override public void paint() {
sides.forEach(Graphic::paint);
}
}
//
Leaf
class Line implements Graphic {
String name; public
Line(String name) {
this.name = name;
}

@Override public void paint() {
System.out.printf("Line [%s]%n", name);
}
}
public class Client {
public static void main(String[] args) {
System.out.println("--- Composite Pattern ---");
new Shape().addSide(new Line("left")).addSide(new Line("top"))
.addSide(new Line("right")).addSide(new Line("bottom")).paint();
} }

```

Một đối tượng đồ họa (Graphic) phức hợp (Shape) là tổ hợp các đối tượng đồ họa đơn (Line). Để xử lý chúng (paint) như nhau, đưa chúng vào một cấu trúc cây và áp dụng mẫu thiết kế Composite.

## 2. Liên quan

- Chain of Responsibility: các liên kết đến lớp cha thường được dùng trong Chain of Responsibility.
- Decorator: thường được sử dụng với Composite, lúc đó Component chỉ khai báo operation() trong giao diện, Decorator sẽ mở rộng giao diện bằng cách thêm các tác vụ truy cập đối tượng con (add, remove, getChild).
- Flyweight: cho phép dùng chung các Component, nhưng không tham chiếu đến lớp cha.
- Iterator: thường dùng để duyệt danh sách con trong đối tượng Composite. - Visitor: xác định các tác vụ và hành vi sẽ "viếng thăm" các lớp Leaf và Composite.

## 3. Java API

Mẫu thiết kế Composite dùng nhiều trong các framework UI, cho phép dễ dàng thể hiện các cây UIControl. java.awt.Container là một Composite, phương thức add(Component) của nó cho phép thêm vào nó một Component hoặc một Container khác. Framework jUnit áp dụng mẫu thiết kế Composite, với giao diện Test (Component), TestCase (Leaf) và TestSuite (Composite). Trong JavaEE, mẫu thiết kế Composite View (lớp Presentation) thường dùng cho các ứng dụng portal, được phát triển từ mẫu thiết kế Composite.

## 4. Sử dụng Ta có:

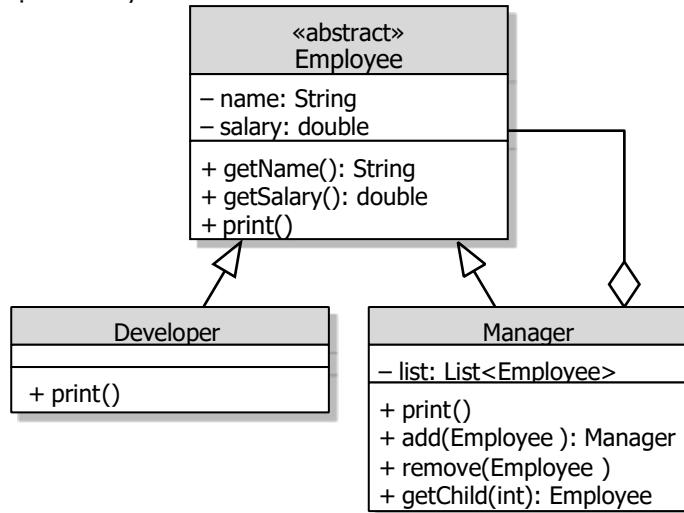
- Một cấu trúc chứa các đối tượng đơn và đối tượng phức hợp hoặc một cấu trúc đệ quy. Ta muốn:
- Client bỏ qua tất cả những khác biệt cơ bản giữa các đối tượng đơn và đối tượng phức hợp.
- Đối xử thống nhất cả các đối tượng đơn lẫn đối tượng phức hợp.

Xem xét sử dụng:

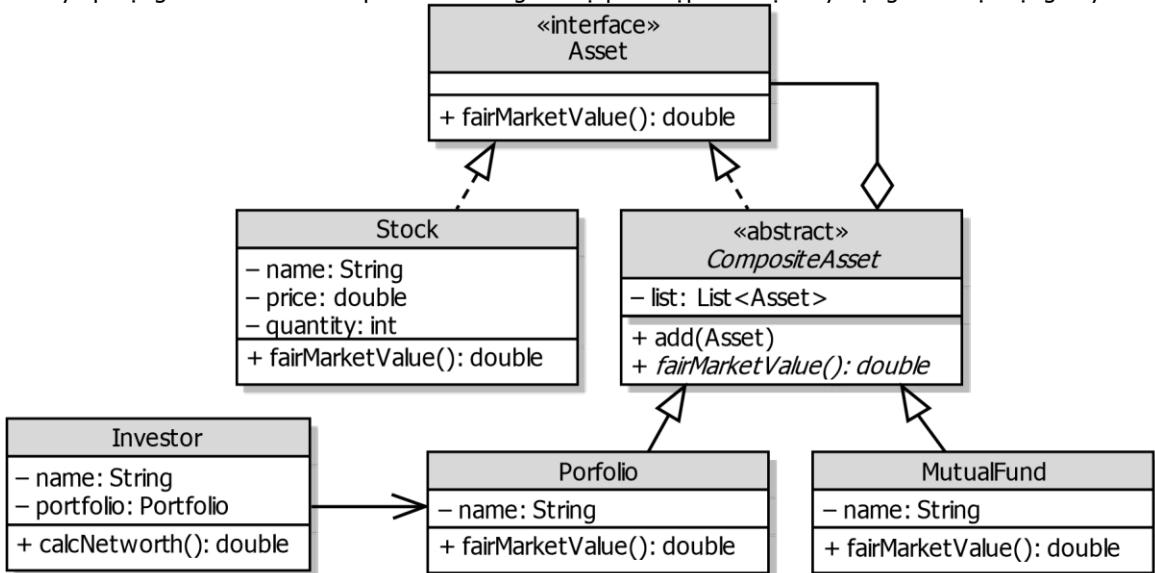
- Decorator, cung cấp các tác vụ mở rộng như thêm đối tượng, loại bỏ đối tượng và tìm kiếm đối tượng.
- Flyweight, để dùng chung trạng thái cho các Component.
- Visitor, xác định các tác vụ được phân phối trên các lớp Leaf và Composite.

## 5. Bài tập

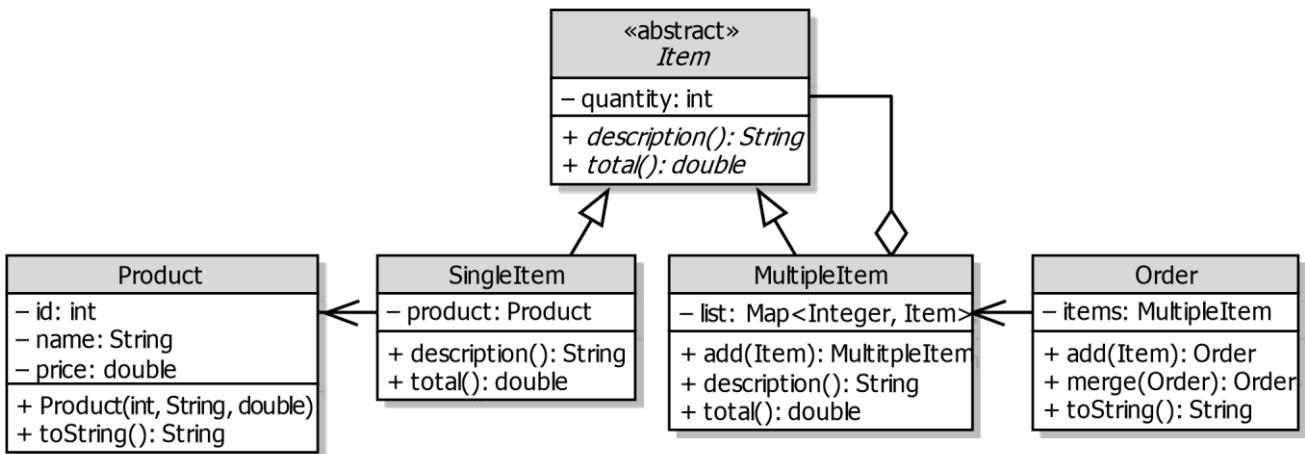
- a) Developer và Manager đều là các Employee, nhưng Manager quản lý một danh sách (có thể rỗng) các Employee khác, bao gồm các Developer và các Manager khác dưới quyền. Với Developer, phương thức print() chỉ in ra thông tin cơ bản. Với Manager, ngoài thông tin cơ bản, phương thức print() còn in thông tin các Employee do Manager đó quản lý. Áp dụng mẫu thiết kế Composite để thực hiện điều này.



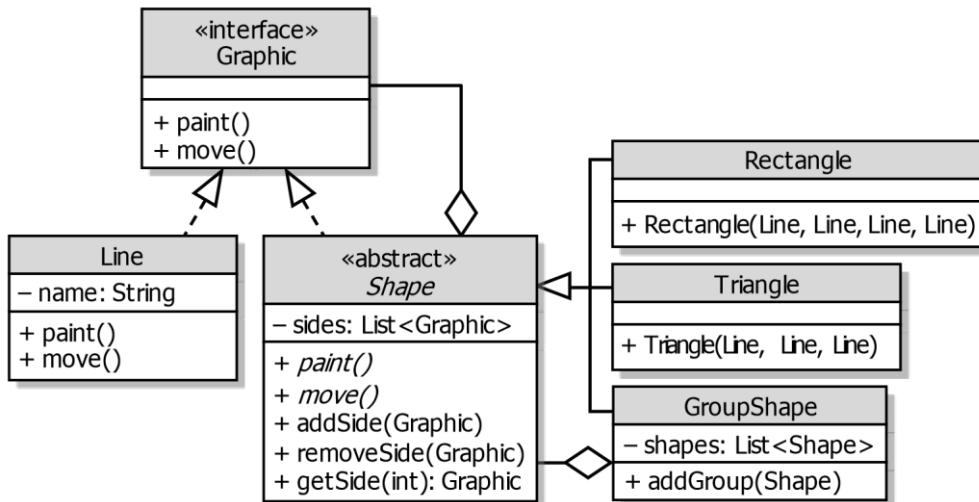
- b) Một hệ thống quản lý danh mục đầu tư (Portfolio), bao gồm Stock (cổ phiếu) và Mutual Fund (quỹ đầu tư mở), một Mutual Fund là một danh sách các Stock. Ta muốn áp dụng một giao diện chung cho cả Stock và Mutual Fund để đơn giản hóa việc xử lý. Điều này cho phép thực hiện các hoạt động như tính toán giá trị thực sự của tài sản (fair market value), hoặc mua, bán các loại tài sản. Hãy áp dụng mẫu thiết kế Composite để làm giảm sự phức tạp của việc xây dựng các hoạt động này.



- c) Một Order (đơn hàng) chứa một Item. Một Item có thể là một SingleItem (Product, quantity) hoặc một MultipleItem (danh sách các Item). Tổ chức như vậy cho phép dễ dàng ghép (merge) hai Order thành một Order duy nhất từ nhu cầu thực tế. Hãy áp dụng mẫu thiết kế Composite để thực hiện yêu cầu.



d) Cài đặt cho mẫu thiết kế Composite, mở rộng ví dụ minh họa.



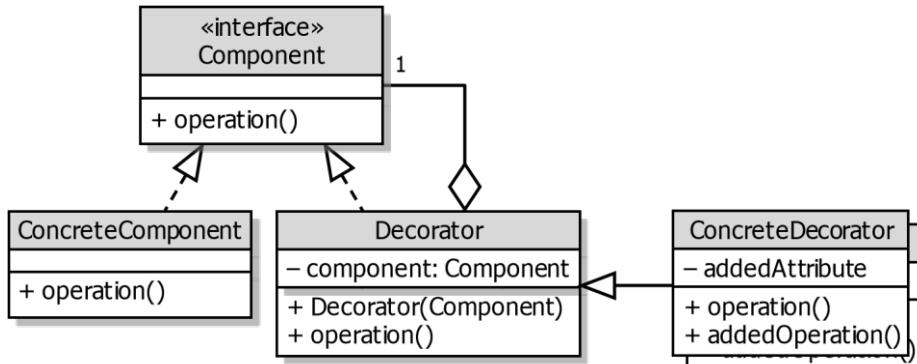
## Decorator

Dynamically add responsibility

Mẫu thiết kế Decorator bổ sung thêm tác vụ (hành vi) cho một đối tượng, nhưng không theo chiến lược thừa kế để mở rộng thêm chức năng mà dùng phương pháp "bao bọc" đối tượng gốc lại để "trang trí" thêm chức năng. Điều này cho phép các tác vụ bổ sung có thể được loại bỏ khi không cần, hoặc được thêm vào theo một thứ tự khác. Do đặc điểm này, Decorator còn gọi là Wrapper giống như mẫu thiết kế Adapter.

Ý tưởng là cho phép phát triển tiếp code, không cần thay đổi trên code gốc mà vẫn đáp ứng được yêu cầu thay đổi, đảm bảo nguyên tắc OCP và SRP.

### 1. Cài đặt



- Component: định nghĩa giao diện của đối tượng mà ta muốn thêm tác vụ đến nó một cách động.
- ConcreteComponent: định nghĩa đối tượng cụ thể ta muốn thêm tác vụ đến nó.
- Decorator: giữ một tham chiếu tới một đối tượng Component, định nghĩa giao diện phù hợp với giao diện của Component. Chú ý đặc điểm constructor của nó nhận đối số là đối tượng được bổ sung thêm chức năng.

Client sử dụng khả năng "lồng" Decorator này vào Decorator khác cho phép bạn xây dựng nên các cấu trúc động.

- ConcreteDecorator: ủy nhiệm lời gọi operation() đến đối tượng Component trong nó, thêm tác vụ bổ sung addedOperation().

### Các bước thực hiện

- Định nghĩa interface Component, khai báo giao diện (operation()) mà Client cần dùng.

- Cài đặt các ConcreteComponent.
- Định nghĩa Decorator, tham chiếu đến Component, Decorator "lồng" với Component có trước bằng cách truyền Component đó như tham số trong constructor.
- Định nghĩa ConcreteDecorator.
- + Phương thức operation() ủy nhiệm cho Component trong nó thực hiện.
- + Phương thức addOperation() cung cấp thêm hành vi muốn bổ sung cho thực thể Component trong nó.

```

// Component
interface Component {
void draw();
}

// ConcreteComponent
class Window implements Component {
@Override public void draw() {
 System.out.println("Draw window");
}
}

// Decorator
class Decorator implements Component {
Component component;
public Decorator(Component component) {
this.component = component;
}

@Override public void draw() {
component.draw();
}
}

// ConcreteDecorator
class ScrollbarWindow extends Decorator {
String scrollbar = "scrollbar";
public ScrollbarWindow(Component component) {
super(component);
}

@Override public void draw() {
super.draw(); System.out.println("Draw "
+ scrollbar);
}
}
class IconWindow extends Decorator {
String icon = "icon";
public IconWindow(Component component) {
super(component);
}

@Override public void draw() {
super.draw();
 System.out.println("Draw " + icon);
}
}
public class Client {
public static void main(String[] args) {
System.out.println("--- Decorator Pattern ---");
Component iconWindow = new IconWindow(new ScrollbarWindow(new Window()));
iconWindow.draw();
}
}

```

## 2. Liên quan

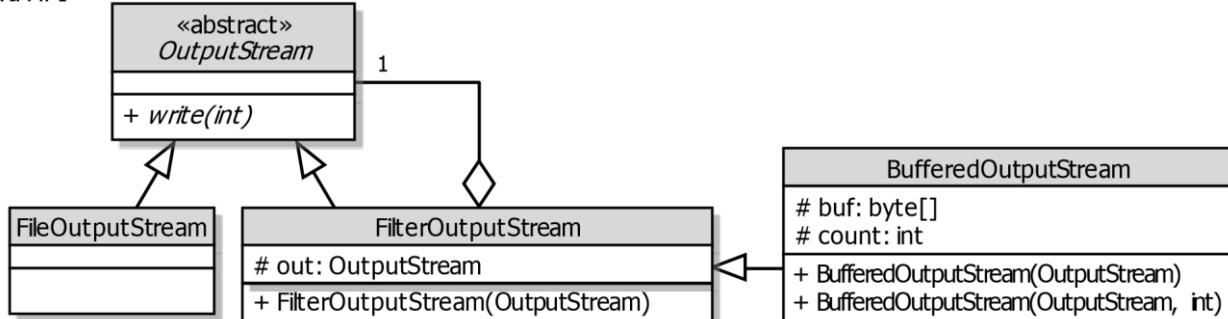
- Adapter: Decorator cũng gọi là Wrapper như Adapter, nhưng chúng có mục đích khác nhau. Adapter bao đổi tương để thay đổi giao diện của chúng, trong lúc Decorater bao đổi tương để bổ sung thêm hành vi cho nó.
- Composite: Decorator cũng có thể coi như một Composite bị thoái hóa với duy nhất một thành phần (Leaf). Tuy nhiên Decorator không có mục tiêu tạo đổi tương phức hợp (Composite), nó chỉ thêm chức năng cho đổi tương.
- Strategy: Decorator thay đổi bên ngoài của đổi tương, trong lúc Strategy thay đổi bên trong của đổi tương.
- Factory Method: đôi khi được dùng với Decorator để đóng gói các thủ tục thiết lập.

```

// dùng interface và các lớp trong ví dụ trên
interface Creator {
Component createWindow();
}
class ScrollbarWindowCreator implements Creator {
@Override public Component createWindow() {
return new ScrollbarWindow(new Window());
}
}
class IconWindowCreator implements Creator {
@Override public Component createWindow() {
return new IconWindow(new ScrollbarWindow(new Window()));
}
}
public class Client {
public static void main(String[] args) {
System.out.println("--- Decorator + Factory Method ---");
Creator[] creators = { new ScrollbarWindowCreator(), new IconWindowCreator() };
for (Creator creator : creators) {
creator.createWindow().draw();
}
}
}

```

### 3. Java API



Stream API (java.io.InputStream, OutputStream, Reader và Writer) được thiết kế theo mẫu thiết kế Decorator. Điều này cho phép "lồng stream": gọi constructor của stream có tính năng tăng cường với đối số là đối tượng stream có tính năng cơ bản, nhằm tăng tính năng của stream cơ bản. Thường các stream được lồng nhau cho đến khi nhận được stream có phương thức phù hợp với nhu cầu sử dụng.

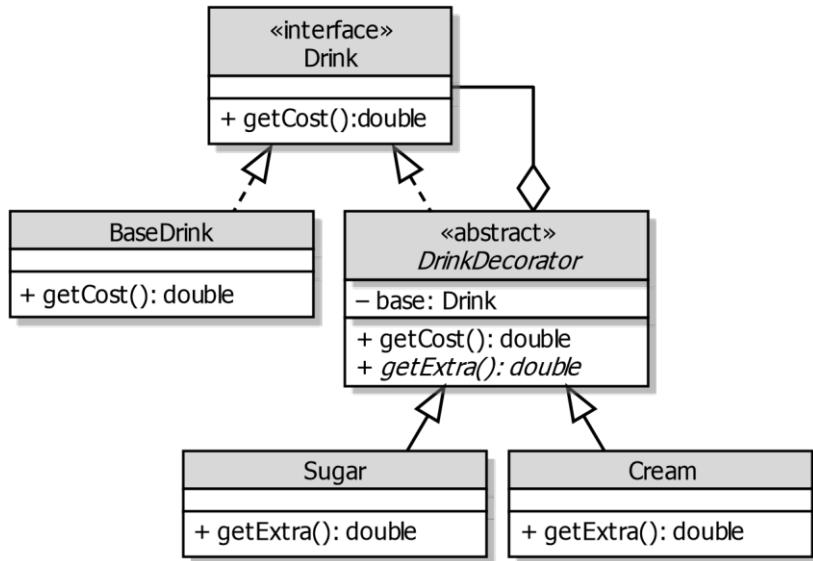
### 4. Sử dụng Ta

có:

- Lớp Component nhưng không dễ thừa kế nó. Ta muốn:
- Bổ sung thuộc tính hoặc hành vi cho một đối tượng một cách động.
- Thay đổi một số đối tượng trong một lớp mà không ảnh hưởng đến đối tượng khác.
- Tránh thừa kế vì có thể dẫn đến có quá nhiều lớp. Xem xét sử dụng:
- Adapter, cho phép thiết lập một giao diện giữa các lớp khác nhau.
- Composite, tạo đối tượng phức hợp mà cũng không cần mở rộng giao diện.
- Proxy, cho phép điều khiển truy cập đến đối tượng nó đại diện.
- Strategy, làm thay đổi đối tượng mà không cần "bao bọc" nó.

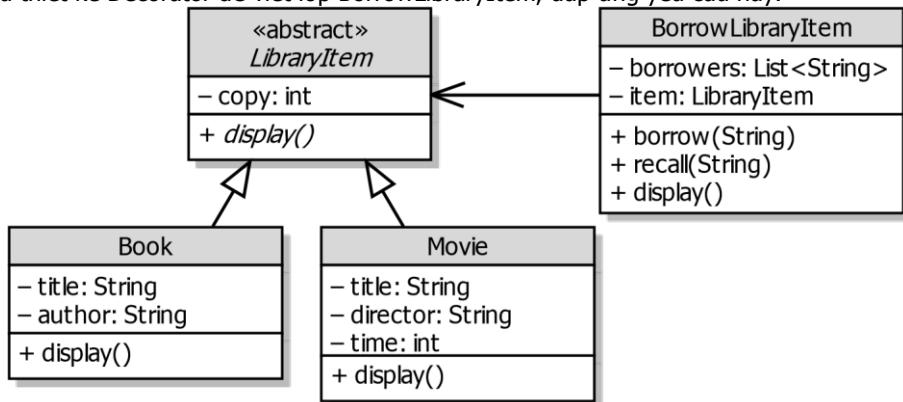
### 5. Bài tập

- Cửa hàng cung cấp thức uống (BaseDrink) với giá cơ bản là \$1.0. Khách hàng có thể tùy chọn thêm đường (Sugar) với phụ phí \$0.5, thêm kem (Cream) với phụ phí \$0.25, hoặc thêm cả hai. Tổng giá được tính tùy theo "thiết kế" thức uống của khách. Bạn hãy áp dụng mẫu thiết kế Decorator, đáp ứng yêu cầu này.



b) Tài nguyên của thư viện (LibraryItem) gồm hai loại Book và Movie, để xây dựng chương trình quản lý thư viện, ta cần mở rộng lớp LibraryItem thành lớp BorrowLibraryItem, thêm vào đó danh sách độc giả mượn tài nguyên và thêm hai phương thức mới: borrow (cho mượn), recall (thu hồi).

Bạn hãy áp dụng mẫu thiết kế Decorator để viết lớp BorrowLibraryItem, đáp ứng yêu cầu này.



c) Ban đầu logic nghiệp vụ của tập tin văn bản chỉ là đọc (ReadData) và ghi (writeData) dữ liệu. Khách hàng muốn tăng cường thêm các hành vi này thành hai cấp: đầu tiên là có thể mã hóa/giải mã dữ liệu đọc/ghi (bằng Base64), tiếp theo là có thể nén/giảm nén dữ liệu đọc/ghi (bằng run-length). Thực hiện cấp xử lý dữ liệu nào trước cũng được. Hãy dùng mẫu thiết kế Decorator để thực hiện yêu cầu này.

## Facade

### Interface for subsystem

Mẫu thiết kế Facade (façade [fə'sa:d]: mặt tiền) cung cấp một giao diện chung đơn giản thay cho một nhóm các giao diện có trong một hệ thống con (subsystem<sup>4</sup>). Mẫu thiết kế Facade định nghĩa một giao diện cấp cao hơn để giúp cho người dùng có thể dễ dàng sử dụng hệ thống con này vì chỉ cần giao tiếp với một giao diện đơn giản.

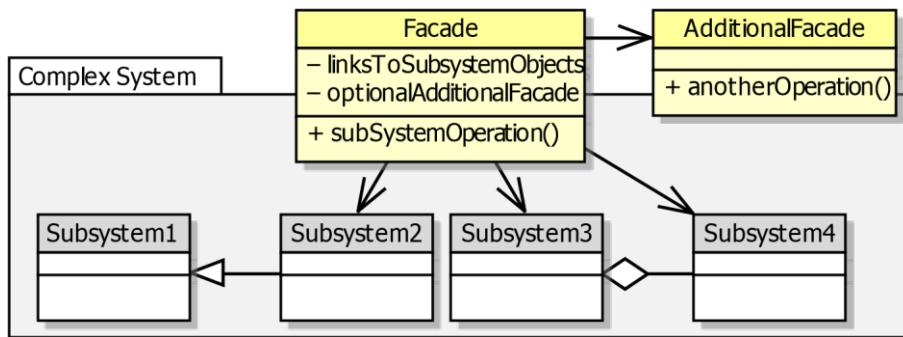
Mẫu thiết kế Facade cho phép các đối tượng truy cập vào hệ thống con bằng cách sử dụng giao diện chung này để giao tiếp với các giao diện có trong hệ thống con. Mục tiêu là che giấu các hoạt động phức tạp trong hệ thống con.

Việc sử dụng mẫu thiết kế Facade đem lại các lợi ích sau:

- Người dùng không cần biết đến sự phức tạp bên trong hệ thống con mà dễ dàng sử dụng hệ thống con vì chỉ giao tiếp với hệ thống con thông qua một giao diện chung đơn giản<sup>5</sup>.
- Chủ ý là mẫu thiết kế Facade không "đóng gói" (encapsulation) hệ thống con theo nghĩa hạn chế truy xuất; nó cung cấp một giao diện đơn giản trong lúc vẫn bộc lộ đầy đủ các chức năng của hệ thống con. Người dùng cao cấp vẫn có thể truy xuất trực tiếp vào sâu bên trong hệ thống con khi cần thiết, chẳng hạn để sửa chữa nâng cấp hệ thống con.
- Nâng cao khả năng độc lập của hệ thống con do cho phép nâng cấp đơn thể trong hệ thống con mà không cần phải sửa lại mã lệnh từ phía người dùng.
- Giúp phân lớp (layer) hệ thống con và phân nhóm sự phụ thuộc của các đối tượng trong hệ thống con.

## 1. Cài đặt

<sup>4</sup> Subsystem là nhóm các lớp, hoặc nhóm các lớp với các subsystem khác; chúng cộng tác với nhau để thực hiện một số tác vụ. <sup>5</sup> Mỗi hệ thống con có thể có nhiều giao diện Facade.



- Facade: biết rõ lớp của hệ thống con nào đảm nhận việc đáp ứng yêu cầu của Client, sẽ ủy nhiệm việc thực hiện đến các đối tượng của hệ thống con tương ứng. Đôi khi đối tượng Façade được gọi là "God Object", nó biết cách điều hành các phần của hệ thống để thực hiện yêu cầu của Client.

- AdditionalFacade: lớp tùy chọn này có thể được bổ sung để ngăn chặn sự trộn lẫn lớp Facade với các tính năng không liên quan, làm cho nó trở thành một cấu trúc phức tạp khác. Lớp này có thể được dùng bởi cả Client và các Facade khác.

- Các lớp Subsystem: cài đặt các chức năng của hệ thống con, phối hợp xử lý các công việc được Facade bộc lộ ra bên ngoài. Các lớp này không cần biết Facade và không tham chiếu đến nó.

Hai hệ thống con của hai lớp Facade khác nhau có thể có những lớp chung.

- Client sử dụng Facade thay vì gọi trực tiếp các đối tượng Subsystem. **Các bước thực hiện**

- Tạo lớp sẽ phục vụ như Facade.
- Xác định tất cả "use case" (task) mà hệ thống con sẽ cung cấp.
- Trong Facade, cài đặt các phương thức thực hiện các "use case". Các phương thức này phối hợp làm việc với các lớp hoặc giao diện có trong hệ thống con.

Nếu Facade trở nên quá lớn, xem xét tách một phần hành vi của nó vào một lớp Facade mới, tinh chỉnh hơn.

```
import java.util.HashMap; import
java.util.LinkedList; import
java.util.List;
```

```
// Subsystem class
Student { int
code; String
name;
Student(int code, String name) {
this.code = code; this.name =
name;
}

@Override public String toString() {
return String.format("[%d] %s", code, name);
}
} class
Course { int
code; String
name;
Course(int code, String name) {
this.code = code;
this.name = name;
}
} class
Section {
String name;
Course course;
List<Student> students = new LinkedList<>();
Section(String name) { this.name = name; }
void addStudent(Student student) { students.add(student); }
List<Student> getStudents() { return students; }
} class Campus { static HashMap<Integer, Student>
students = new HashMap(); static HashMap<Integer, Course>
courses = new HashMap();
static void setStudent(int studentCode, String studentName) {
students.put(studentCode, new Student(studentCode, studentName));
} static Student getStudent(int
studentCode) { return
students.get(studentCode);
}
```

```

 static void setCourse(int courseCode, String courseName) {
 courses.put(courseCode, new Course(courseCode, courseName));
 }
 static Course getCourse(int courseCode) {
 return courses.get(courseCode);
 }
}

// Facade class
Facade {
 static HashMap<String, Section> sections = new HashMap<>();
 public static void buildCampus() {
 Campus.setCourse(1000, "Operating System");
 Campus.setCourse(2000, "Core Java");
 Campus.setStudent(100, "Bill Gates");
 Campus.setStudent(101, "James Gosling");
 Campus.setStudent(102, "Linus Tovarld");
 }
 public static void buildSection(String sectionName, int courseCode) {
 Section section = new Section(sectionName);
 section.course = Campus.getCourse(courseCode);
 sections.put(sectionName, section);
 }
 public void enroll(String sectionName, int... studentCode) {
 for (int code : studentCode)
 sections.get(sectionName).addStudent(Campus.getStudent(code));
 }
 public void display(String sectionName) {
 Section section = sections.get(sectionName);
 String sName = section.name;
 String cName = section.course.name;
 List<Student> students = section.getStudents();
 System.out.printf("Course Name: %s [%s]\n", cName, sName);
 System.out.println(students.stream()
 .map(s -> "\n " + s).reduce("Student List: ", String::concat));
 }
}
public class Client {
 public static void main(String[] args) {
 System.out.println("----");
 Facade Pattern ----");
 Facade facade = new Facade();
 Facade.buildCampus();
 Facade.buildSection("OS1000", 1000);
 Facade.buildSection("CJ2000", 2000);
 facade.enroll("OS1000", 100, 102);
 facade.enroll("CJ2000", 101, 100);
 facade.display("OS1000");
 facade.display("CJ2000");
 }
}

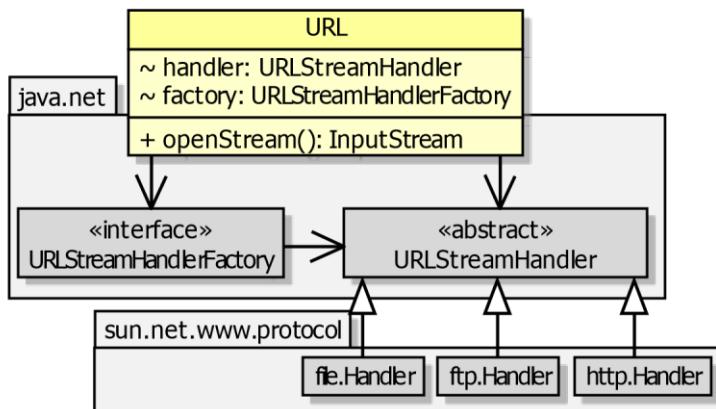
```

## 2. Liên quan

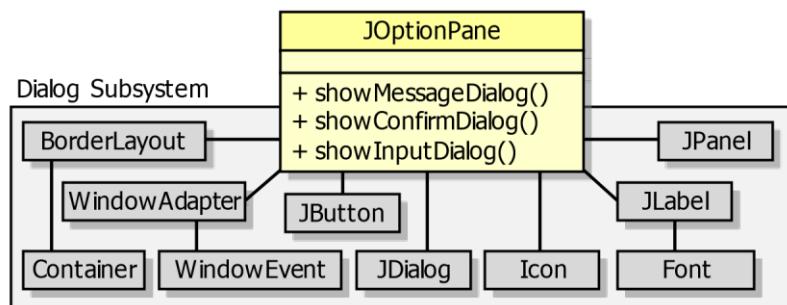
- Abstract Factory: thường dùng để tạo giao diện cho một hệ thống con một cách độc lập, có thể dùng như một Facade.
- Singleton: đối tượng Facade thường là một Singleton vì chỉ cần một đối tượng Facade.
- Mediator: tương tự như Facade, "bao bọc" một hệ thống con làm cho nó dễ sử dụng hơn. Nhưng Facade không định nghĩa chức năng mới cho hệ thống con, Facade chỉ giúp đơn giản hóa việc truy cập vào các lớp ủy nhiệm. Lớp Facade cũng không được các lớp của hệ thống con biết đến.
- Adapter: Facade có thể được xem như Adapter cho một hệ thống con. Trong trường hợp lời gọi đến hệ thống con quá phức tạp hoặc không phù hợp, lớp Facade "bao bọc" toàn bộ hệ thống con và cung cấp giao diện đơn giản hơn.

## 3. Java API

JDBC trong Java là một ví dụ tốt cho mẫu thiết kế Facade, nó đem đến một khung công việc (framework) dễ dùng khi truy xuất cơ sở dữ liệu. Trong JavaEE, mẫu thiết kế Session Facade (lớp Business) được phát triển từ mẫu thiết kế Facade. java.net.URL là một ví dụ của Facade, cung cấp phương thức đơn giản openStream() cho người dùng cuối.



JOptionPane là một lớp Facade, nó đơn giản hóa việc truy cập hệ thống các lớp giúp hiển thị các loại hộp thoại khác nhau.



#### 4. Sử dụng Ta có:

- Các hệ thống con có phần trừu tượng và phần hiện thực liên kết chặt chẽ.
- Hệ thống tiền hóa và trở nên phức tạp hơn, nhưng vẫn phải giữ giao diện giao tiếp đơn giản. Ta muốn:
- Phân cấp giao diện cho người dùng, người dùng phân quyền khác nhau có giao diện khác nhau.
- Phân lớp hệ thống với mỗi lớp có điểm nhập xác định.
- Che giấu các tác vụ của hệ thống con, chỉ có thể gọi chúng thông qua giao diện Facade.

#### 5. Bài tập

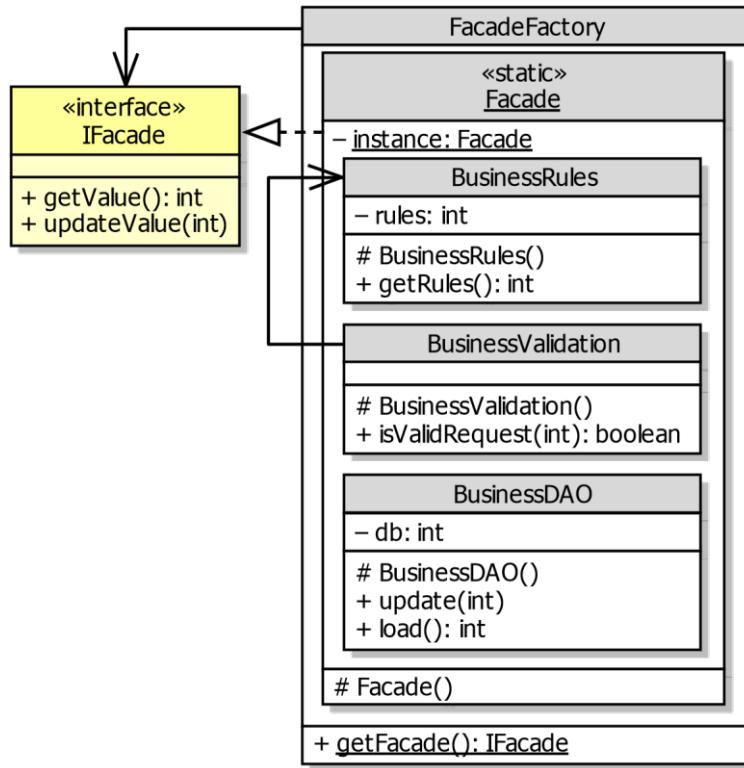
a) Tác vụ lưu một trị vào cơ sở dữ liệu được thực hiện phức tạp với các hệ thống con:

- Từ lớp BusinessRules, lấy trị rules bằng phương thức getRules().
- Trong lớp BusinessValidation, kiểm tra tính hợp lệ của trị value muốn lưu bằng phương thức isValidRequest(). Giả sử luật dùng lưu trị đơn giản: value < rules.
- Nếu trị muốn lưu hợp lệ, dùng phương thức update() của lớp BusinessDAO, lưu trị vào cơ sở dữ liệu. Có thể thực hiện đơn giản bằng cách lưu vào trị db của BusinessDAO.

Tác vụ lấy một trị từ cơ sở dữ liệu (từ trị db) phải gọi phương thức load() của lớp BusinessDAO.

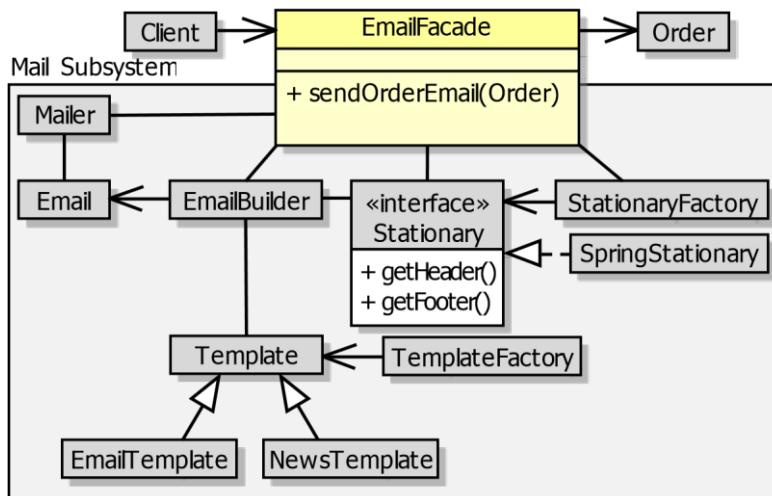
Để cung cấp giao diện đơn giản cho người dùng, áp dụng mẫu thiết kế Facade:

- Dùng lớp nội (inner class) để che giấu các hệ thống con: các lớp BusinessRules, BusinessValidation và BusinessDAO là lớp nội của Facade; Facade là lớp nội của FacadeFactory.
- Phương thức factory getFacade() sẽ cung cấp đối tượng Singleton Facade, cài đặt giao diện Ifacade cho người dùng. Hãy hiện thực yêu cầu trên.



b) Hệ thống con tạo và gửi email được mô tả như sau:

- Email được cấu hình và tạo từ EmailBuilder. EmailBuilder tạo Email theo:
  - + Template, có hai loại template là email và newsletter.
  - + Stationary (văn phòng phẩm), hiện có một loại là văn phòng phẩm có trang trí (SpringStationary).
  - Template và Stationary có nhiều loại khác nhau nên chúng được tạo từ các lớp factory: TemplateFactory và StationaryFactory.
  - Mailer dùng để gửi mail.
- Để người dùng sử dụng hệ thống con một cách đơn giản, mẫu thiết kế Facade được áp dụng. Người dùng chỉ đơn giản gọi phương thức sendOrderEmail() với Order cần gửi. hãy cài đặt hệ thống con và EmailFacade.



## Flyweight

Efficiently share objects

Trạng thái của một đối tượng có thể chứa một trong hoặc cả hai loại<sup>5</sup> sau:

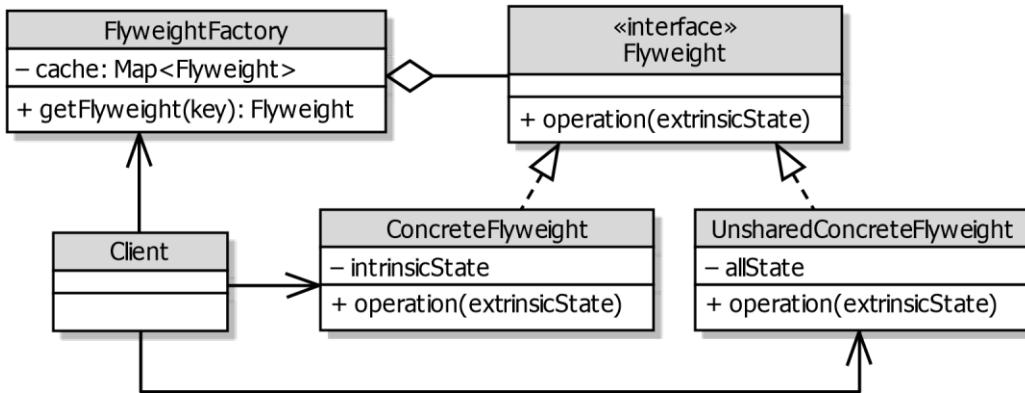
- Trạng thái bên trong (intrinsic): độc lập với ngữ cảnh (context-free) của đối tượng, trạng thái chung (sharing) cho tất cả (hoặc một nhóm) đối tượng của một lớp. Ví dụ, thông tin về công ty trên danh thiếp của tất cả nhân viên thuộc công ty là giống nhau.
- Trạng thái bên ngoài (extrinsic) là phụ thuộc và biến đổi theo ngữ cảnh của đối tượng, trạng thái đặc thù cho từng đối tượng thuộc lớp. Ví dụ, tên và số điện thoại của nhân viên trên danh thiếp là khác nhau dù họ chung một công ty.

Mẫu thiết kế Flyweight đề nghị tách trạng thái intrinsic và đóng gói vào một đối tượng riêng gọi là Flyweight. Nhóm với số lượng lớn các đối tượng có thể dùng chung một đối tượng Flyweight này để thể hiện phần trạng thái intrinsic của chúng, dẫn đến thu giảm yêu cầu lưu trữ.

Như vậy, trạng thái intrinsic được lấy từ thực thể dùng chung, trạng thái extrinsic được truyền như tham số.

<sup>5</sup> Có tài liệu phân thành ba loại: unshared (tương đương extrinsic), intrinsic, extrinsic (trạng thái được tính trong thời gian chạy).

## 1. Cài đặt



- FlyweightFactory: bào đảm tạo và lưu trữ các Flyweight khác nhau. FlyweightFactory được thiết kế như một Singleton. Khi Client cần đến thực thể Flyweight, nó gọi phương thức getFlyweight() của FlyweightFactory với tham số là kiểu của Flyweight yêu cầu.
- Flyweight: giao diện giúp ConcreteFlyweight có thể thao tác với trạng thái extrinsic.
- ConcreteFlyweight: chứa trạng thái intrinsic của đối tượng, thường được thiết kế như lớp nội (inner class) của FlyweightFactory với constructor là private để ngăn Client tạo trực tiếp đối tượng Flyweight.
- UnsharedConcreteFlyweight: không phải lớp cài đặt nào của Flyweight cũng đều dùng chung, lớp không dùng chung này bổ sung cho cây dẫn xuất từ Flyweight. Thường hiếm cài đặt.
- Từ góc nhìn của Client, Flyweight là một khuôn mẫu để sinh đối tượng "nhẹ", sau đó đối tượng được cấu hình bổ sung trong thời gian chạy bằng cách truyền cho chúng dữ liệu thay đổi theo ngữ cảnh. **Các bước thực hiện**
- Xác định các trạng thái intrinsic chung và trạng thái extrinsic đặc thù của đối tượng.
- Tạo interface Flyweight cung cấp phương thức operation() truy cập đến trạng thái extrinsic.
- Cài đặt ConcreteFlyweight dùng chung chứa các trạng thái intrinsic, phương thức operation() truy cập đến trạng thái extrinsic và dùng trạng thái intrinsic.
- Cài đặt UnsharedFlyweight các trạng thái của đối tượng không dùng chung, phương thức operation() bỏ qua không dùng trạng thái extrinsic được truyền cho nó.
- Cài đặt FlyweightFactory lưu giữ (cache) các Flyweight và cung cấp phương thức lấy chúng.

```

import java.util.HashMap; import
java.util.Map;

// Flyweight
interface Letter {
String getValue();
}

// FlyweightFactory class
LetterFactory {
 public static LetterFactory factory;
 public static Map<String, Letter> list = new HashMap<>();

 private LetterFactory() { }

 public static synchronized LetterFactory getFactory() {
if (factory == null) factory = new LetterFactory();
return factory;
 }
 public Letter getLetter(String key) {
if (!list.containsKey(key)) {
list.put(key, new Char(key));
 return list.get(key);
}
}

// ConcreteFlyweight
class Char implements Letter {
String value;
 private Char(String value) {
 System.out.println("create letter: " + value);
this.value = value;
 }

 @Override public String getValue() {
return value;
}
}

```

```

 }
}

class WordProcessor {
 StringBuilder sb = new StringBuilder();
 public void add(Letter letter) {
 sb.append(letter.getValue());
 }
 public void print() {
 System.out.println(sb.toString());
 }
}
public class Client {
 public static void main(String[] args) {
 System.out.println("--- Flyweight Pattern ---");
 WordProcessor processor = new WordProcessor();
 String s = "Google is good"; for (int i = 0; i
< s.length(); ++i) { String value =
s.substring(i, i + 1);
 processor.add(LetterFactory.getFactory().getLetter(value));
 }
 processor.print();
} }

```

WordProcessor làm việc không hiệu quả với nhiều Letter. Bạn có thể thấy điều này nếu đưa lớp Char ra ngoài và viết lại lệnh: `processor.add(new Char(value))`

Các Letter có trạng thái intrinsic là value, có thể dùng chung nếu chúng có value giống nhau. Đóng gói trạng thái intrinsic này vào đối tượng Flyweight để giảm số đối tượng cần tạo. Trạng thái extrinsic, ví dụ số lần ký tự đó xuất hiện, không mô tả ở đây.

## 2. Liên quan

- Composite: nên cài đặt Flyweight kết hợp Composite để tạo nên các cấu trúc phân cấp có các node dùng chung.
- State và Strategy: nên cài đặt các đối tượng State và Strategy như là các Flyweight.

## 3. Java API

`java.lang.String`.

Các lớp wrapper như `java.lang.Integer`, `Short`, `Byte`, ... phương thức `static valueOf()` được dùng như một phương thức factory.

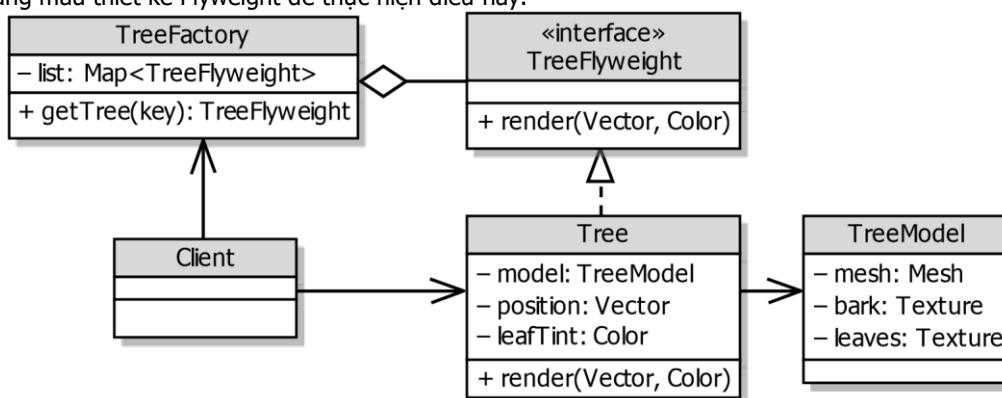
## 4. Sử dụng Ta

có:

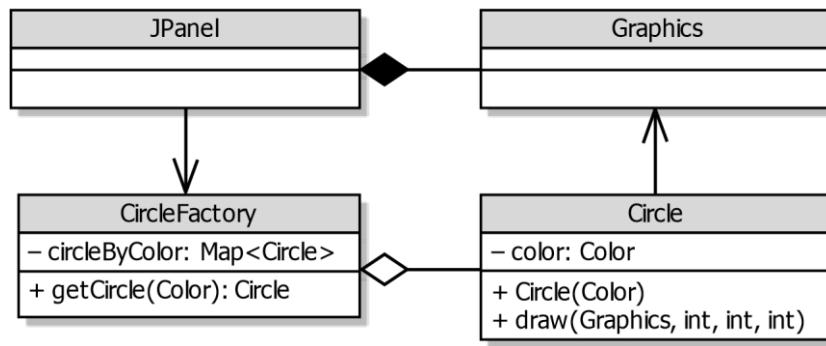
- Chương trình sử dụng một số lớn đối tượng trong bộ nhớ. Cần giảm số đối tượng này.
- Nhóm các đối tượng có một số trạng thái dùng chung. Ta muốn:
- Cài đặt một hệ thống mà việc sử dụng bộ nhớ bị hạn chế.

## 5. Bài tập

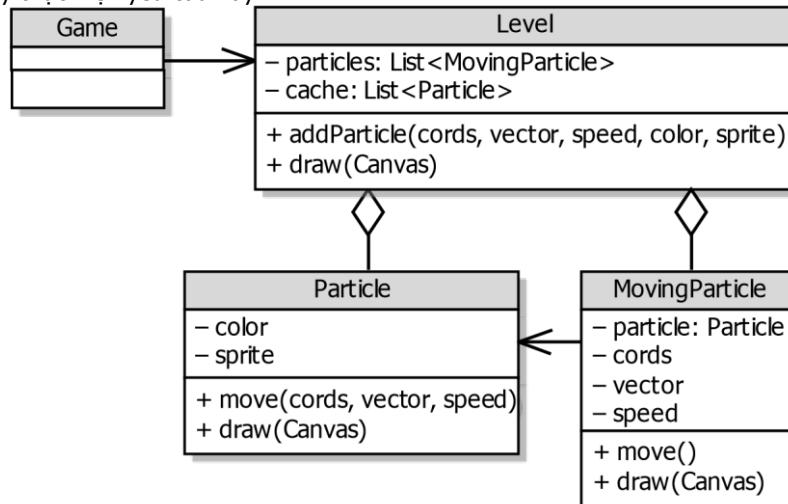
- a) Mẫu thiết kế Flyweight được xem là mẫu thiết kế GoF duy nhất được hỗ trợ phần cứng thực tế. Một đối tượng trong game thường chia làm phần model (intrinsic state, dùng chung) và phần tham số hóa (extrinsic state, thay đổi với từng cá thể). Card đồ họa và API đồ họa cho phép sử dụng một mô hình chung để hiển thị (render) nhiều thực thể, gọi là `instanced rendering`. Ví dụ chỉ dùng một mô hình cây `TreeModel` để hiển thị hàng ngàn cây Tree khác nhau về vị trí và màu sắc trong một khu rừng. Áp dụng mẫu thiết kế Flyweight để thực hiện điều này.



- b) Viết chương trình vẽ 1000 hình tròn (Circle) lên một JPanel. Các hình tròn có tâm và bán kính khác nhau, được vẽ bằng 6 màu: red, blue, yellow, orange, black và white. Nếu áp dụng mẫu thiết kế Flyweight, thay vì phải tạo 1000 đối tượng `Circle`, `CircleFactory` chỉ tạo 6 đối tượng `Circle`. Hãy thực hiện yêu cầu này.



c) Trong một Level của Game, một số lượng lớn các Particle (đạn, tên lửa, cầu lửa) được tạo ra. Các Particle có tọa độ (cords), hướng di chuyển (vector) và tốc độ (speed) khác nhau. Mỗi loại Particle dùng các sprite (hình ảnh động) giống nhau với bộ màu giới hạn. Lưu ý là sprite và màu của Particle tiêu thụ nhiều bộ nhớ nhưng được sử dụng lặp đi lặp lại, trong lúc các đặc tính khác thay đổi theo ngữ cảnh của Level. Vì vậy đội phát triển quyết định dùng mẫu thiết kế Flyweight để cải thiện hiệu năng của chương trình. Hãy thực hiện yêu cầu này.



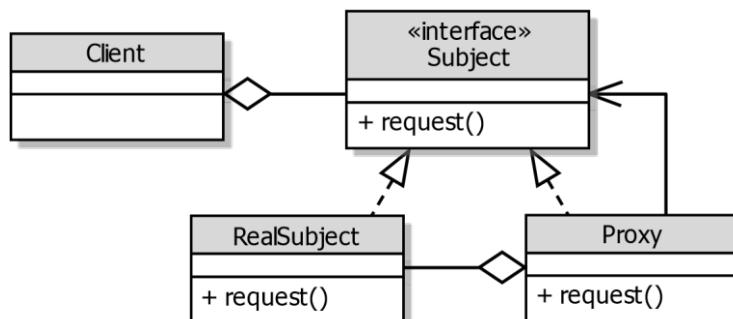
## Proxy

Placeholder for objects

Mẫu thiết kế Proxy cung cấp một đối tượng đại diện (Proxy) điều khiển việc tạo ra và truy cập đến một đối tượng thực khác (RealSubject). Proxy thường là một đối tượng nhỏ đúng chẩn trước đối tượng thực phức tạp hơn, Proxy chỉ kích hoạt đối tượng thực phức tạp này sau khi đạt được một số điều kiện nhất định.

Ý tưởng của mẫu thiết kế Proxy là cung cấp một đối tượng thay thế (surrogate) hoặc giữ chỗ (placeholder) cho một đối tượng thực. Đối tượng thực có thể chưa có sẵn (nằm trên máy ở xa, chưa nạp vào, đã chuyển ra đĩa (swap out)) hoặc cần kiểm soát được việc truy cập vào nó (authentication, audit, preprocessing, logging, transaction context setup). Nói cách khác, con đường truy cập đến đối tượng thực (RealSubject) buộc phải thông qua Proxy.

### 1. Cài đặt



- Subject: giao diện chung cho RealSubject lẫn Proxy, để Proxy có mặt mọi nơi mà RealSubject được quan tâm, đại diện được cho RealSubject. Client không nhận thấy sự tồn tại của Proxy. Tùy nhiệm vụ, có nhiều loại Proxy:

- + Virtual proxy: xử lý việc tạo RealSubject dựa trên thông tin khác (ví dụ từ tên tập tin), vì vậy có trì hoãn (lazy loading) khi tạo RealSubject.
- + Protective proxy: khi gọi yêu cầu, phải vượt qua xác thực rồi mới tạo RealSubject.
- + Remote proxy: còn gọi là Stub. Stub sẽ mã hóa yêu cầu tạo đối tượng và gửi chúng qua mạng, RealSubject sẽ được tạo trong một không gian địa chỉ khác.
- + Smart proxy: thêm yêu cầu, thay đổi yêu cầu, đơn giản hóa yêu cầu trước khi gửi yêu cầu để tạo hoặc truy cập đối tượng.
- RealSubject: lớp mà Proxy sẽ đại diện.

- Proxy: lớp được dùng để tạo ra, điều khiển, tăng cường, xác thực truy cập đến RealSubject. Proxy giữ một tham chiếu đến RealSubject để ủy nhiệm các lời gọi cho RealSubject khi đạt điều kiện gọi.
- Client làm việc với cả RealSubject và Proxy thông qua interface Subject. Vì vậy, bạn có thể truyền Proxy vào bất kỳ code nào tham chiếu đến một RealObject.

### Các bước thực hiện

#### Cài đặt Proxy

- Proxy có cùng interface Subject với RealSubject. Nói cách khác, với Client, Proxy và RealSubject có thể hoán chuyển cho nhau.
- Giữ tham chiếu đến Subject, có thể tạo RealSubject sau trong constructor của Proxy.
- Phương thức của Proxy cài đặt các chức năng của Proxy, trước khi ủy nhiệm đến RealSubject trong nó.

```
// Subject
interface Subject {
String request();
}

// RealSubject class RealSubject
implements Subject { @Override
public String request() { return
"RealSubject::request()";
}
}
//
Proxy
class VirtualProxy implements Subject {
Subject subject = null;
@Override public String request() {
if (subject == null) {
System.out.println("Virtual Proxy: Subject inactive");
subject = new RealSubject();
}
System.out.println("Virtual Proxy: Subject active");
return "Virtual Proxy: " + subject.request();
}
}
class ProtectionProxy implements Subject {
Subject subject = null;
private final String PASSWORD = "s3kr3t";
public String authenticate(String password) {
if (!password.equals(PASSWORD)) {
return "Protection Proxy: Access denied";
} else {
subject = new RealSubject();
return "Protection Proxy: Access granted";
}
}
@Override public String request() {
if (subject == null) {
return "Protection Proxy: Authenticate first";
} else {
return "Protection Proxy: " + subject.request();
}
}
}

public class Client { public static void
main(String[] args) { System.out.println("--"
- Proxy Pattern ---");
Subject subject = new VirtualProxy();
System.out.println(subject.request());
System.out.println();
Subject pSubject = new ProtectionProxy();
System.out.println(pSubject.request());
System.out.println(((ProtectionProxy)pSubject).authenticate("password"));
System.out.println(((ProtectionProxy)pSubject).authenticate("s3kr3t"));
System.out.println(pSubject.request());
}
}
}
```

## 2. Liên quan

- Adapter: Adapter cung cấp một giao diện khác cho đối tượng mà nó tiếp hợp. Trong lúc Proxy cung cấp giao diện giống với Subject mà nó đại diện. Nhiệm vụ hai mẫu thiết kế này khác nhau.
- Decorator: Proxy cài đặt giống như Decorator nhưng có nhiệm vụ khác.

## 3. Java API

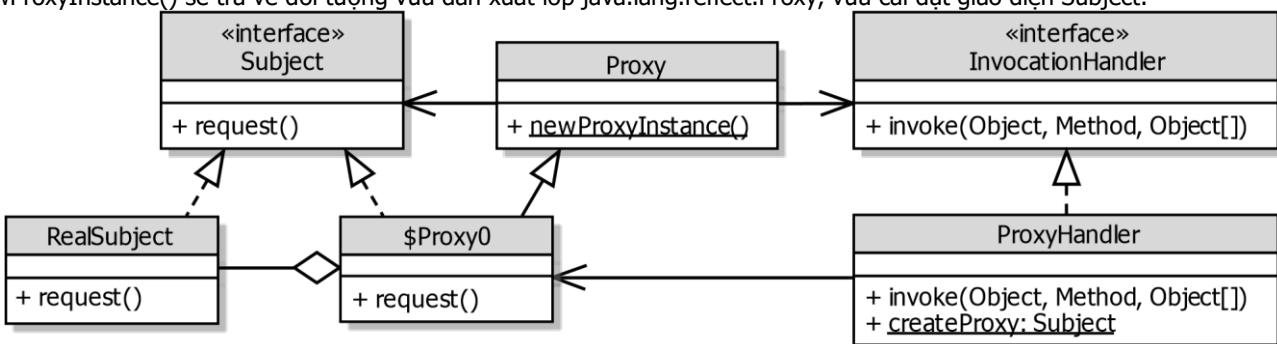
Mẫu thiết kế Proxy đã được tích hợp vào Java API giúp người dùng dễ dàng tạo một Proxy cho đối tượng RealSubject cần kiểm soát truy cập. Để áp dụng mẫu thiết kế này, bạn cần tạo một lớp xử lý cho proxy (ProxyHandler):

- Trong lớp này, khi đạt điều kiện tạo đối tượng, tạo đối tượng RealSubject để ủy nhiệm các lời gọi cho RealSubject.
- Phương thức invoke(): cài đặt từ giao diện java.lang.reflect.InvocationHandler. Trong phương thức này, chọn phương thức gọi và ủy nhiệm lời gọi phương thức cho đối tượng RealSubject thực hiện.

```
Object invoke(Object proxy, Method method, Object[] args); proxy: đối tượng proxy do
Proxy.newProxyInstance() trả về. method: đóng gói phương thức triệu gọi, dùng phương thức getName()
để xác định phương thức cần gọi.
args: đối số của phương thức triệu gọi, định nghĩa tại giao diện Subject.
```

- Phương thức createProxy(): tạo đối tượng proxy bằng phương thức static newProxyInstance() của lớp Proxy. Các đối số của phương thức newProxyInstance() dùng cơ chế reflection của Java để lấy thông tin từ Subject, giao diện của đối tượng RealSubject mà proxy đại diện. static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler handler); loader: bộ nạp lớp của Subject, Subject.class.getClassLoader() hoặc subject.getClass().getClassLoader(). interfaces: interface của Subject, new Class<?>[] { Subject.class } hoặc subject.getClass().getInterfaces(). handler: lớp xử lý new ProxyHandler().

newProxyInstance() sẽ trả về đối tượng vừa dẫn xuất lớp java.lang.reflect.Proxy, vừa cài đặt giao diện Subject.



```

import java.lang.reflect.InvocationHandler; import
java.lang.reflect.InvocationTargetException; import
java.lang.reflect.Method; import
java.lang.reflect.Proxy;

// Subject
interface Subject {
String request();
}

// RealSubject class RealSubject
implements Subject { @Override
public String request() { return
"RealSubject::request();";
}
}

// Proxy Handler
class ProxyHandler implements InvocationHandler {
private Subject subject = null; private final
String PASSWORD = "s3kr3t"; private
ProxyHandler(String password) { if
(!password.equals(PASSWORD)) {
System.out.println("Protection Proxy: Access denied");
} else {
subject = new RealSubject();
System.out.println("Protection Proxy: Access granted");
}
}
public static Subject createProxy(String password) {
return (Subject)Proxy.newProxyInstance(Subject.class.getClassLoader(),
new Class<?>[]{ Subject.class }, new ProxyHandler(password)); }
}

@Override

```

```

public Object invoke(Object proxy, Method method, Object[] args) throws IllegalAccessException {
try {
 if (method.getName().equals("request")) return method.invoke(subject, args);
else throw new IllegalAccessException(); } catch (InvocationTargetException e)
{ e.printStackTrace(System.err); }
return null;
}
}

public class Client {
public static void main(String[] args) {
System.out.println("--- Proxy Pattern in Java API ---");
Subject pSubject = ProxyHandler.createProxy("s13kr3t");
try {
 System.out.println(pSubject.request());
} catch (NullPointerException e) {
 System.err.println("Protection Proxy: Authenticate first");
}
}
}

```

#### 4. Sử dụng Ta

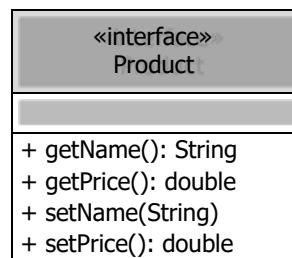
có:

- Đổi tượng tốn nhiều chi phí để tạo.
- Đổi tượng cần xác thực để truy cập.
- Đổi tượng truy cập từ xa.
- Đổi tượng cần thực hiện một số hành động trước khi truy cập. Ta muốn:
- Tạo đổi tượng chỉ khi có yêu cầu đến tác vụ của chúng.
- Thực hiện việc kiểm tra hoặc dọn dẹp khi truy cập đến đổi tượng.
- Một đổi tượng cục bộ truy cập đến một đổi tượng từ xa.
- Cài đặt các quyền truy cập lên đổi tượng khi yêu cầu đến các tác vụ của đổi tượng đó.

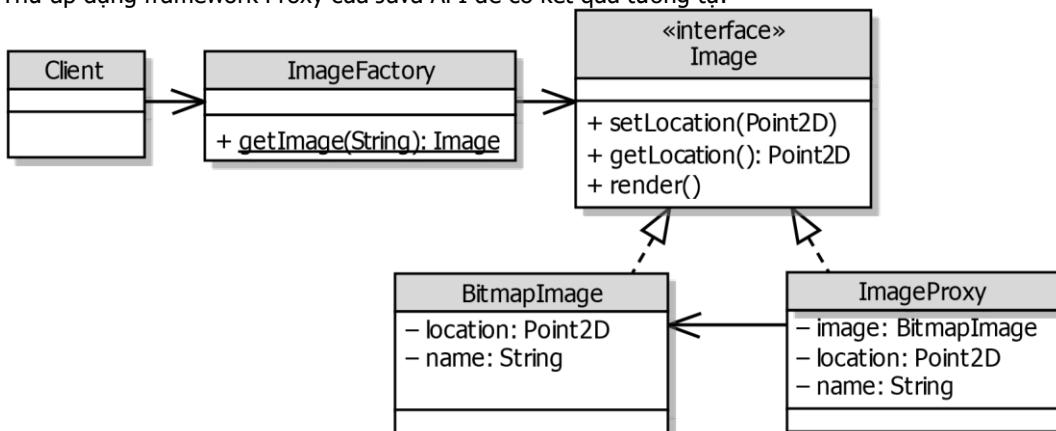
#### 5. Bài tập

a) Lớp RealProduct cài đặt giao diện Product như hình dưới. Áp dụng framework Proxy của Java API, tạo hai lớp xử lý:

- EmployeeHandler tạo đổi tượng proxy chỉ gọi được các phương thức getters của Product.
- ManagerHandler tạo đổi tượng proxy gọi được tất cả các phương thức của Product nhưng có ghi nhận vào tập tin log mỗi khi gọi phương thức.



b) Client cần ảnh cho chương trình sẽ lấy ảnh từ phương thức static getImage() của ImageFactory. Ảnh BitmapImage lấy từ tập tin trên đĩa để hiển thị (render) ra màn hình. Nếu ảnh không tồn tại, một ảnh mặc định sẽ được sử dụng thay thế. Ngoài ra, ảnh sẽ được xử lý (thêm khung, gắn watermark, xử lý grayscale, ...) trước khi hiển thị. Hãy áp dụng mẫu thiết kế Proxy để thực hiện yêu cầu này. Thử áp dụng framework Proxy của Java API để có kết quả tương tự.



## Behavioral Design Patterns

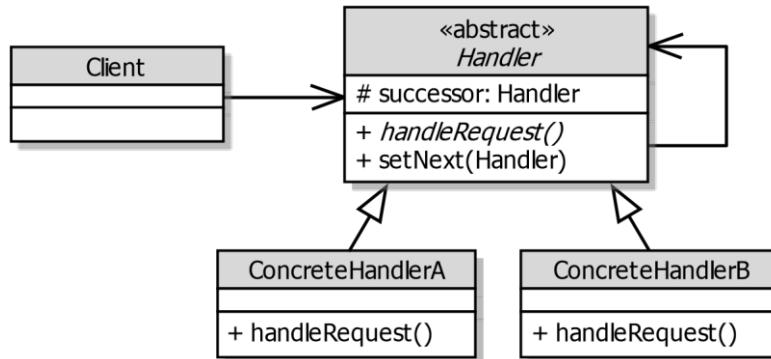
### Chain of Responsibility

Decouple sender and receiver

Mẫu thiết kế Chain of Responsibility giúp tránh kết nối trực tiếp, quá chặt giữa đối tượng gửi yêu cầu và đối tượng nhận yêu cầu, khi yêu cầu được xử lý bởi một hoặc nhiều đối tượng.

Mẫu thiết kế này tạo một dây chuyền (chain) các đối tượng nhận yêu cầu, rồi truyền yêu cầu theo dây chuyền đó. Các đối tượng nhận yêu cầu để xử lý (handler) có thể là toàn bộ dây chuyền, một phần dây chuyền hoặc chỉ là một đối tượng trong dây chuyền. Bằng cách này, đối tượng gửi yêu cầu không cần biết yêu cầu sẽ được xử lý bởi đối tượng nào. Một đối tượng trong chuỗi không cần biết toàn bộ hệ thống xử lý mà chỉ biết khâu tiếp theo. Do kết nối không quá chặt, chúng ta có thể sửa đổi và thêm vào chuỗi xử lý mà không phải viết lại phần lớn logic của ứng dụng. Các tên khác: CoR, Chain of Command.

#### 1. Cài đặt



- Handler: định nghĩa giao diện chung để xử lý yêu cầu. Chứa tham chiếu đến đối tượng xử lý yêu cầu ngay sau nó, gọi là successor. Phương thức setNext() dùng thiết lập đối tượng successor nếu cần chuyển tiếp xử lý yêu cầu.
- ConcreteHandler: xử lý yêu cầu với các mức độ khác nhau, nếu không xử lý yêu cầu nó có thể chuyển tiếp yêu cầu đến successor của nó. Các ConcreteHandler thường đóng kín và bất biến, nhận dữ liệu chỉ một lần thông qua constructor.
- Client có thể sắp xếp chuỗi xử lý. Lưu ý là một yêu cầu có thể được gửi đến bất kỳ Handler trong chuỗi, không nhất thiết phải là Handler đầu tiên.

#### Các bước thực hiện

- Định nghĩa lớp abstract hoặc interface Handler.
- + Định nghĩa phương thức handleRequest() tiếp nhận và xử lý yêu cầu.
- + Định nghĩa phương thức setNext() truy cập đến Handler kế tiếp trong chuỗi (successor). Nếu Handler là lớp abstract, có thể lưu successor như biến thực thể.
- Cài đặt một số ConcreteHandler, ConcreteHandler kiểm tra xem nó có tham gia xử lý yêu cầu không, nếu không nó chuyển tiếp yêu cầu cho successor.
- Chuỗi xử lý bao gồm các Handler kế tiếp nhau được thiết kế trong Client. Client truyền yêu cầu cho Handler đầu tiên này hoặc handler bất kỳ trong chuỗi.

```
// Handler
abstract class AbstractLogger {
 enum Level {
 INFO ("Information level") { @Override int getValue() { return 1; } },
 DEBUG ("Debug level") { @Override int getValue() { return 2; } },
 ERROR ("Error level") { @Override int getValue() { return 3; } };
 }
 private final String name;
 private Level(String name) { this.name = name; }
 public String getName() { return name; }
 abstract int getValue();
 protected Level level;
 protected AbstractLogger nextLogger = null;
 protected void write(String message);

 AbstractLogger setNextLogger(AbstractLogger nextLogger) {
 this.nextLogger = nextLogger;
 return this;
 }
 void logMessage(Level level) {
 if (this.level.getValue() <= level.getValue()) write(level.getName());
 if (nextLogger != null) nextLogger.logMessage(level);
 else System.out.println("--- end of chain ---");
 }
}

// ConcreteHandler
```

```

class ConsoleLogger extends AbstractLogger {
 ConsoleLogger(Level level) {
 this.level = level;
 }

 @Override protected void write(String message) {
 System.out.println("[Standard Console]: " + message);
 }
}

class ErrorLogger extends AbstractLogger {
 ErrorLogger(Level level) {
 this.level = level;
 }

 @Override protected void write(String message) {
 System.out.println("[Error Console]: " + message);
 }
}

class FileLogger extends AbstractLogger {
 FileLogger(Level level) {
 this.level = level;
 }

 @Override protected void write(String message) {
 System.out.println("[Log File]: " + message);
 }
}

public class Client {
 public static void main(String[] args) {
 System.out.println("--- Chain of Responsibility ---");
 AbstractLogger loggerChain = new ErrorLogger(AbstractLogger.Level.ERROR)
 .setNextLogger(new FileLogger(AbstractLogger.Level.DEBUG))
 .setNextLogger(new ConsoleLogger(AbstractLogger.Level.INFO)));
 loggerChain.logMessage(AbstractLogger.Level.INFO); loggerChain.logMessage(AbstractLogger.Level.DEBUG);
 loggerChain.logMessage(AbstractLogger.Level.ERROR);
 }
}

```

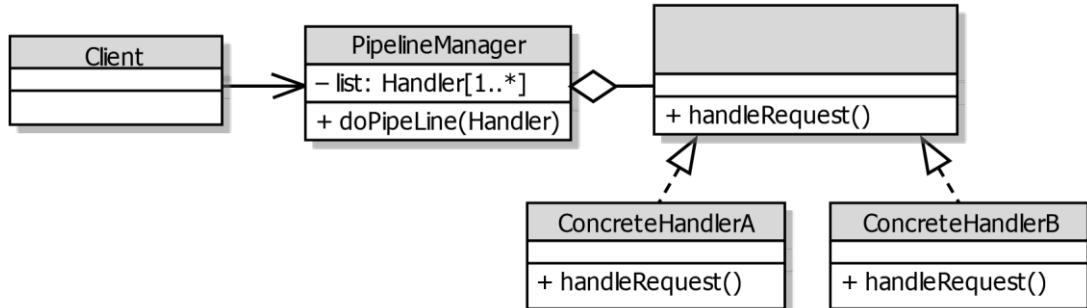
AbstractLogger định nghĩa chuỗi xử lý "đổ xuống": ERROR → DEBUG → INFO, tùy mức độ lỗi mà toàn chuỗi hoặc một phần chuỗi xử lý ghi nhận lỗi.

## 2. Liên quan

- Composite: mẫu thiết kế Chain of Responsibility thường kết hợp với mẫu thiết kế Composite, khi đó các thành phần con của một Composite được xem như successor của nó.
- Decorator: có cấu trúc giống Chain of Responsibility, truyền yêu cầu qua một cấu trúc đệ quy để thực hiện, cũng cho phép ngắt chuỗi xử lý. Tuy nhiên các Handler của Chain of Responsibility hoạt động độc lập so với các Decorator.
- Pipeline: (không thuộc GoF)

Lưu ý rằng khi cần thêm một đối tượng xử lý yêu cầu (Handler) vào dây chuyền, ta thao tác giống như thêm một node vào danh sách liên kết đơn: thiết lập lại successor của đối tượng xử lý ngay trước vị trí chèn vào dây chuyền và thiết lập successor của đối tượng xử lý mới chỉ đến đối tượng xử lý ngay sau vị trí chèn. Điều này làm cho mẫu thiết kế Chain of Responsibility trở nên kém linh hoạt, nhất là khi sử dụng với các framework trong đó danh sách các đối tượng xử lý được "tiêm" vào ứng dụng. Mẫu thiết kế Pipeline (còn gọi là Filter) giải quyết điều này. Với mẫu thiết kế này, PipelineManager sẽ quản lý dây chuyền xử lý dưới hình thức một danh sách có thứ tự các đối tượng xử lý, nó quyết định thứ tự sử dụng các đối tượng xử lý không cần đến successor. Thêm một đối tượng xử lý mới chỉ là chèn theo thứ tự vào danh sách này. PipelineManager cũng điều hành dây chuyền xử lý.

mỗi đối  
ly không  
phương  
của đối  
successor



lý, nghĩa là  
tương xử  
phải gọi  
thức xử lý  
tương  
của nó.  
«interface»

Ví dụ minh họa bên dưới là ví dụ minh họa cho mẫu thiết kế Chain of Responsibility được cấu trúc lại theo mẫu thiết kế Pipeline.

```

import java.util.LinkedList;
import java.util.List; import
java.util.stream.Collectors; import
java.util.stream.Stream;
// Handler
abstract class AbstractLogger {
 enum Level {
 INFO ("Information level") { @Override int getValue() { return 1; } },
 DEBUG ("Debug level") { @Override int getValue() { return 2; } },
 ERROR ("Error level") { @Override int getValue() { return 3; } };
 private final String name;
 private Level(String name) { this.name = name; }
 public String getName() { return name; } abstract
 int getValue();
 }
 protected Level level;
 abstract protected void write(String message);
}

// ConcreteHandler
class ConsoleLogger extends AbstractLogger {
 ConsoleLogger(Level level) {
this.level = level;
 }

@Override protected void write(String message) {
 System.out.println("[Standard Console]: " + message);
}
}
class ErrorLogger extends AbstractLogger {
 ErrorLogger(Level level) {
this.level = level;
 }

@Override protected void write(String message) {
 System.out.println("[Error Console]: " + message);
}
}
class FileLogger extends AbstractLogger {
 FileLogger(Level level) {
this.level = level;
 }

@Override
protected void write(String message) {
System.out.println("[Log File]: " + message);
}
}
}

// PipelineManager class
PipelineManager {
List<AbstractLogger> list = new LinkedList<>();

PipelineManager(AbstractLogger... handlers) {
list.addAll(Stream.of(handlers)
 .sorted((h1, h2) -> Integer.compare(h1.level.getValue(), h2.level.getValue()))
.collect(Collectors.toList()));
}
}

```

```

 void doPipeline(AbstractLogger.Level level) {
list.stream()
 .filter(h -> h.level.getValue() <= level.getValue())
 .forEach(h -> h.write(level.getName()));
 System.out.println("--- end of chain ---");
}
}

public class Client {
 public static void main(String[] args) {
System.out.println("--- Pipeline Pattern ---");
PipelineManager manager = new PipelineManager(
 new ErrorLogger(AbstractLogger.Level.ERROR),
 new FileLogger(AbstractLogger.Level.DEBUG),
 new ConsoleLogger(AbstractLogger.Level.INFO)
);
manager.doPipeline(AbstractLogger.Level.INFO);
manager.doPipeline(AbstractLogger.Level.DEBUG);
manager.doPipeline(AbstractLogger.Level.ERROR);
 }
}

```

Trong mẫu thiết kế Pipeline, Handler chỉ có một phương thức abstract nên thuận lợi để cài đặt thành biểu thức Lambda trong Java. Ví dụ sau minh họa điều này: nhân vật trong game được "xử lý" sau mỗi màn chơi, thứ tự xử lý là trang bị vũ khí (weapon), bảo vệ (shield) và tăng năng lượng (power); áp dụng mẫu thiết kế Pipeline để thực hiện yêu cầu.

```

import java.util.LinkedList; import
java.util.List;
import java.util.stream.Collectors; import
java.util.stream.Stream;

class Hero {
String weapon;
String shield;
int power;

 Hero setWeapon(String weapon) {
this.weapon = weapon; return
this;
 }

 Hero setShield(String shield) {
this.shield = shield; return
this;
 }

 Hero setPower(int power) {
this.power = power;
return this;
 }

 Hero(String weapon, String shield, int power) {
this.weapon = weapon; this.shield = shield;
this.power = power;
 }

 @Override public String toString() {
 return String.format("Hero (%s, %s, %d)", weapon, shield, power);
 }
}

// Handler
@FunctionalInterface interface
Handler<E> {
 E process(E input);
} class PipelineManager<E> implements
Handler<E> {
 List<Handler<E>> list = new LinkedList<E>();
 PipelineManager(Handler<E>... handlers) {
 list.addAll(Stream.of(handlers).collect(Collectors.toList()));
 }
}

```

```

@Override public E process(E input) {
 E e = input;
 for (Handler<E> handler : list) e = handler.process(e);
 return e;
}
}

public class Client {
 public static void main(String[] args) {
 System.out.println("--- Pipeline Pattern ---");
 Hero hero = new Hero("N/A", "N/A", 100);
 System.out.println(new PipelineManager<Hero>(
 h -> h.setWeapon("Sword"), h ->
 h.setShield("Armor"), h ->
 h.setPower(h.power + 50)
).process(hero));
 }
}

```

Lưu ý là Handler<E> giống với java.util.function.Function<E, E>, nghĩa là có thể viết gọn hơn:

```

import java.util.function.Function; // ...
public class Client { public static void
main(String[] args) { Hero hero = new
Hero("N/A", "N/A", 100);
 Function<Hero, Hero> f = h -> h.setWeapon("Sword");
 System.out.println(f
 .andThen(h -> h.setShield("Armor"))
 .andThen(h -> h.setPower(h.power + 50))
 .apply(hero)); }

```

Mẫu thiết kế Pipeline với các cấu trúc phức tạp hơn thường áp dụng trong thiết kế các ứng dụng song hành, tham khảo sách: Timothy G. Mattson , Beverly A. Sanders , Berna L. Massingill - Patterns for Parallel Programming - Addison-Wesley, 2004. ISBN 0321228111

### 3. Java API

Các bước xử lý ngoại lệ (exception) được thực hiện theo mẫu thiết kế Chain of Responsibility.

java.util.logging.Logger với phương thức xử lý log().

HttpServletRequest.RequestDispatcher() trong API của servlet/JSP với phương thức forward(). javax.servlet.Filter với phương thức xử lý doFilter().

### 4. Sử dụng Ta có:

- Nhiều hơn một đối tượng xử lý yêu cầu (handler), không chỉ định rõ đối tượng nào sẽ xử lý yêu cầu.
- Một đối tượng xử lý yêu cầu cần chuyển yêu cầu đến một đối tượng xử lý yêu cầu khác trong một dây chuyền.
- Tập hợp các đối tượng xử lý yêu cầu có thể thay đổi một cách động.

Ta muốn:

- Linh hoạt khi gán các yêu cầu để xử lý.

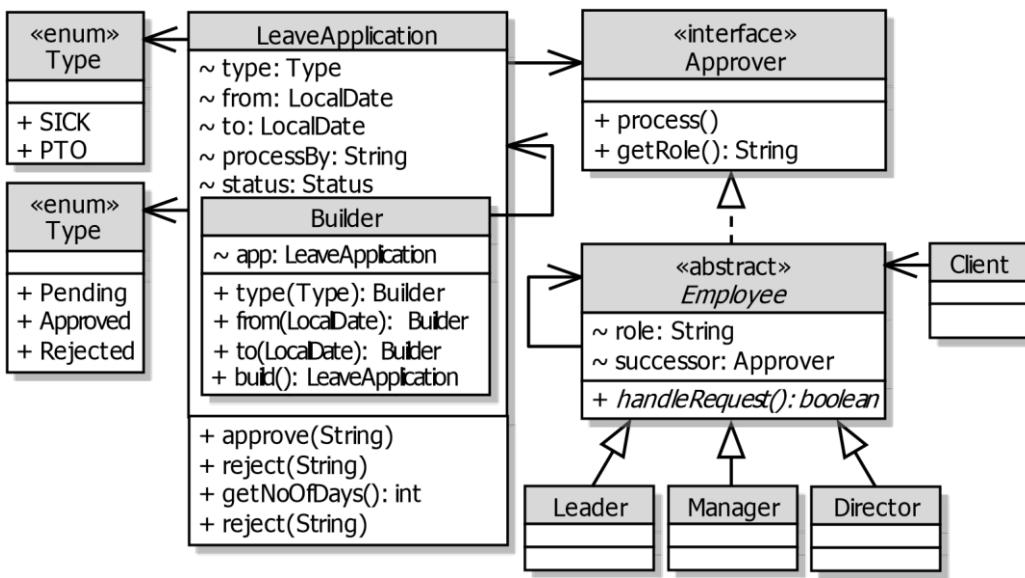
### 5. Bài tập

a) Đơn xin nghỉ phép (LeaveApplication) được tạo mẫu thiết kế Builder, được chấp thuận từ một Approver nhất định. Quy trình đưa đơn là từ Leader, nếu Leader không đủ quyền chấp thuận sẽ chuyển lên Manager. Nếu Manager không đủ quyền chấp thuận sẽ chuyển lên cấp cao nhất là Director.

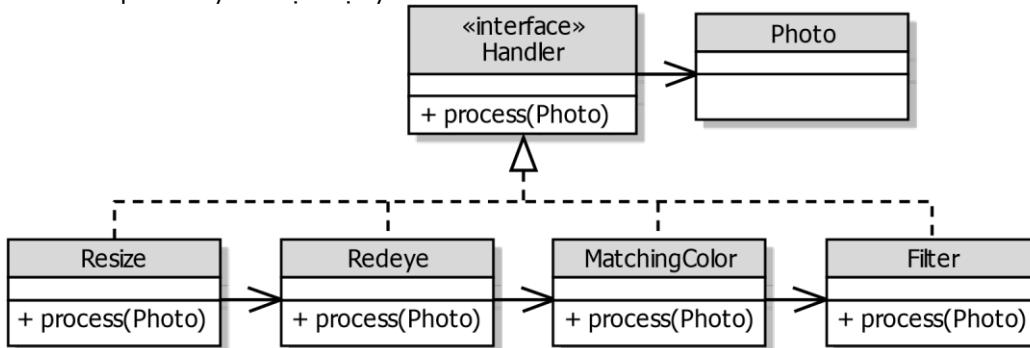
Quyền chấp thuận được xác định từ kiểu đơn (Type) và số ngày nghỉ:

- Nếu nghỉ bệnh (SICK), ít hơn 2 ngày, đơn phải được Leader chấp thuận.
- Nếu nghỉ bệnh, nhiều hơn 2 ngày hoặc nghỉ phép (PTO, Paid Time Off) ít hơn 5 ngày, đơn phải được Manager chấp thuận.
- Nếu nghỉ phép nhiều hơn 5 ngày, đơn phải được Director chấp thuận.

Hãy dùng mẫu thiết kế Chain of Responsibility để thực hiện yêu cầu trên.



b) Chương trình xử lý ảnh cho phép chuỗi xử lý ảnh tự động theo lô: Resize (hiệu chỉnh kích thước) □ Redeye (loại bỏ lỗ mắt đỏ) □ Color matching (hiệu chỉnh màu) □ Filter (áp dụng các bộ lọc định sẵn). Ảnh sẽ được tự động nhận dạng và xử lý tại mỗi khâu, khâu xử lý không cần thiết (ví dụ bỏ qua khâu Redeye nếu ảnh không có lỗ mắt đỏ) có thể được bỏ qua. Áp dụng mẫu thiết kế Chain of Responsibility để thực hiện yêu cầu trên.



c) Tiến trình xử lý checkout một đơn hàng gồm các bước: CalculatePayment (tính giá trị đơn hàng) và AddCustomerInfo (điền thông tin khách hàng vào đơn hàng). Một bước mới cần đưa vào là SaveOrderInfo (lưu thông tin đơn hàng vào tập tin log). Áp dụng mẫu thiết kế Chain of Responsibility để tiến trình có hai bước như lúc đầu, sau đó hiệu chỉnh để đưa bước thứ ba vào.

d) Khi tiến hành chuyển đổi một ứng dụng cũ, chương trình cần xử lý các User trong ứng dụng cũ theo các tầng xử lý sau:

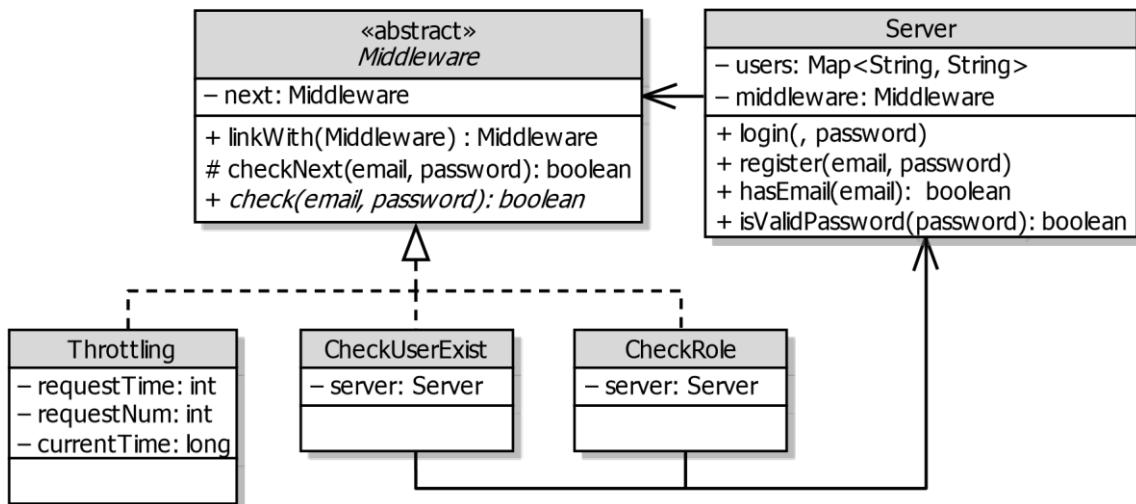
- Chuyển id của User thành một định dạng khác.
- Cấu trúc thực thể User thành thực thể `TransformedUser` theo đặc tả mới.
- Lưu các `TransformedUser` thành định dạng JSON.
- Lưu các `TransformedUser` vào bảng mới trong cơ sở dữ liệu. Hãy áp dụng mẫu thiết kế Pipeline để thực hiện yêu cầu này.

e) Người dùng muốn thực hiện một yêu cầu cần phải đăng nhập, cung cấp cặp email - password. Quá trình đăng nhập kèm dữ liệu người dùng này cần phải vượt qua chuỗi xử lý tuần tự:

- Throttling: (validation) kiểm tra số lần đăng nhập và khoảng thời gian giữa hai lần đăng nhập.
- CheckUserExist: (authentication) kiểm tra cặp email - password có tồn tại trên server.
- CheckRole: (authorization) kiểm tra role của người dùng có đủ quyền truy cập. Để đơn giản, chỉ cần in lời chào người dùng với role của họ.

Cơ sở dữ liệu được đăng ký và lưu trong Server.

Áp dụng mẫu thiết kế Chain of Responsibility để thực hiện yêu cầu này.



## Command

Capture actions

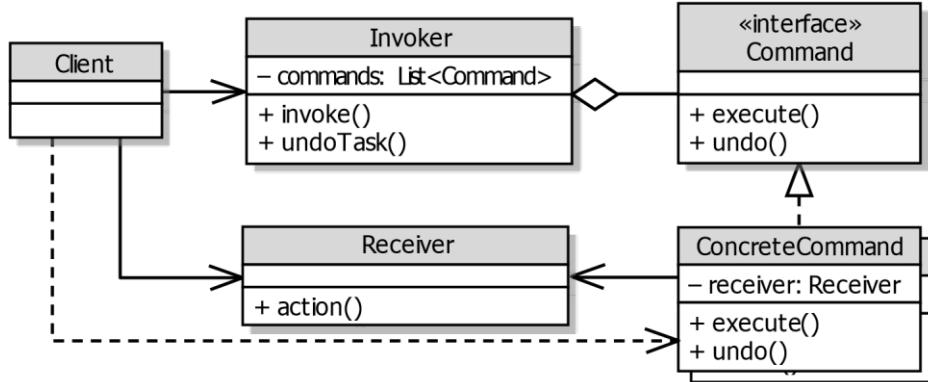
Mẫu thiết kế Command đóng gói yêu cầu (request) như một đối tượng lệnh (Command) rồi truyền đối tượng này cho đối tượng triệu gọi (Invoker) như một tham số. Tùy theo yêu cầu của Client, Invoker triệu gọi đối tượng lệnh thích hợp để thực thi.

Đối tượng lệnh ủy nhiệm việc thực hiện thực sự cho đối tượng Receiver mà đối tượng lệnh tác động lên nó.

Vì yêu cầu được đóng gói, ta có thể lưu trữ, xóa bỏ, làm lại, truy cập, thay đổi, cho áp dụng, ... một yêu cầu. Do tính chất linh hoạt đó, mẫu thiết kế Command được sử dụng để:

- Gửi yêu cầu đến các đối tượng nhận khác nhau.
- Tạo hàng đợi yêu cầu, ghi nhận (logging) yêu cầu và từ chối yêu cầu.
- Tích hợp giao tác cấp cao từ các tác vụ cơ bản.
- Tạo chuỗi thao tác ghi sẵn (macro recording).
- Cài đặt chức năng Redo và Undo nhiều cấp.

### 1. Cài đặt



- Command: khai báo giao diện cho việc thực thi một yêu cầu, ví dụ phương thức execute().
- ConcreteCommand: loại Command cụ thể, đã được "tiêm" đối tượng nhận yêu cầu (Receiver) thích hợp. Phương thức execute() của ConcreteCommand được cài đặt bằng cách ủy nhiệm cho đối tượng Receiver đã "tiêm" vào nó thực hiện.
- Invoker: đối tượng triệu gọi yêu cầu (Command), quản lý chuỗi các đối tượng Command để triệu gọi. Invoker có thể có một danh sách lưu trữ các Command, ví dụ cho tác vụ undo và redo. Invoker còn gọi là Command Manager.
- Receiver: Command ủy nhiệm thực thi thực sự cho Receiver, Receiver nhận yêu cầu từ Command, dùng các phương thức của mình (action()) để thực thi yêu cầu đó.
- Client: tạo ra các đối tượng Invoker, rồi dùng Invoker triệu gọi các đối tượng ConcreteCommand mong muốn. **Các bước thực hiện**

- Tạo interface Command, định nghĩa phương thức thực hiện lệnh (execute()) được đóng gói trong Command. Cung cấp tác vụ undo() nếu hệ thống cần đến.

- Viết các ConcreteCommand, chứa tham chiếu đến Receiver để ủy nhiệm cho Receiver thực hiện yêu cầu.

- Client sử dụng các Command, nó thiết lập Receiver và các tham số yêu cầu cho Command, rồi gọi execute() của Command.

```

import java.util.LinkedList;
import java.util.List; import
java.util.Stack;

```

```

// Invoker class
Invoker {
 Stack<Command> history = new Stack<>();
 Stack<Command> redoList = new Stack<>(); public
 void invoke(Command command) {
 command.execute();
 }
}

```

```

 if (!(command instanceof UndoCommand) && !(command instanceof RedoCommand))
 history.push(command);
 }
 public void undo() { if
 (!history.isEmpty()) {
 Command command = history.pop();
 command.undo(); redoList.push(command);
 }
 }
 public void redo() { if
 (!redoList.isEmpty()) {
 Command command = redoList.pop();
 command.execute(); history.push(command);
 }
 }
}
// Command abstract class
Command { public abstract void
execute(); public abstract void
undo();
}

// ConcreteCommand
class UndoCommand extends Command {
Invoker receiver;
 public UndoCommand(Invoker receiver) {
this.receiver = receiver;
 }
@Override public void execute() { receiver.undo(); }
@Override public void undo() { }
}
class RedoCommand extends Command {
Invoker receiver;
 public RedoCommand(Invoker receiver) {
this.receiver = receiver;
 }
@Override public void execute() { receiver.redo(); }
@Override public void undo() { }
}
class InsertCommand extends Command {
Document document;
String content;
 public InsertCommand(Document document, String content) {
this.document = document; this.content = content;
 }
@Override public void execute() { document.insert(content); }
@Override public void undo() { document.remove(); }
}
// Receiver
class Document {
List<String> list = new LinkedList<>();
public void insert(String str) {
list.add(str);
}
public void remove() {
list.remove(list.size() - 1);
}

@Override public String toString() {
return String.join("\n", list);
}
}
public class Client { public static void
main(String[] args) { System.out.println("---
Command Pattern ---");
 Document doc = new Document();
 Invoker invoker = new Invoker();
 UndoCommand undo = new UndoCommand(invoker);
 RedoCommand redo = new RedoCommand(invoker);
}

```

```

 invoker.invoke(new InsertCommand(doc, "Java Programming Language"));
 invoker.invoke(new InsertCommand(doc, "Object-Oriented Programming"));
 invoker.invoke(new InsertCommand(doc, "Design Patterns"));
 System.out.println("Insert 3 lines:\n" + doc); invoker.invoke(undo);
 invoker.invoke(undo);
 System.out.println("Undo 2 times:\n" + doc);
 invoker.invoke(redo);
 System.out.println("Redo 1 time:\n" + doc);
 }
}

```

Mỗi lệnh InsertCommand triệu gọi được lưu vào stack history, Receiver của lệnh này là Document. InsertCommand tác động vào list document của Document.

Các lệnh UndoCommand và RedoCommand không cần lưu vào stack history, Receiver của các lệnh này là Invoker vì chúng tác động vào các stack history và redoList của Invoker. redoList lưu các InsertCommand được undo để redo nếu cần.

## 2. Liên quan

- Composite: có thể được dùng để cài đặt các vĩ lệnh (macro), kết hợp phức nhiều lệnh phức tạp. Ví dụ: lệnh restart() là kết quả của việc gọi lệnh stop() rồi gọi lệnh start().
- Memento: có thể giữ trạng thái của lệnh, cần cho tác vụ khôi phục (undo).
- Prototype: Command có thể được nhân bản bằng Prototype rồi đặt vào danh sách history.

## 3. Java API

Interface java.lang.Runnable là một ví dụ của mẫu thiết kế Command.

Giao diện javax.swing.Action, thừa kế giao diện ActionListener, chính là giao diện Command trong mẫu thiết kế Command. Trong lớp dẫn xuất, cài đặt phương thức actionPerformed(ActionEvent).

```

import java.awt.event.ActionEvent; import
javax.swing.AbstractAction;

```

```

class FileAction extends AbstractAction {
 FileAction(String name) {
 super(name);
 }

 @Override public void actionPerformed(ActionEvent ae) {
 // add action logic here
 }
}

```

Thêm action vào thanh menu:

```

import java.awt.Event;
import javax.swing.*; //
...
JMenu fileMenu = new JMenu("File");
FileAction newAction = new FileAction("New"); JMenuItem
item = fileMenu.add(newAction);
item.setAccelerator(KeyStroke.getKeyStroke('N', Event.CTRL_MASK));
// To add action to a toolbar
JToolBar toolbar = new JToolBar(); toolbar.add(newAction);

```

## 4. Sử dụng

có:

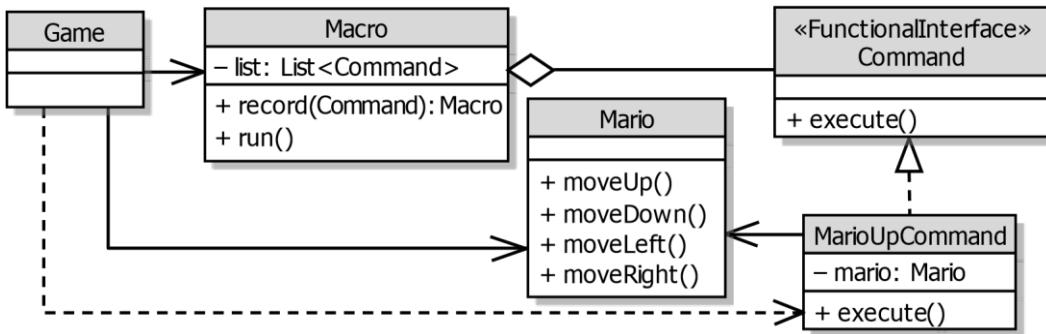
- Các lệnh với đối tượng nhận khác nhau sẽ được xử lý theo cách khác nhau.
- Một tập lệnh cấp cao được thực thi bởi các tác vụ cơ bản.

Ta muốn:

- Chỉ định, xếp thứ tự thực thi và thực thi các lệnh vào những thời điểm khác nhau. Đặc biệt với yêu cầu callback.
- Hỗ trợ chức năng Undo cho các lệnh.
- Hỗ trợ ghi nhận, kiểm soát các thay đổi do tác động của lệnh.

## 5. Bài tập

- Áp dụng mẫu thiết kế Command, đóng gói các lệnh chuyển động của nhân vật Mario. Các đối tượng lệnh này dùng lưu trong danh sách lệnh của lớp Macro để tạo các vĩ lệnh (macro).



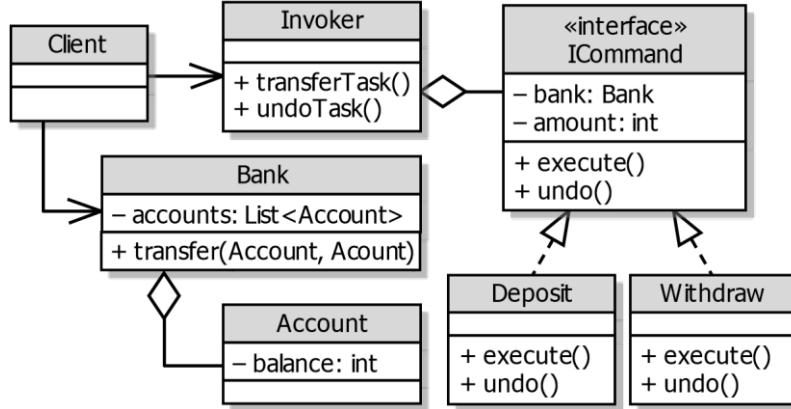
Bạn có thể dùng Command để chuyển đổi một tác vụ vào một đối tượng (Command là first-class object). Vì vậy, các đối tượng Command có thể viết gọn bằng biểu thức Lambda hoặc tham chiếu đến phương thức. Nghĩa là, thay vì bạn truyền một đối tượng Command, bạn truyền hành vi của đối tượng mà Command ủy nhiệm thực thi:

```

// cài đặt lớp MarioUpCommand trước.
Mario mario = new Mario(); // Receiver, đối tượng Command sẽ ủy nhiệm thực thi
MarioUpCommand up = new MarioUpCommand(mario); // Command
new Macro().record(up).run(); // truyền Command, nhưng thực thi là Receiver
// không cần cài đặt lớp Command cụ thể, chỉ cần truyền hành vi của đối tượng ủy nhiệm thực thi
Mario mario = new Mario(); // Receiver, đối tượng Command sẽ ủy nhiệm thực thi
new Macro().record(() -> mario.moveUp()).run(); // truyền hành vi, dùng Lambda new
Macro().record(mario::moveUp).run(); // truyền hành vi, dùng method reference

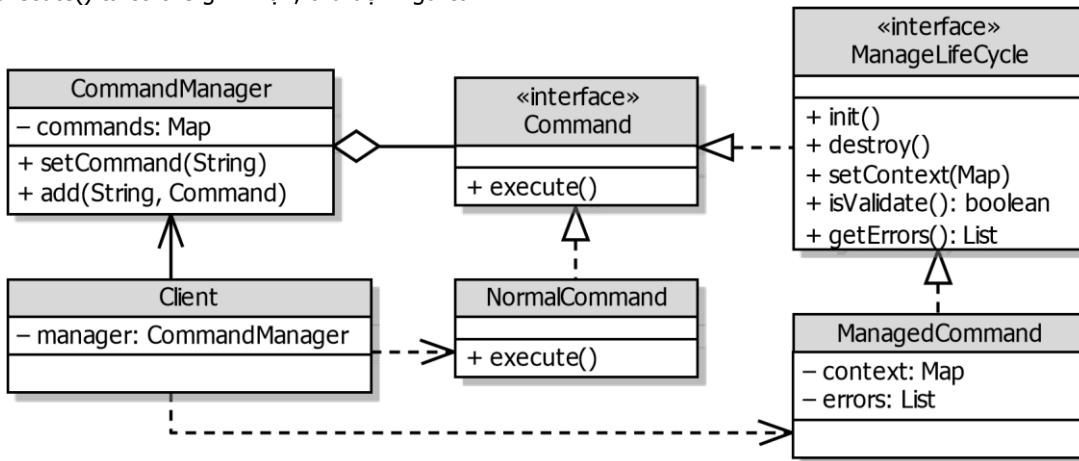
```

b) Các tác vụ nạp tiền (deposit) và rút tiền (withdraw) từ tài khoản (Account) của một ngân hàng (Bank) có thể đóng gói vào đối tượng ( ICommand) để tạo nên các tác vụ phức tạp hơn như tác vụ chuyển tiền (transfer) giữa hai tài khoản và hồi tác (undo) việc chuyển tiền. Hãy áp dụng mẫu thiết kế Command để thực hiện yêu cầu này.



c) Mẫu thiết kế Command có thể được mở rộng, hỗ trợ thêm một số dịch vụ cho Command như xác thực quyền, kiểm thử dữ liệu nhập (validation), ghi nhận thực hiện lệnh.

Điều này được thực hiện bằng cách dùng giao diện ManageLifeCycle, mở rộng giao diện Command. Giao diện ManageLifeCycle cung cấp nhiều phương thức, sẽ được gọi cho mỗi yêu cầu, thay vì chỉ gọi một phương thức execute(). Cụ thể, đối với Command mở rộng (Command được quản lý, ManagedCommand), trước khi gọi execute() ta có thể xác thực hoặc kiểm thử, sau khi gọi execute() ta có thể ghi nhận, thu dọn ngữ cảnh.



## Interpreter

Interpret a language using its grammar

Mẫu thiết kế Interpreter cài đặt bộ dịch cho một ngôn ngữ. Dịch (interpret) được hiểu như là một tác vụ trên biểu thức của ngôn ngữ đó, kết quả việc dịch có thể là:

- Định trị biểu thức, ví dụ định trị một biểu thức boolean.
- Thay đổi cấu trúc biểu thức, ví dụ chuyển biểu thức từ infix thành postfix.
- Kiểm tra cú pháp, ví dụ kiểm tra tính hợp lệ một biểu thức.
- Chuyển sang ngôn ngữ khác, ví dụ chuyển biểu thức từ XML thành JSON.

Để cài đặt cho bộ dịch, cần định nghĩa ngữ pháp (grammar) của ngôn ngữ nguồn. Ngữ pháp bao gồm một tập các ký hiệu nonterminal, một tập các ký hiệu terminal, các luật khi áp dụng sẽ thay thế về trái thành về phải.

Ví dụ, ngữ pháp của biểu thức số học infix.

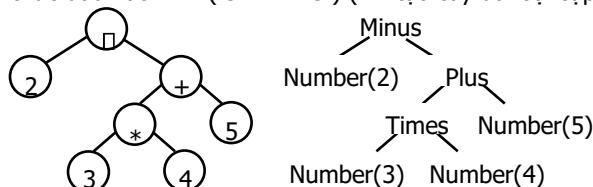
Grammar:

```
Expression ::= Plus | Minus | Times | Div | Number
Plus ::= Expression '+' Expression
Minus ::= Expression '-' Expression
Times ::= Expression '*' Expression
Div ::= Expression '/' Expression
Number ::= int
```

Mẫu thiết kế Interpreter định nghĩa ngữ pháp bằng cách cài đặt với một hệ thống phân cấp các lớp, mỗi lớp xử lý một thành phần ngữ pháp đó.

Đầu vào của bộ dịch thường là một câu (sentence), vì một biểu thức cũng là một câu nên thường nhầm lẫn hai khái niệm này. Câu được thể hiện thành cây cú pháp trừu tượng (AST – Abstract Syntax Tree) với các thành phần lấy từ ngữ pháp của ngôn ngữ đó. Mỗi câu/biểu thức đầu vào khác nhau tạo thành cây AST khác nhau.

Ví dụ, hình dưới là cây AST của biểu thức đầu vào:  $2 - (3 * 4 + 5)$  (khi tạo cây đã loại bỏ ngoặc).

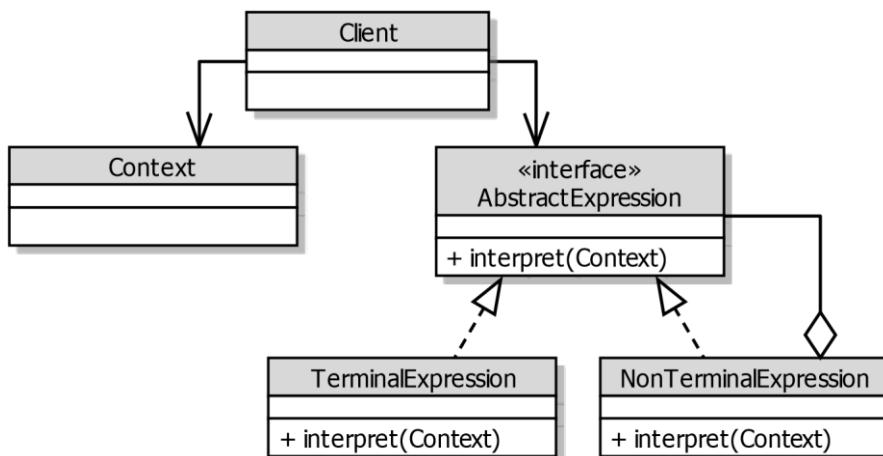


Bạn có thể dùng công cụ tại: <http://mshang.ca/syntree/>, nhập vào biểu thức sau để thấy cây AST.

[Minus [Number(2)] [Plus [Times [Number(3)] [Number(4)]] [Number(5)]]] Cây AST bao gồm:

- Thành phần non-terminal, node trong của cây, thường định nghĩa đệ quy. Ví dụ, các biểu thức Plus, Minus, Times, Div.
  - Thành phần terminal, node lá của cây. Ví dụ, biểu thức Number.
- Các thành phần này chính là các thành phần ngữ pháp được mẫu thiết kế Interpreter định nghĩa. Dịch một câu/biểu thức thực chất là duyệt cây AST tương ứng và xử lý từng thành phần ngữ pháp quyết qua thành kết quả đích. GoF không mô tả giải thuật tạo và duyệt cây AST, bạn phải:  
- Dùng giải thuật phù hợp. Ví dụ, dùng giải thuật Dijkstra tạo cây AST từ biểu thức infix.  
- Có thể sử dụng mẫu thiết kế Visitor.

### 1. Cài đặt



- AbstractExpression: khai báo tác vụ dịch interpret() trừu tượng dùng chung cho tất cả các node của AST.
- TerminalExpression: cài đặt tác vụ interpret() liên kết với các ký hiệu terminal trong ngữ pháp của ngôn ngữ. Một thể hiện của lớp này tương ứng một ký hiệu terminal.
- NonTerminalExpression: cài đặt tác vụ interpret() liên kết với các ký hiệu non-terminal trong ngữ pháp của ngôn ngữ, thường viết đệ quy. Một thể hiện của lớp này tương ứng một luật trong ngữ pháp.
- Context: lưu thông tin toàn cục cho việc dịch, thường chứa chuỗi nhập và kết quả. Lưu ý là không nhất thiết phải có lớp Context, bạn có thể truyền chuỗi nhập như tham số và lấy kết quả như trả về. **Các bước thực hiện**
- Viết ra ngữ pháp (tập luật) của ngôn ngữ sẽ áp dụng mẫu thiết kế Interpreter.

- + Định nghĩa lớp abstract/interface thể hiện Expression, trong đó định nghĩa phương thức interpret() để dịch biểu thức đó. + Mỗi luật trở thành một Expression. Expression không cần expression khác để dịch trở thành TerminalExpression.
- + Các lớp NonTerminalExpression chứa các Expression khác. Phương thức interpret() của nó có thể gọi interpret() của các Expression con để dịch.
- Xây dựng cây AST bằng cách dùng các Expression trên được thực hiện bởi Client hoặc dùng lớp Builder riêng. Client dùng cây AST để xây dựng một câu (sentence) gồm một hay nhiều Expression.
- Context thường chứa câu được truyền cho phương thức interpret để dịch.

```

import java.util.ArrayList;
import java.util.List; import
java.util.Scanner; import
java.util.Stack; import
java.util.stream.Stream;

// AbstractExpression interface
Expression {
 boolean interpret(User user);
}

// NonTerminal Expression
class AndExpression implements Expression {
 Expression left;
 Expression right;

 public AndExpression(Expression left, Expression right) {
 this.left = left; this.right = right;
 }

 @Override public boolean interpret(User user) {
 return left.interpret(user) && right.interpret(user);
 }

 @Override public String toString() {
 return left +" AND "+ right;
 }
}

class OrExpression implements Expression {
 Expression left;
 Expression right;

 public OrExpression(Expression left, Expression right) {
 this.left = left; this.right = right;
 }

 @Override public boolean interpret(User user) {
 return left.interpret(user) || right.interpret(user);
 }

 @Override public String toString() {
 return left + " OR " + right;
 }
}

class NotExpression implements Expression {
 Expression exp;

 public NotExpression(Expression exp) {
 this.exp = exp;
 }

 @Override public boolean interpret(User user) {
 return !exp.interpret(user);
 }

 @Override public String toString() {
 return " NOT " + exp;
 }
}

```

```

// TerminalExpression
class Permission implements Expression{
 String permission;

 public Permission(String permission) {
 this.permission = permission.toLowerCase();
 }

 @Override public boolean interpret(User user) {
 return user.permissions.contains(permission);
 }

 @Override public String toString() {
 return permission;
 }
}
// Context
class User {
 List<String> permissions = new ArrayList<>();
 String username;
 public User(String username, String... permissions) {
 this.username = username;
 Stream.of(permissions).forEach(e -> this.permissions.add(e.toLowerCase()));
 }
}
class Report {
 String name;
 String permission;

 // DOCUMENT_NAME PERMISSION_EXPRESSION public
 Report(String name, String permission) {
 this.name = name;
 this.permission = permission;
 }
}

class ExpressionBuilder {
 Stack<Expression> permissions = new Stack<>();
 Stack<String> operators = new Stack<>();

 public Expression build(Report report) {
 parse(report.permission); buildSentence();
 if (permissions.size() > 1 || !operators.isEmpty())
 System.out.println("ERROR!");
 return permissions.pop();
 }

 private void parse(String permission) {
 Scanner sc = new Scanner(permission.toLowerCase());
 while (sc.hasNext()) {
 String token;
 switch ((token = sc.next())) {
 case "and": operators.push("and"); break;
 case "not": operators.push("not"); break;
 case "or": operators.push("or"); break;
 default:
 permissions.push(new Permission(token));
 }
 }
 }

 private void buildSentence() {
 while (!operators.isEmpty()) {
 String operator = operators.pop();
 Expression exp1;
 Expression exp2;
 Expression exp;
 switch (operator) {
 case "not": exp1 = permissions.pop(); exp = new NotExpression(exp1); break;
 case "and": exp1 = exp; exp = new AndExpression(exp1); break;
 case "or": exp1 = exp; exp = new OrExpression(exp1); break;
 }
 permissions.push(exp);
 }
 }
}

```

```

permissions.pop(); exp2 =
permissions.pop(); exp = new
AndExpression(exp1, exp2);
break; case "or": exp1
= permissions.pop(); exp2 =
permissions.pop(); exp = new
OrExpression(exp1, exp2);
break; default: throw
new IllegalArgumentException("Unknown
operator:" + operator);
}
permissions.push(exp);
}
}
}

public class Client {
public static void main(String[] args) {
System.out.println("--- Interpreter Pattern ---");
Report report = new Report("Secret report", "FINANCE_ADMIN OR ADMIN");
Expression sentence = new ExpressionBuilder().build(report);
System.out.println(sentence);
User user1 = new User("Trump", "USER");
System.out.printf("[%s] access [%s]: %s%n", user1.username, report.name,
sentence.interpret(user1) ? "granted" : "denied");
User user2 = new User("Obama", "ADMIN");
System.out.printf("[%s] access [%s]: %s%n", user2.username, report.name,
sentence.interpret(user2) ? "granted" : "denied");
}
}
}

```

## 2. Liên quan

- Composite: cây AST được xây dựng bằng mẫu thiết kế Composite.
- Flyweight: áp dụng để tạo phần dùng chung cho các ký hiệu terminal trong cây AST.
- Iterator: dùng để duyệt cây AST.
- Visitor: dùng đóng gói hành vi xử lý từng node trên cây AST.

3. Java API `java.util.regex.Pattern` và `java.util.regex.Matcher` dùng một thiết kế Interpreter nội.

`java.text.Normalizer`.

Tất cả lớp con của `java.text.Format` (`DateFormat`, `MessageFormat`, `NumberFormat`).

## 4. Sử dụng Ta có:

- Ngữ pháp của ngôn ngữ cần diễn dịch không quá phức tạp. -  
Hiệu quả diễn dịch, như tốc độ dịch, không phải là yêu cầu chính.

## 5. Bài tập

a) Biểu thức đầu vào là một biểu thức số học infix, có ngữ pháp như sau:

Grammar:

```

Expression ::= Plus | Minus | Times | Div | Number
Plus ::= Expression '+' Expression
Minus ::= Expression '-' Expression
Times ::= Expression '*' Expression
Div ::= Expression '/' Expression
Number ::= int

```

Áp dụng mẫu thiết kế Interpreter, viết chương trình thực hiện các tác vụ "dịch" sau:

- `evaluation()`, "dịch" biểu thức infix sang trị.
- `preorder()`, "dịch" biểu thức infix sang biểu thức prefix.
- `postorder()`, "dịch" biểu thức infix sang biểu thức postfix.

Chi tiết về các thuật toán "dịch" trình bày trong tài liệu "Cấu trúc dữ liệu và thuật toán", cùng người viết.

b) Ngôn ngữ nguồn là số La mã, có ngữ pháp như sau (□ là rỗng):

Grammar:

```

Thousand ::= 'M' Thousand | □; // 1000 □
Hundred ::= 'C''D' | 'C''M' | 'D' 1e300 | 1e300; // 400 | 900 | 500 - 800 | 000 - 300
1e300 ::= □ | 'C''C''C' | 'C''C' | 'C';
Ten ::= 1e30 | 'X''L' | 'L' 1e30 | 'X''C'; 1e30 // 00 - 30 | 40 | 50 - 80 | 90

```

```

 ::= 0 | 'X''X''X' | 'X''X' | 'X';
Unit ::= le3 | 'I''V' | 'V' le3 | 'I''X' ; // 0 - 3 | 4 | 5 - 8 | 9
le3 ::= 0 | 'I''I''I' | 'I''I' | 'I';

```

Áp dụng mẫu thiết kế Interpreter, viết chương trình dịch số Lã mã sang số thập phân (Arabic) hiện dùng.

c) Cho ngữ pháp của biểu thức Boolean.

Grammar:

```

e ::= e '&' e
| e '|'| e
| '!' e
| '(' e ')'
| var '=' e
| var

```

Áp dụng mẫu thiết kế Interpreter, cài đặt tác vụ "dịch" là định trị một biểu thức Boolean.

Biểu thức Boolean, ví dụ: X = (A & B) | !C được tạo như sau:

```

Logic term = new ANDLogic(new Variable("A"), new Variable("B"));
term = new ORLogic(term, new NOTLogic(new Variable("C"))); term
= new AssignmentLogic(new Variable("X"), term);

```

## Iterator

Access aggregated objects

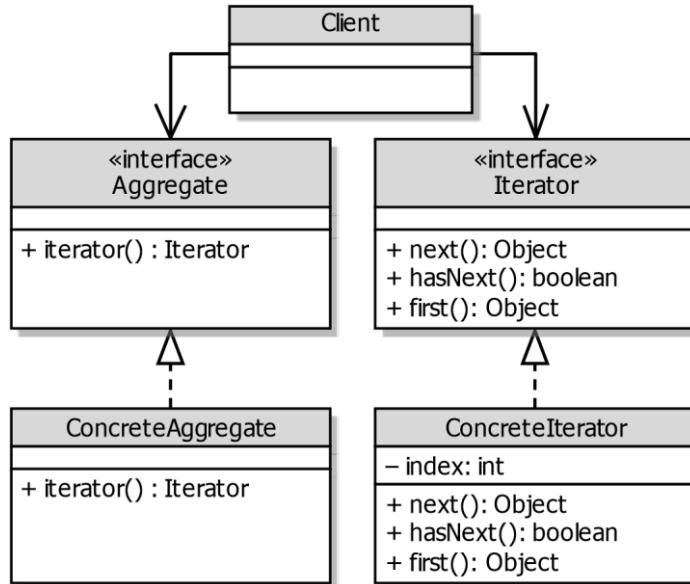
Mẫu thiết kế Iterator cung cấp một cách truy cập thống nhất đến các đối tượng thành phần, nằm bên trong một đối tượng chứa (container hoặc collection), mà không cần phải hiểu rõ đến cấu trúc nội tại của đối tượng chứa đó.

Mặc dù một đối tượng chứa thường có đủ phương thức để truy cập các đối tượng thành phần của nó, ví dụ:

```
ArrayList<Book> books = Bookstore.getBooks(); for (int i = 0; i < books.size(); ++i)
 System.out.println(i + ": " + books.get(i));
```

nếu để bảo đảm nguyên tắc SRP, tác vụ duyệt các đối tượng thành phần của đối tượng chứa được tách ra và đóng gói vào một đối tượng chuyên dụng gọi là Iterator. Iterator được hình dung như một "con trỏ" dịch chuyển trong đối tượng chứa, dùng để truy cập các đối tượng thành phần của đối tượng chứa. Do đặc điểm này, Iterator còn gọi là Cursor.

### 1. Cài đặt



- Iterator: định nghĩa một giao diện chuẩn để truy cập và duyệt các đối tượng thành phần của đối tượng chứa. Các tác vụ điển hình bao gồm: trả đến đối tượng thành phần kế tiếp (next()), kiểm tra xem có đối tượng thành phần kế tiếp không (hasNext()).

- ConcreteIterator: cài đặt cho giao diện Iterator, giữ tham chiếu chỉ đến vị trí hiện tại khi duyệt đối tượng chứa, tức vị trí của đối tượng thành phần mà Iterator hiện đang trả đến.

- Aggregate: giao diện của đối tượng chứa các đối tượng thành phần khác. Khai báo phương thức iterator(), trả về đối tượng Iterator dùng để duyệt các đối tượng thành phần của nó. Còn gọi là IterableCollection.

- ConcreteAggregate: cài đặt giao diện Aggregate để tạo đối tượng Iterator, trả về một đối tượng ConcreteIterator cụ thể. Khi trả về Iterator, Aggregate trao cho Iterator này tham chiếu chỉ đến chính nó để Iterator có thể duyệt các đối tượng thành phần của Aggregate đó. Còn gọi là ConcreteCollection.

Khi sử dụng, Client gọi phương thức iterator() của đối tượng chứa Aggregate để nhận được đối tượng Iterator dùng duyệt nó. Iterator này được gọi là external iterator (iterator chủ động), được điều khiển bởi Client. Ngoài ra, còn các internal iterator (iterator thụ động) được điều khiển bởi Aggregate.

- Client làm việc với Iterator và Aggregate thông qua interface, bằng cách này Client không kết nối chặt với các lớp cụ thể, cho phép dùng nhiều loại collection và iterator trong code của Client. **Các bước thực hiện**

- Định nghĩa interface Iterator: có phương thức kiểm tra một phần tử có tồn tại trong Aggregate (hasNext()), phương thức lấy phần tử (next()), phương thức thiết lập lại con trỏ (first()).

- Cài đặt lớp ConcreteIterator, thường là lớp nội của lớp ConcreteAggregate. Lớp cần có trạng thái nội lưu trữ vị trí của phần tử trong tập hợp. Các phương thức có thể ném ra exception để báo lỗi truy cập nếu sự thay đổi của cấu trúc dữ liệu bên dưới làm iterator hoạt động sai.

- Trong ConcreteAggregate, thêm phương thức trả về ConcreteIterator tương ứng.

```
import java.util.Arrays; import
java.util.Collection; import
java.util.Collections; import
java.util.List;

// Aggregate interface
CollectionIF {
 IteratorIF iterator();
 Collection elements();
}

// Iterator interface
IteratorIF {
 boolean
```

```

hasNext(); Object
next();
}

// ConcreteAggregate
class ConcreteCollection implements CollectionIF {
List list;
 public ConcreteCollection(Object[] objectList) {
list = Arrays.asList(objectList);
 }

@Override public IteratorIF iterator() {
return new ConcreteIterator(this);
}

@Override public Collection elements() {
return Collections.unmodifiableList(list);
}
}

// ConcreteIterator
class ConcreteIterator implements IteratorIF {
List list; int index;
 public ConcreteIterator(CollectionIF collection) {
list = (List)collection.elements(); index = 0;
 }

@Override public boolean hasNext() {
return (index < list.size());
}

@Override public Object next() {
try {
 return list.get(index++);
} catch (IndexOutOfBoundsException e) {
 throw new RuntimeException("No Such Element");
}
}
}

public class Client {
public static void main(String[] args) {
System.out.println("--- Iterator Pattern ---");
String[] books = { "Sequential", "Procedural", "OOP", "Design Patterns" };
CollectionIF collection = new ConcreteCollection(books);
System.out.println("Getting an iterator for the collection...");
IteratorIF iterator = collection.iterator();
System.out.println("Iterate through the list.");
for (int i = 0; iterator.hasNext(); ++i)
 System.out.println(i + ": " + iterator.next());
}
}
}

```

## 2. Liên quan

- Composite: Iterator thường dùng để duyệt một cấu trúc đệ quy như các Composite.
- Factory Method: phương thức iterator() là một Factory Method của Aggregate, lớp dẫn xuất của nó mới quyết định việc tạo Iterator thích hợp cho loại ConcreteAggregate tương ứng.
- Memento: thường dùng cùng với Iterator. Iterator có thể sử dụng Memento để lưu trạng thái duyệt.

## 3. Java API

Mẫu thiết kế Iterator đã được tích hợp vào Java API, các lớp Collection đều có phương thức iterator() trả về thực thể cài đặt giao diện java.util.Iterator dùng truy cập các đối tượng thành phần của Collection đó.

Hơn thế nữa, nếu người dùng tạo các collection tùy biến riêng, Java API hỗ trợ người dùng áp dụng mẫu thiết kế Iterator, dễ dàng tạo iterator cho collection, cho phép dùng cả vòng lặp for tăng cường (foreach) tiện dụng, cả phương thức default forEach() cho collection.

Để áp dụng mẫu thiết kế Iterator:

- Collection của bạn phải cài đặt giao diện java.lang.Iterable, bạn cần hiện thực phương thức iterator() trả về một java.util.Iterator. Giao diện Iterable đóng vai trò giao diện Aggregate trong mẫu thiết kế Iterator.

- ConcreteIterator phải cài đặt giao diện java.util.Iterator, cần hiện thực các phương thức hasNext(), next() và remove(). Giao diện Iterator đóng vai trò giao diện Iterator trong mẫu thiết kế Iterator.

Ví dụ sau đây, viết lại ví dụ trên nhưng áp dụng mẫu thiết kế Iterator được tích hợp trong Java API.

```
import java.util.Arrays; import
java.util.Collection; import
java.util.Collections; import
java.util.Iterator; import
java.util.List;

interface CollectionIF<E> extends Iterable<E> {
 Collection<E> elements();
}

class ConcreteCollection<E> implements CollectionIF<E> {
 List<E> list;
 public ConcreteCollection(E[] objectList) {
 list = Arrays.asList(objectList);
 }

 @Override public Iterator<E> iterator() {
 return new ConcreteIterator(this);
 }

 @Override public Collection<E> elements() {
 return Collections.unmodifiableList(list);
 }
}

class ConcreteIterator<E> implements Iterator<E> {
 List<E> list;
 int index;
 public ConcreteIterator(CollectionIF<E> collection) {
 list = (List<E>)collection.elements(); index = 0;
 }

 @Override public boolean hasNext() {
 return (index < list.size());
 }

 @Override public E next() {
 try {
 return list.get(index++);
 } catch (IndexOutOfBoundsException e) {
 throw new RuntimeException("No such element");
 }
 }

 @Override public void remove() { }
}

public class Client {
 public static void main(String[] args) {
 System.out.println("--- Iterator Pattern in Java API ---");
 String[] books = {"Sequential", "Procedural", "OOP", "Design Patterns"};
 CollectionIF<String> collection = new ConcreteCollection(books);
 System.out.println("Foreach through the list.");
 int i = 0;
 for (String s : collection) System.out.println((i++) + ": " + s);
 }
}
```

Đôi khi bạn cần chạy vòng lặp duyệt qua các thuộc tính của một đối tượng Java. Ví dụ với lớp Mark bên dưới, chứa các điểm Math, Physis, Chemistry. Bạn cần xác định điểm cao nhất hoặc tính điểm trung bình. Ngoài ra, phải có khả năng thêm thuộc tính mới như thêm điểm Biology chẳng hạn. Có thể dùng Java Reflection lấy và duyệt tập thuộc tính của đối tượng. Tuy nhiên, bạn chỉ đơn giản sử dụng một cài đặt đặc biệt cho lớp, gọi là array-backed properties (các thuộc tính hậu thuẫn bởi mảng), nghĩa là đặt các thuộc tính vào một mảng.

Nếu muốn thêm một thuộc tính mới:

- Thừa kế lớp hiện hành.
- Mở rộng mảng thuộc tính bằng cách thêm một phần tử.
- Viết cặp getter/setter cho thuộc tính mới.

```
import java.util.Arrays; import java.util.Iterator; import
java.util.Spliterator; import java.util.function.Consumer; import java.util.stream.IntStream;
class Mark implements Iterable<Integer>
{ private int[] stats = new int[3];
```

```

void setMath(int math) { stats[0] = math; } void
setPhysics(int physics) { stats[1] = physics; } void
setChemistry(int chemistry) { stats[2] = chemistry; }
int getMath() { return stats[0]; }
int getPhysics() { return stats[1]; }
int getChemistry() { return stats[2]; }

@Override public void forEach(Consumer<? super Integer> action) {
for (int x : stats) action.accept(x);
}

@Override public Spliterator<Integer> spliterator() {
return Arrays.spliterator(stats);
}

@Override public Iterator<Integer> iterator() {
return IntStream.of(stats).iterator();
}
public int max() {
 return IntStream.of(stats).max().getAsInt();
}
public double average() {
 return IntStream.of(stats).average().getAsDouble();
}
}

public class Client { public static void
main(String[] args) { Mark m = new
Mark(); m.setMath(78);
 m.setPhysics(84);
 m.setChemistry(62);
 System.out.printf("Max: %d - Average: %.1f%n",
m.max(), m.average());
}
}

```

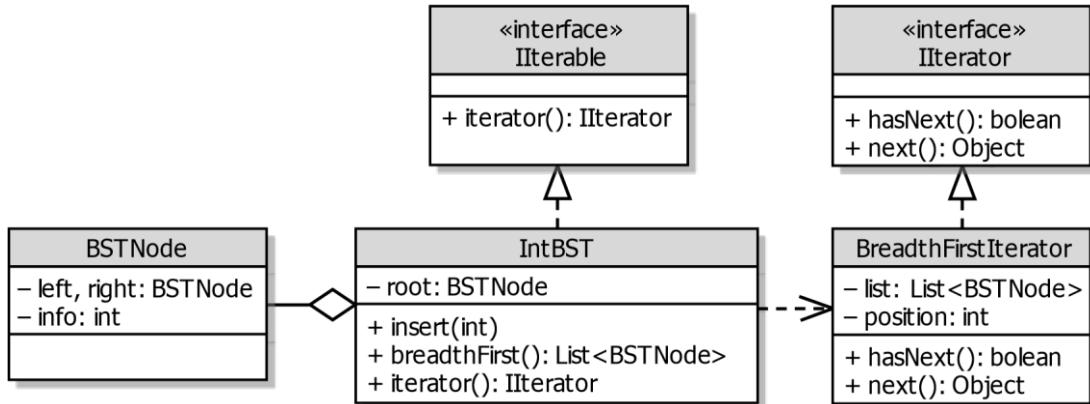
#### 4. Sử dụng Ta

muốn:

- Truy cập các đối tượng thành phần của đối tượng chứa mà không bộc lộ cấu trúc bên trong đối tượng chứa.
- Hỗ trợ nhiều phương án duyệt của các đối tượng thành phần của một đối tượng chứa.
- Cung cấp một giao diện đơn giản, tổng quát cho nhiều kiểu đối tượng chứa có cấu trúc khác nhau.

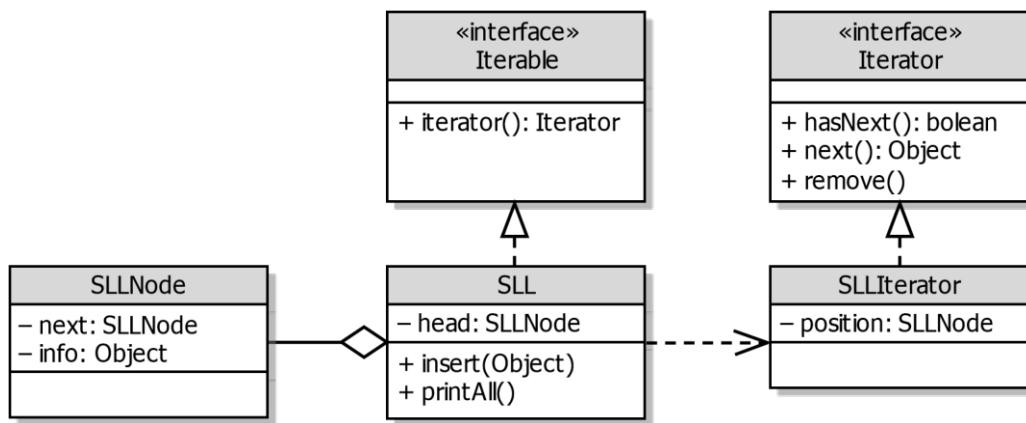
#### 5. Bài tập

- a) Tạo cây nhị phân IntBST với các phần tử là các node thuộc lớp BSTNode, chứa trị nguyên. Áp dụng mẫu thiết kế Iterator để có thể lấy được đối tượng BreadthFirstIterator từ cây IntBST. Đối tượng này cho phép duyệt các node của cây InBST theo mức.



- b) Tạo một danh sách liên kết đơn SLL (Singly Linked List) với các phần tử là các node thuộc lớp SLLNode (xem tài liệu "Cấu trúc dữ liệu và thuật toán", cùng người viết).

Áp dụng framework Iterator của Java API để tạo một Iterator cho SLL, cho phép duyệt SLL bằng vòng lặp for tăng cường.



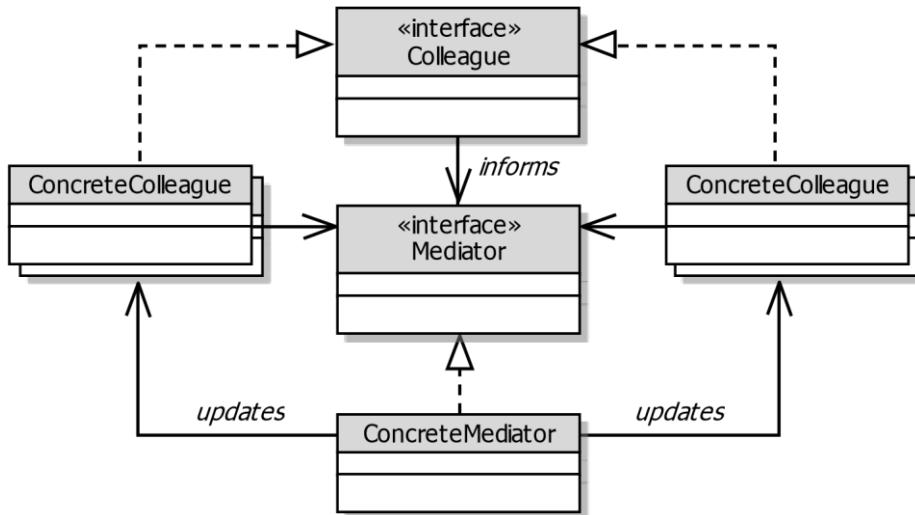
## Mediator

Define object interaction

Mẫu thiết kế Mediator dùng giải quyết độ phức tạp trong tương tác và liên lạc trực tiếp giữa các đối tượng/lớp. Mẫu thiết kế này cung cấp một lớp giữ vai trò trung gian (Mediator) giữa các đối tượng/lớp, đóng gói cách mà một tập đối tượng tương tác với nhau, bao gồm thông tin trao đổi, hành vi liên lạc giữa các đối tượng/lớp đó.

Mẫu thiết kế Mediator giúp đơn giản hóa giao thức liên lạc giữa các đối tượng/lớp, điều khiển tập trung tương tác giữa chúng, loại bỏ các thao tác liên lạc đặc thù tạo nên kết nối chặt giữa các đối tượng/lớp. Mẫu thiết kế Mediator còn gọi là Intermediary, Controller.

### 1. Cài đặt



- Mediator: khai báo giao diện cho việc liên lạc, tương tác giữa các đối tượng Colleague.
- ConcreteMediator: cài đặt các hành vi liên lạc, tương tác giữa các Colleague và tham chiếu đến các Colleague có liên quan.
- Colleague: mỗi Colleague biết đối tượng Mediator của nó và nó giao tiếp với đối tượng Mediator này khi muốn liên lạc với đối tượng Colleague khác.
- ConcreteColleague: cài đặt chức năng xử lý thông báo nhận từ Mediator.

### Các bước thực hiện

- Định nghĩa Mediator
  - + Các phương thức của Mediator sẽ được các Colleague gọi.
  - + Mediator cần biết tất cả Colleague tham gia "liên lạc" mà nó là trung gian. Vì vậy, hoặc các đối tượng Colleague phải đăng ký với Mediator.
  - + Mediator cũng có phương thức báo cho các Colleague còn lại biết có thay đổi trạng thái mà một Colleague báo đến nó.
- Định nghĩa các Colleague
  - + Colleague đăng ký với Mediator trong constructor để tham gia "liên lạc".
  - + Colleague tham chiếu đến Mediator để thông báo hoặc thay đổi các Colleague tham gia "liên lạc".

```
import java.util.ArrayList;
import java.util.Date; import
java.util.List;

// Mediator interface
Mediator<T> { void
addUser(User<T> user);
 void sendMessage(User<T> user, T message);
}

// ConcreteMediator class ChatMediator<T>
implements Mediator<T> { List<User<T>>
userList = new ArrayList<>(); @Override
public void addUser(User<T> user) {
userList.add(user);
}

@Override public void sendMessage(User<T> user, T message) {
userList.stream()
.filter(u -> (u != user))
.forEachOrdered(u -> u.receive(message));
}
```

```

// Colleague abstract class
User<T> { protected String
name; public User(String
name) { this.name =
name;
 }
 public abstract void send(Mediator<T> mediator, T message);
public abstract void receive(T message);
}

// ConcreteColleague
class ChatUser<T> extends User<T> {
public ChatUser(String name) {
super(name);
}

@Override public void send(Mediator<T> mediator, T message) {
mediator.sendMessage(this, message);
}

@Override public void receive(T message) {
System.out.println(this.name + " received: " + message);
}
}
class Message {
String message;
public Message(String message) {
this.message = new Date() + ", " + message;
}
@Override public String toString() { return message; }
}
public class Client {
public static void main(String[] args) {
System.out.println("--- Mediator Pattern ---");
ChatUser<Message> obama = new ChatUser<>("Barack Obama");
ChatUser<Message> un = new ChatUser<>("Kim Jong Un");
ChatUser<Message> putin = new ChatUser<>("Vladimir Putin");
ChatMediator<Message> msn = new ChatMediator<>();
msn.addUser(obama); msn.addUser(putin);
msn.addUser(un);
ChatMediator<Message> yahoo = new ChatMediator<>();
yahoo.addUser(putin); yahoo.addUser(un);
un.send(msn, new Message("[Kim Jong Un]: Ultimate Letter"));
un.send(yahoo, new Message("[Kim Jong Un]: Secret Letter"));
}
}

```

## 2. Liên quan

- Facade: mẫu thiết kế Facade trừu tượng hóa một hệ thống con để cung cấp một giao diện dễ dùng hơn, đây là giao thức một hướng (Client ↔ Facade ↔ Subsystem). Trong lúc Mediator là trung gian giao tiếp giữa các Colleague, đây là giao thức đa hướng.

- Observer: các Colleague có thể liên lạc với nhau bằng cách dùng mẫu thiết kế Observer.

3. Java API java.util.Timer, các phương thức scheduleXxx().  
java.util.concurrent.Executor, phương thức execute().  
java.util.concurrent.ScheduledExecutorService, phương thức schedule().  
java.lang.reflect.Method, phương thức invoke(). javax.swing.ButtonGroup là một Mediator.

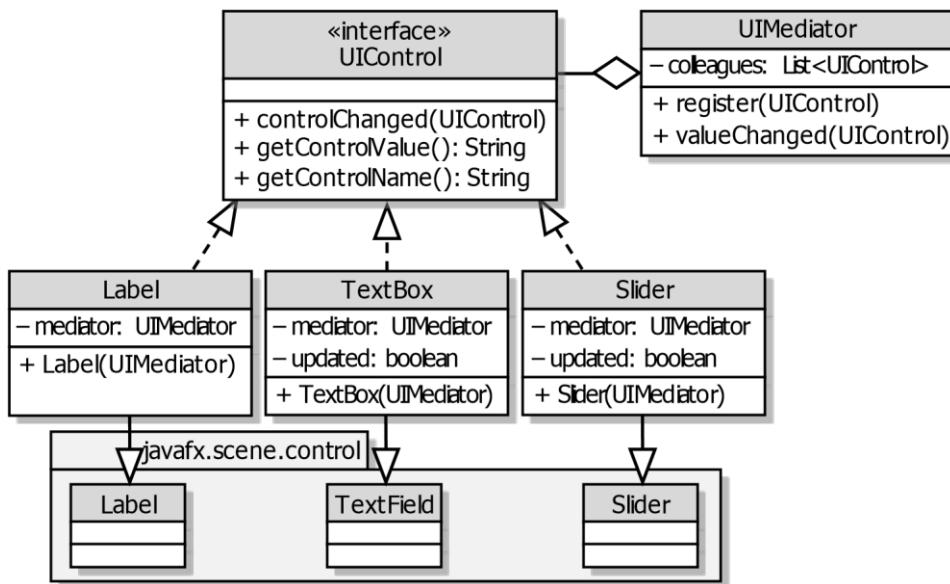
Xem xét java.util.Map. Cách điển hình để cài đặt Map là lưu trữ từng cặp key-value dưới dạng node (Map.Entry). Các node sau đó được chèn vào bảng băm (HashMap) hoặc cây tìm kiếm (TreeMap). Client sử dụng Map thường không tương tác trực tiếp với Map.Entry mà dùng các phương thức get() và put() của Map. Như vậy, cài đặt của Map (HashMap, TreeMap) giữ vai trò Mediator giữa Client và Map.Entry. Vai trò trung gian này có thể làm cho code kém hiệu quả, thay vào đó nên truy cập Map.Entry trực tiếp.

## 4. Sử dụng Ta có:

- Một tập các đối tượng liên lạc với nhau theo những cách có cấu trúc tốt nhưng lại phức tạp.
- Cần phải tùy biến hành vi liên lạc của nhóm đối tượng mà không phải dồn xuất chúng.
- Một hệ thống hoạt động dựa trên thông điệp.

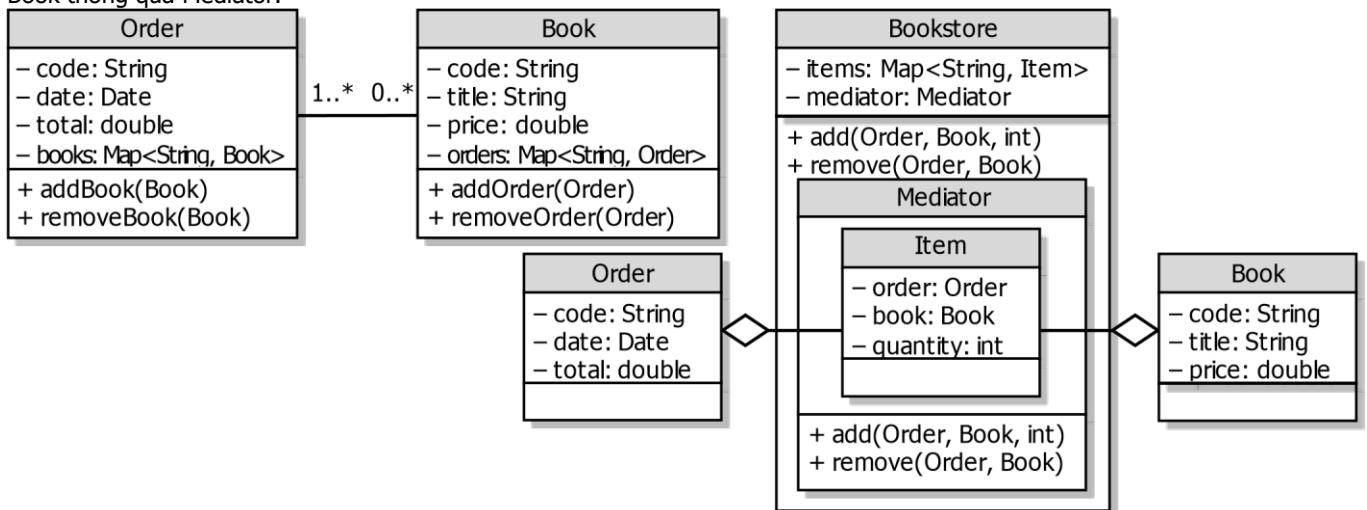
## 5. Bài tập

- a) Các colleague UIControl (Label, TextBox, Slider) có "liên lạc" với nhau. Nội dung văn bản trong TextBox thay đổi sẽ làm thay đổi con trỏ trên Slider và văn bản trên Label. Con trỏ của Slider thay đổi, vị trí của nó sẽ hiển thị trên TextBox và Label. Áp dụng mẫu thiết kế Mediator, giúp chúng "liên lạc" được với nhau.

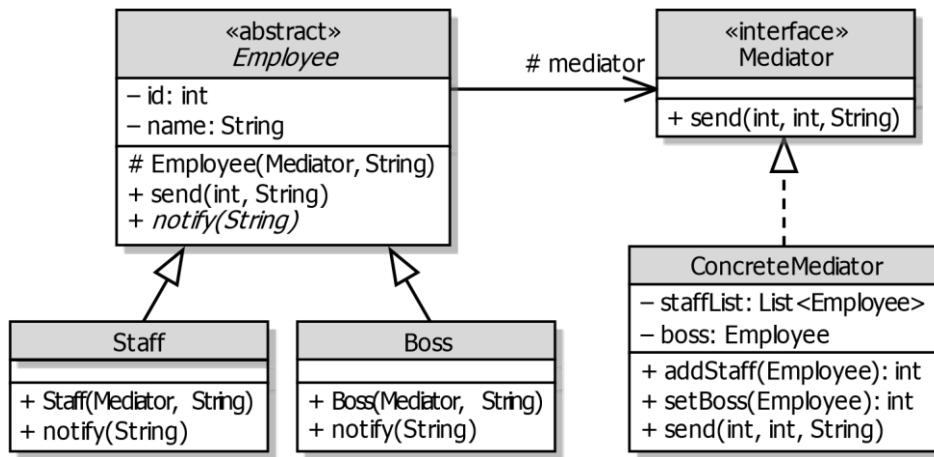


- b) Sơ đồ bên trái thể hiện quan hệ nhiều-nhiều giữa hai lớp Order và Book, làm cả hai phải giữ một Map lưu trữ thông tin của nhau. Tương tác trực tiếp giữa chúng trở nên phức tạp.

Sơ đồ bên phải áp dụng mẫu thiết kế Mediator để giảm sự phức tạp này. Lớp Bookstore làm việc với quan hệ giữa Order và Book thông qua Mediator.



- c) Các nhân viên (Employee), bao gồm quản lý (Boss) và các nhân viên khác trong công ty (Staff). Mỗi nhân viên định danh bằng id, là thứ tự khi đưa vào staffList. Các nhân viên trao đổi với nhau thông qua chat server. Thông điệp được gửi từ id nguồn (from) đến id đích (to). Boss có quyền thấy các thông điệp gửi/nhận, mặc dù Boss không quan tâm đến nội dung thông điệp. Boss cũng muốn gửi thông điệp quảng bá đến tất cả các nhân viên (trừ Boss). Dùng mẫu thiết kế Mediator để thực hiện yêu cầu này.

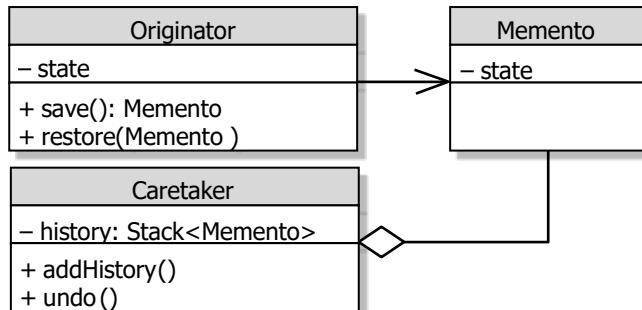


## Memento

Externalize object internal state

Mẫu thiết kế Memento được dùng khi muốn khôi phục lại trạng thái lần trước của một đối tượng (rollback). Mẫu thiết kế Memento không vi phạm tính đóng gói (encapsulation) mà vẫn có thể lấy và lưu trữ trạng thái nội của một đối tượng, vì vậy nó cung cấp khả năng khôi phục lại trạng thái đó (undo, rollback) nếu cần. Mẫu thiết kế Memento còn gọi là Token hat Snapshot.

### 1. Cài đặt



- Memento: đối tượng Memento chứa các trạng thái cần lưu trữ của một đối tượng (Originator), mỗi Memento thể hiện các trạng thái của đối tượng tại một thời điểm, gọi là "bản chụp" (snapshot) của đối tượng. Lý tưởng là đảm bảo chỉ Originator mới có quyền truy cập Memento.

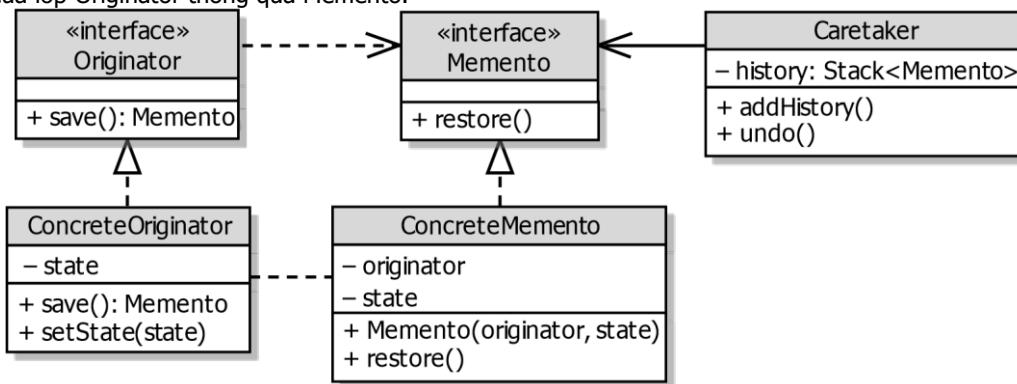
- Originator: chính là đối tượng mà ta quan tâm lưu trữ trạng thái. Thường có hai phương thức quan trọng, giúp nó lưu trữ trạng thái mà không vi phạm tính đóng gói.

+ Tạo đối tượng Memento và gán trạng thái cần lưu trữ vào đối tượng Memento đó. Nói cách khác, Originator sinh ra các "bản chụp" Memento chứa trạng thái nội của nó để lưu trữ.

+ Khôi phục trạng thái từ đối tượng Memento được lưu trữ. Nói cách khác, ta khôi phục trạng thái nội trước đây từ thông tin lấy từ "bản chụp" Memento.

- Caretaker: giữ một ArrayList hoặc Stack lưu giữ các "bản chụp" Memento. Dùng nó để lưu trữ và tìm lại các Memento đã lưu. Caretaker không quan tâm đến trạng thái được Memento lưu trữ, mà chỉ quan tâm đến các Memento được nó lưu trữ. Caretaker yêu cầu một hành động lưu trữ, yêu cầu một "bản chụp" của Originator tại một thời điểm.

Một số ngôn ngữ không hỗ trợ lớp lồng (ví dụ PHP), một cách cài đặt khác vẫn bảo đảm các lớp khác không truy cập vào được trạng thái nội của lớp Originator thông qua Memento.



### Các bước thực hiện

- Xác định trạng thái nội của Originator sẽ được lưu trữ trong Memento, lưu ý kích thước lưu trữ không quá lớn.
- Đóng gói trạng thái lưu trữ vào Memento, cài đặt sao cho chỉ có Originator truy cập và thay đổi được Memento, thường như một lớp nội của Originator.
- Originator cung cấp phương thức save() để lấy "bản chụp" hiện tại, trả về một thực thể của Memento.

- Phương thức restore(Memento) của Originator lấy đối tượng Memento như một tham số dùng nó để tự phục hồi đối tượng Originator với trạng thái lưu trữ đó. Khi phục hồi trạng thái phải quan tâm đến các đối tượng có liên quan.
- Memento được lưu trữ nội trong Originator nhưng điều này làm Originator trở nên phức tạp. Caretaker từ bên ngoài quản lý các Memento làm cài đặt thêm linh hoạt.

```

// Memento class
Memento { private
String name; private
double cost;
 public Memento(Product product) {
this.name = product.getName(); this.cost
= product.getCost();
 }
 public String getName() { return name; }
 public void setName(String name) { this.name = name; }
 public double getCost() { return cost; }
 public void setCost(double cost) { this.cost = cost; } }

// Originator class
Product { private
String name; private
double cost;
 public Product(String name, double cost) {
this.name = name; this.cost = cost;
 }
 public String getName() { return name; }
 public void setName(String name) { this.name = name; }
 public double getCost() { return cost; }
 public void setCost(double cost) { this.cost = cost; }

@Override
public String toString() {
 return String.format("%s [%.2f]", name, cost);
}
public Memento save() {
return new Memento(this);
}
public void restore(Memento memento) {
this.setName(memento.getName());
this.setCost(memento.getCost());
}
}

// Caretaker class
Caretaker {
 java.util.Stack<Memento> history = new java.util.Stack<>();
public void addHistory(Memento memento) {
history.push(memento);
}
public Memento get() {
 return history.isEmpty() ? null : saveList.pop();
}
}
public class Client { public static void
main(String[] args) { Caretaker
caretaker = new Caretaker();
 Product product = new Product("Book", 50.00);
 System.out.println(product);
 System.out.println("Change the object: ");
caretaker.addHistory(product.save());
product.setName("Meat");
 caretaker.addHistory(product.save());
 product.setCost(60.00);
System.out.println(product);
 System.out.println("Restore state via the memento...");
product.restore(caretaker.get());
product.restore(caretaker.get());
 System.out.println(product);
}
}

```

## 2. Liên quan

- Command: dùng Memento để lưu trạng thái các tác vụ có thể khôi phục lại (undo).
  - Iterator: Memento có thể được dùng cho thao tác lặp.

### 3. Java API

Lớp `java.util.Date` (long 值 Date), các lớp cài đặt giao diện `java.io.Serializable` (cho phép phục hồi trạng thái đối tượng). Hỗ trợ hoàn tác (undo) đã được cung cấp bởi `javax.swing.text.JTextComponent` và các lớp con như `JTextField`, `JTextArea`. `javax.swing.undo.UndoManager` hành động như `Caretaker`, cài đặt của interface `javax.swing.undo.UndoableEdit` làm việc như các `Memento`. `javax.swing.text.Document` cài đặt như model cho các đơn thể văn bản trong Swing giữ vai trò `Originator`.

#### 4. Sử dụng

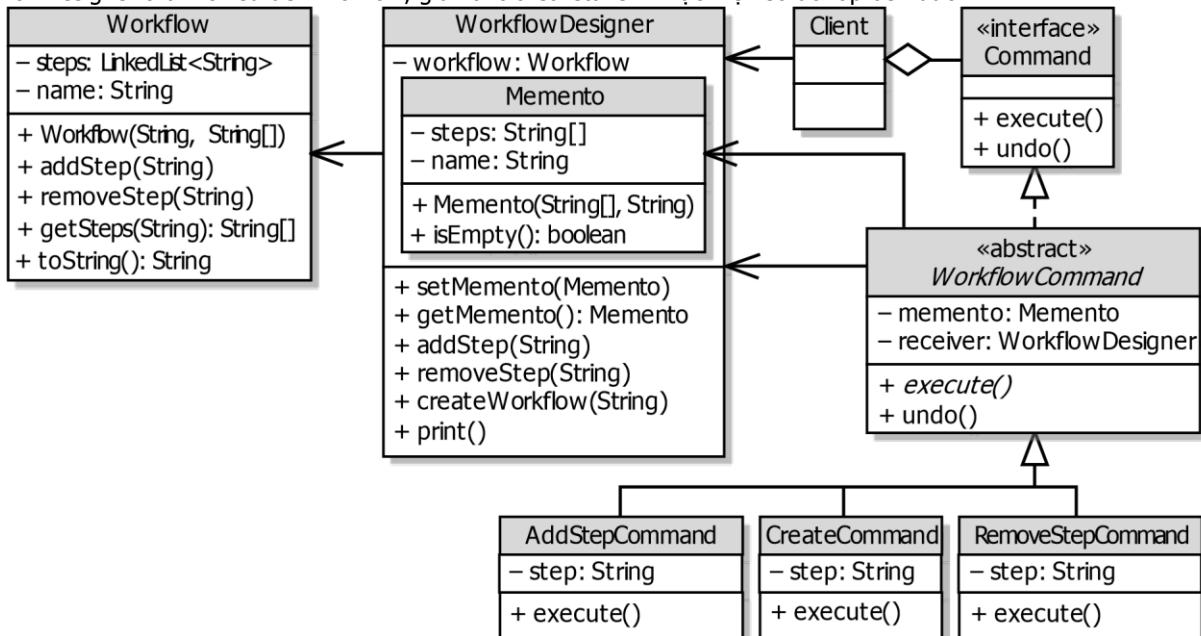
Ta muốn:

- Lưu trữ bản sao trạng thái của một đối tượng để có thể khôi phục lại sau này (chức năng Undo). - Thay đổi, khôi phục trạng thái của một đối tượng mà không can thiệp trực tiếp đến trạng thái đó.

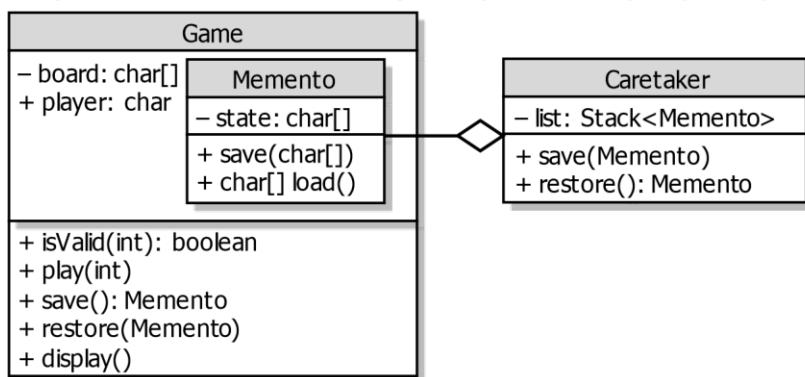
## 5. Bài tập

a) Trong ứng dụng thiết kế đồ họa, Client thực hiện (execute()) chuỗi các lệnh (Command) và đưa các lệnh được thực hiện vào stack. Lệnh tại đỉnh stack có thể được đẩy ra và thực hiện hoàn tác (undo()). Để hỗ trợ hoàn tác, người ta áp dụng mẫu thiết kế Memento. Trong đó:

- WorkflowDesigner giữ vai trò Originator, WorkflowCommand giữ tham chiếu đến WorkflowDesigner nên WorflowDesigner lưu được trạng thái vào Memento khi một lệnh được thực hiện.
  - Memento là lớp nội của WorkflowDesigner, chứa trạng thái nội cần lưu trữ, đơn giản là tên của bước lệnh thực hiện. Lưu ý, WorkflowCommand giữ tham chiếu đến Memento, nên một lệnh thực hiện
  - WorkflowDesigner tham chiếu đến Worflow, giữ vai trò Caretaker. Thực hiện sơ đồ lớp bên dưới.



b) Trò chơi TicTacToe áp dụng mẫu thiết kế Memento để lưu giữ trạng thái bàn cờ (mảng board) và người chơi (board[0]).



Đến lượt chơi của mình, người chơi ('X' hoặc 'O', 'X' đi trước) có các lựa chọn sau:

- Chọn nước đi, nước đi hợp lệ là từ 1 đến 9 và ô chọn còn trống.
  - Chọn U để hoàn tác (undo), trạng thái bàn cờ sẽ lui lại 2 bước. Nếu 'O' mới đi một nước, bàn cờ sẽ trở lại trạng thái đầu.
  - Chọn Q để thoát trò chơi. Các tùy chọn không hợp lệ sẽ yêu cầu nhập lại

--- TicTacToe ---

Player 'X' move? 5

Player '0' move? 3

Player 'X' move? U

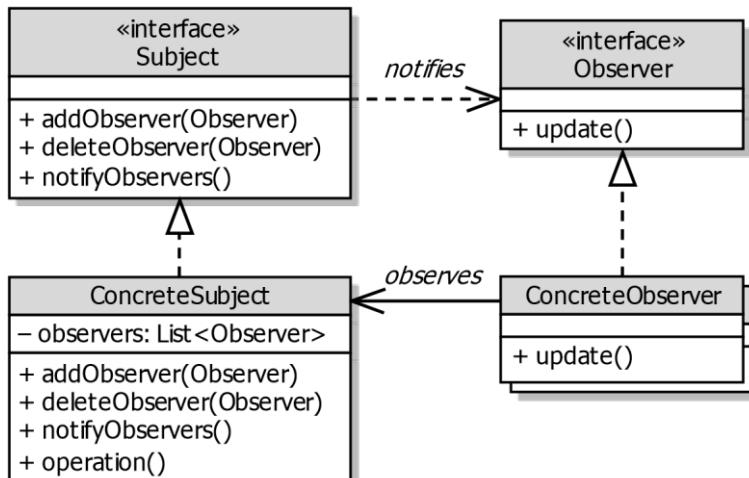
|           |           |           |           |       |       |
|-----------|-----------|-----------|-----------|-------|-------|
| 1   2   3 | 1   2   3 | 1   2   0 | 1   2   3 | ----- | ----- |
| 4   5   6 | 4   X   6 | 4   X   6 | 4   5   6 |       |       |
| 7   8   9 | 7   8   9 | 7   8   9 | 7   8   9 |       |       |

## Observer

Subscribe to object changes

Mẫu thiết kế Observer định nghĩa một phụ thuộc một-nhiều, trong đó nếu một đối tượng (Subject, còn gọi là Observable) thay đổi trạng thái, tất cả (nhiều) các đối tượng (Observer) phụ thuộc đối tượng đó sẽ được thông báo và tự động cập nhật. Phía "một" thường là dữ liệu, phía "nhiều" thường là giao diện người dùng (bảng biểu, đồ thị, báo cáo). Mẫu thiết kế này còn gọi là Dependents, Publisher/Subscriber hoặc Source/Listener.

### 1. Cài đặt



- Subject: giao diện cho đối tượng dữ liệu, khai báo các phương thức chính:

- + addObserver(): dùng thêm các Observer vào danh sách đăng ký các đối tượng cần phải thông báo về những thay đổi.
- + deleteObserver(): loại Observer chỉ định ra khỏi danh sách đăng ký các đối tượng cần thông báo về những thay đổi. Do Observer chưa một tham chiếu đến Subject mà nó đăng ký, nên khi nó không còn quan tâm đến sự thay đổi của Subject nữa, Observer sẽ thông qua tham chiếu đó, tự loại nó ra khỏi danh sách đăng ký với Subject.
- + notifyObservers(): thông báo cho các Observer trong danh sách đăng ký về những thay đổi trên Subject.
- ConcreteSubject: cài đặt giao diện Subject. Vì thường là đối tượng dữ liệu, nó lưu trữ trạng thái mà các đối tượng Observer quan tâm. Khi trạng thái này thay đổi, các Observer đăng ký với nó, chứa trong một danh sách nó lưu giữ, sẽ được thông báo. Chú ý là danh sách cần thông báo do ConcreteSubject lưu giữ chứa các thực thể kiểu interface Observer, vì vậy cho phép đăng ký các đối tượng của nhiều lớp ConcreteObserver khác nhau, cài đặt phương thức đa hình update() theo cách khác nhau.
- Observer: khai báo giao diện với phương thức chính update(). Phương thức này có thể truy cập đối tượng Subject mà nó đăng ký, cập nhật Observer với trạng thái thay đổi của Subject.
- Client tạo đối tượng Subject và các Observer tách biệt rồi đăng ký các Observer cho Subject cập nhật.

### Các bước thực hiện

- Tách biệt logic nghiệp vụ thành hai phần: phần có chức năng cốt lõi, liên quan đến thay đổi dữ liệu, độc lập với các phần khác, được xem như Subject; phần phụ thuộc vào những thay đổi của Subject, được đăng ký như các Observer.
- Định nghĩa interface Observer, trong đó định nghĩa phương thức update() để cập nhật khi Subject báo có thay đổi trạng thái.
- Cài đặt cho Subject: phương thức đăng ký (addObserver) và hủy đăng ký (deleteObserver) các Observer quan tâm đến dữ liệu, phương thức thông báo (notifyObservers) cho tất cả các Observer có đăng ký.
- Trong cài đặt thực tế, ConcreteSubject thừa kế lớp dữ liệu cần quan tâm, hoặc chính nó cũng là đối tượng dữ liệu, khi đó không cần interface Subject. Nói cách khác, ConcreteSubject chứa dữ liệu cần theo dõi và chứa danh sách các Observer mà nó cần thông báo khi dữ liệu thay đổi.

Hơn nữa, khi được thông báo, Observer cập nhật chính nó hoặc cập nhật một đối tượng bất kỳ, kể cả cập nhật một phần dữ liệu của Subject, dĩ nhiên không liên quan đến dữ liệu dẫn đến cảnh báo.

Các ConcreteObserver có thể dùng tham chiếu của Subject truyền như tham số của phương thức update(Subject) để lấy thêm thông tin về trạng thái của Subject, hoặc ta truyền thông tin trạng thái như tham số của phương thức notifyObservers().

```

import java.util.ArrayList;
import java.util.List; import
java.util.Random;

```

```

// Observer
interface Observer {
void update();
}

// Subject interface Subject {
void addObserver(Observer observer); void
deleteObserver(Observer observer); void
notifyObservers();
}

// ConcreteObserver
class ConcreteObserver implements Observer {
ConcreteSubject subject;
public ConcreteObserver(ConcreteSubject subject) {
this.subject = subject;
this.subject.addObserver(ConcreteObserver.this);
}

@Override public void update() {
System.out.printf(" [%.2f]", subject.d);
}
}

// ConcreteSubject
class ConcreteSubject implements Subject {
double d;
List<Observer> observers = new ArrayList<>();
@Override public void addObserver(Observer observer) {
observers.add(observer);
}

@Override public void deleteObserver(Observer observer) {
observers.remove(observers.indexOf(observer));
}

@Override public void notifyObservers() {
observers.forEach(Observer::update);
}

public void operation() {
Random random = new Random();
d = random.nextDouble(); if (d
< 0.25 || d > 0.75) {
System.out.print("Yes");
notifyObservers(); } else
System.out.print("No");
}
}

public class Client {
public static void main(String[] args) {
System.out.println("--- Observer Pattern ---");
ConcreteSubject subject = new ConcreteSubject();
Observer observer1 = new ConcreteObserver(subject);
Observer observer2 = new ConcreteObserver(subject);
System.out.println("Doing something in the subject over time...");
System.out.println(" Observable Observer1 Observer2");
System.out.println("Iteration changed? notified? notified?");
for (int i = 0; i < 10; ++i) { System.out.print(i + ": ");
subject.operation();
System.out.println();
}
System.out.println("Removing observer1 from the subject... Repeating...");
System.out.println(" Observable Observer2");
System.out.println("Iteration changed? notified?");
subject.deleteObserver(observer1); for (int i = 0; i <
10; ++i) { System.out.print(i + ": ");
subject.operation();
System.out.println();
}
}
}

```

```
}
```

## 2. Liên quan

- Kết nối giữa Subject và Observer giống một số mẫu thiết kế:
  - + Chain of Responsibility chuyển yêu cầu theo chuỗi động các Handler cho đến khi yêu cầu được xử lý.
  - + Command thiết lập kết nối đơn hướng giữa Invoker và Receiver.
  - + Mediator loại bỏ kết nối trực tiếp giữa các Coleague, buộc họ phải liên lạc gián tiếp qua Mediator.
  - Mediator: bằng cách đóng gói những cập nhật ngữ cảnh phức tạp, Subject hoạt động như đối tượng Mediator giữa các đối tượng Observer.
  - Singleton: các Observable có thể là Singleton để nó trở nên duy nhất và được truy cập toàn cục.

## 3. Java API

Mẫu thiết kế Observer đã được tích hợp vào Java API, gói java.util. Để áp dụng mẫu thiết kế này:

- ConcreteSubject, đối tượng dữ liệu, cần thừa kế lớp Observable. Trong phương thức operation(), sau khi thay đổi dữ liệu, gọi các phương thức setChanged() và notifyObservers() của giao diện Observable để tự động cập nhật cho các đối tượng Observer có đăng ký nhận cảnh báo thay đổi với nó.
- ConcreteObserver, phần hiển thị của ứng dụng (giao diện người dùng GUI, report, sơ đồ, bảng biểu) thường cài đặt giao diện

Observer. Trong constructor, nó tự "đăng ký" để nhận cảnh báo thay đổi diễn ra trên đối tượng Observable mà nó quan tâm. Đồng thời, cài đặt phương thức update(Observable, Object), trong đó nó nhận dữ liệu thay đổi từ đối tượng Observable để cập nhật phần hiển thị của mình.

Ví dụ sau đây, viết lại ví dụ trên nhưng áp dụng mẫu thiết kế Observer được tích hợp trong Java API.

```
import java.util.Observable;
import java.util.Observer; import
java.util.Random;

class ConcreteObserver implements Observer {
public ConcreteObserver(Observable observable) {
observable.addObserver(ConcreteObserver.this);
}

@Override public void update(Observable o, Object arg) {
 if (o instanceof ConcreteSubject) {
ConcreteSubject t = (ConcreteSubject)o;
 System.out.printf(" %.2f", t.d);
}
}
}

class ConcreteSubject extends Observable {
double d; public void operation() {
 Random random = new Random();
d = random.nextDouble(); if (d
< 0.25 || d > 0.75) {
System.out.print("Yes");
this.setChanged();
this.notifyObservers(); } else
System.out.print("No");
}
}

public class Client {
public static void main(String[] args) {
 System.out.println(" --- Observer Pattern in Java API ---");
 System.out.println("Doing something in the subject over time...");
 System.out.println(" Observable Observer1 Observer2");
 System.out.println("Iteration changed? notified? notified?");
 ConcreteSubject subject = new ConcreteSubject();
 Observer observer1 = new ConcreteObserver(subject);
 Observer observer2 = new ConcreteObserver(subject);
 for (int i = 0; i < 10; ++i) {
 System.out.print(i + ": ");
 subject.operation();
 }
 System.out.println();
 System.out.println("Removing observer1 from the subject... Repeating...");
 System.out.println(" Observable Observer2");
 System.out.println("Iteration changed? notified?");
}
```

```

subject.deleteObserver(observer1);
for (int i = 0; i <
10; ++i) {
 System.out.print(i + ":");
 subject.operation();
System.out.println();
}
}
}
}

```

Tuy nhiên, có vẻ như Observer và Observable không được sử dụng nhiều. Từ Java 9, nó được đánh dấu @Deprecated (lạc hậu nhưng chưa có kế hoạch loại bỏ). Lý do là mô hình sự kiện được Observer và Observable hỗ trợ khá hạn chế, nó chỉ thông báo có cái gì đó thay đổi nhưng không cung cấp thông tin nào về những gì đã thay đổi. Khuyến nghị nên xem xét thay thế bằng cách sử dụng gói java.beans hoặc gói java.util.concurrent.

Các ví dụ khác về việc áp dụng mẫu thiết kế Observer trong Java API: java.util.EventListener và javax.jms.Topic.

#### 4. Sử dụng Ta

muốn:

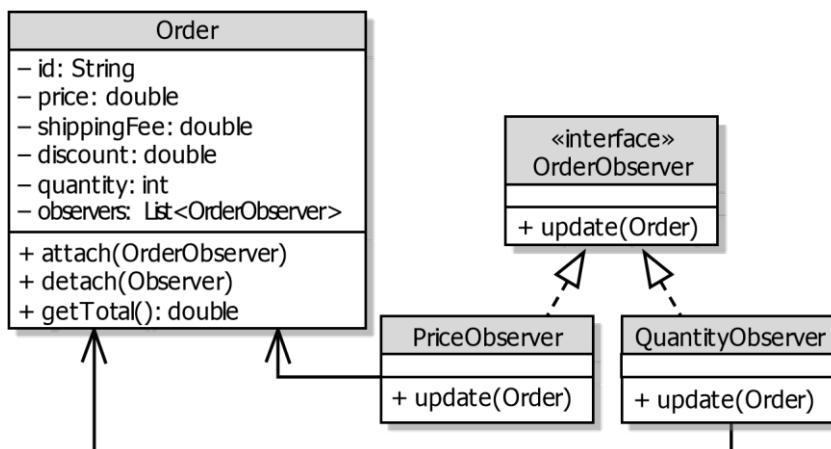
- Cập nhật trên một đối tượng (thường là dữ liệu) sẽ thay đổi một số đối tượng được lựa chọn khác (thường là giao diện người dùng), không xác định được số đối tượng thay đổi kéo theo.
- Một đối tượng cần thông báo cho một số các đối tượng khác mà không cần biết thông tin về các đối tượng được thông báo.

#### 5. Bài tập

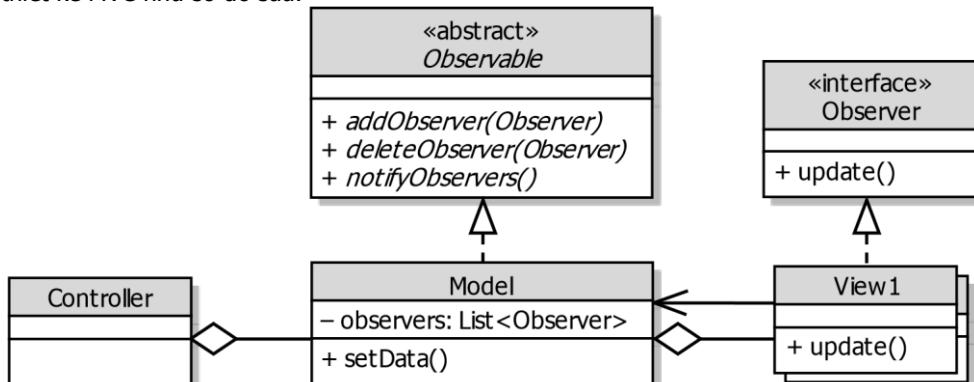
a) Người ta áp dụng mẫu thiết kế Observer để tự động cập nhật một số chi tiết đơn hàng dựa trên chính sách khuyến mãi. - Khi giá (price) mục hàng lớn hơn hoặc bằng \$500, chiết khấu (discount) cho đơn hàng là \$50. Nếu giá mục hàng lớn hơn \$200, chiết khấu cho đơn hàng là \$10.

- Khi số lượng (quantity) nhỏ hơn hoặc bằng 5, phí vận chuyển (shippingFee) là \$10. Nếu số lượng nhiều hơn 5 thu phụ phí \$1.5 với mỗi mục hàng.

Hãy thực hiện mẫu thiết kế Observer trên.



b) Sơ đồ sau trình bày mối quan hệ khi phối hợp giữa mẫu thiết kế Observer và mẫu thiết kế MVC. Hãy viết một chương trình dùng Java Swing, truy xuất cơ sở dữ liệu và hiển thị dữ liệu lên bảng và form trong GUI. Áp dụng framework Observer của Java API và mẫu thiết kế MVC như sơ đồ sau.



c) Công ty Carretta Trucking, Inc. chuyên vận chuyển hàng hóa đường dài, sử dụng xe tải với nhóm hai tài xế, một người lái xe trong khi người kia ngủ. Các xe tải thường di chuyển 10 giờ, sau đó dừng lại 2 giờ để ăn uống, tiếp nhiên liệu và nghỉ ngơi trước khi thay đổi tài xế. Công ty duy trì hai trung tâm điều phối: WestCoast và EastCoast. Tại các điểm dừng, khi xe tải đến và khi xe tải di chuyển tiếp, thông tin của xe tải được tự động chuyển về trung tâm điều phối để theo dõi tình trạng xe tải, bao cáo hành trình gồm một chuỗi các bộ thông tin sau:

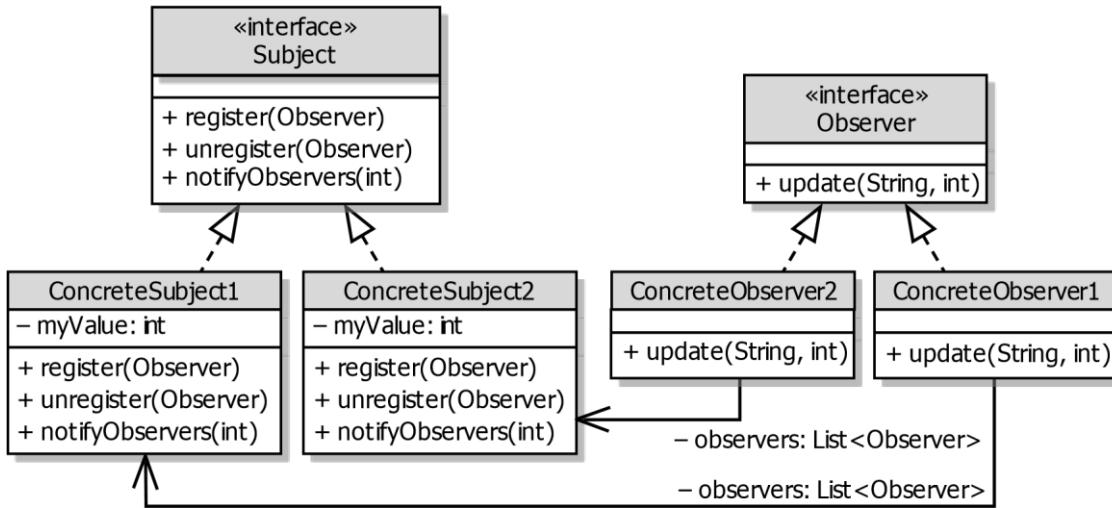
- Location: vị trí hiện tại.
- Arrive: thời gian xuất phát từ điểm dừng trước.

- Depart: thời gian đến điểm dừng hiện tại.
- Miles to destination: khoảng cách còn phải di chuyển đến đích (dặm).
- Mile between stops: khoảng cách giữa các điểm dừng (dặm).

Khi bắt đầu hành trình, trung tâm điều phối gần điểm xuất phát sẽ phát lệnh khởi hành chỉ đích đến và đăng ký trung tâm điều phối còn lại thành trung tâm theo dõi. Khi xe tải đến đích, trung tâm đang theo dõi tự loại bỏ mình khỏi danh sách các trung tâm đang theo dõi xe tải đó.

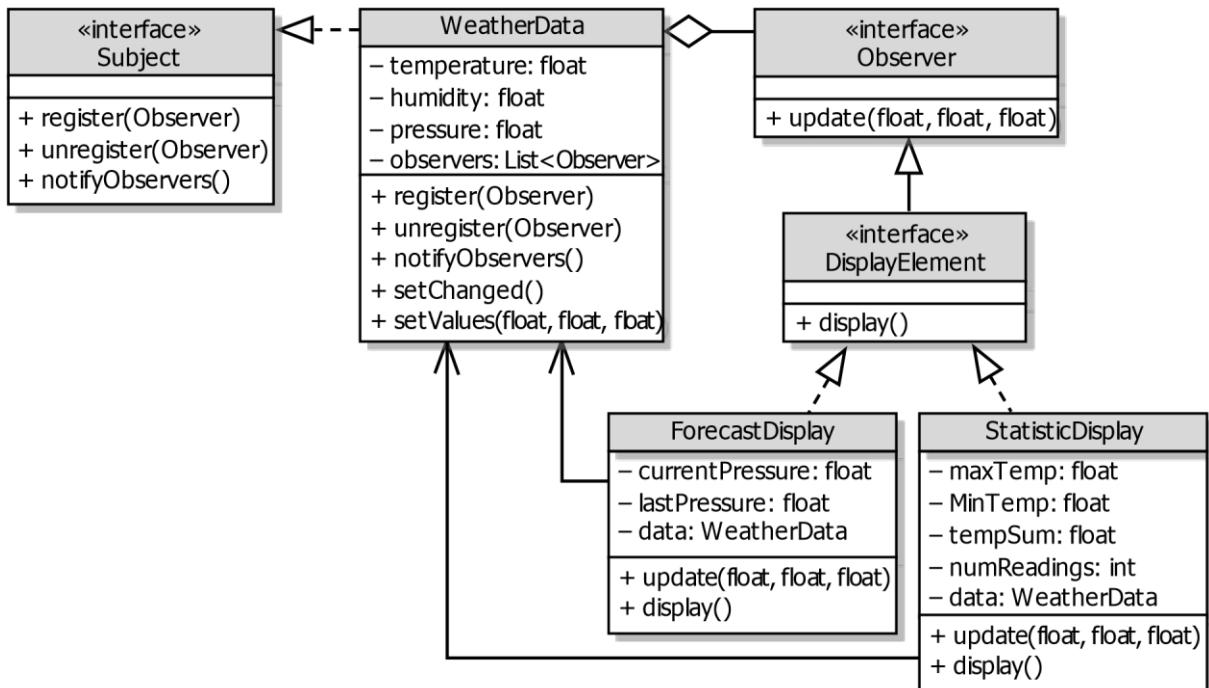
Hãy áp dụng mẫu thiết kế Observer để cập nhật thông tin các xe tải cho hai trung tâm điều phối của công ty.

- d) Thường mẫu thiết kế Observer một Subject và nhiều Observer. Cài đặt mẫu thiết kế Observer có nhiều Subject và nhiều Observer như sơ đồ sau.



- e) Thông tin thời tiết (WeatherData) bao gồm nhiệt độ (temperature), độ ẩm (humidity) và áp suất không khí (pressure) được cập nhật lên hai bảng của trạm thời tiết (WeatherStation):

- Thông tin dự báo (ForecastDisplay): so sánh áp suất không khí hiện tại với áp suất không khí thu thập được, đưa ra dự báo.
- Thông tin thống kê (StatisticalDisplay): từ thông tin thu thập, cho biết nhiệt độ cao nhất, thấp nhất và trung bình. Áp dụng mẫu thiết kế Observer để thực hiện yêu cầu trên.



## State

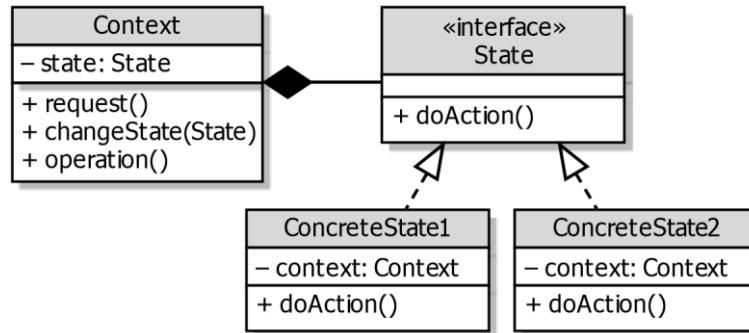
Change behavior based on state

Đối tượng có trạng thái (state) và hành vi (behavior). Đối tượng thay đổi trạng thái dựa trên các sự kiện bên trong và bên ngoài. Nếu một đối tượng mà sự thay đổi qua lại giữa các trạng thái đã được xác định rõ ràng, và hành vi của đối tượng phụ thuộc chặt chẽ vào trạng thái của nó, đối tượng đó là một ứng viên tốt cho mẫu thiết kế State.

Mẫu thiết kế State tạo một số "đối tượng trạng thái", đóng gói hành vi của đối tượng chính (gọi là Context) tương ứng với từng trạng thái (state-specific behavior). Khi trạng thái thay đổi, đối tượng chính sẽ ủy quyền thực hiện hành vi cho đối tượng trạng thái hiện hành.

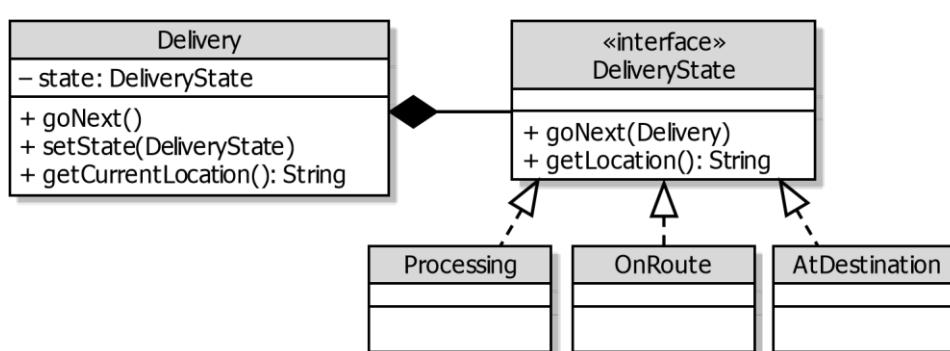
Mẫu thiết kế State định nghĩa giao diện chứa các phương thức phụ thuộc trạng thái, mỗi cài đặt cho giao diện này sẽ định nghĩa một đối tượng trạng thái với phương thức phụ thuộc trạng thái thích hợp.

## 1. Cài đặt



- Context: đối tượng Client quan tâm, ngũ cành của trạng thái hiện tại, chứa trong nó:
  - + đối tượng trạng thái State, định nghĩa trạng thái hiện tại, tham chiếu đến một đối tượng ConcreteState tại mỗi trạng thái.
  - + phương thức request() là hành vi được chỉ định bởi State hiện tại, operation() là hành vi không phụ thuộc trạng thái. + phương thức changeState() để đổi tương ứng ConcreteState chuyển trạng thái cho Context.
- State: giao diện chung đóng gói hành vi tương ứng với một trạng thái của Context.
- ConcreteState: các lớp cài đặt giao diện State, mỗi lớp cài đặt hành vi cụ thể tương ứng một trạng thái cụ thể của Context. Các hành vi này có thể thiết lập lại trạng thái mới cho Context, chuyển Context đến trạng thái kế tiếp. State và các ConcreteState thường được cài đặt như các lớp nội (inner class) của Context. **Các bước thực hiện**
- Vẽ sơ đồ chuyển trạng thái (Finite State Machine). Chuyển trạng thái sẽ diễn ra theo sơ đồ này.
- Nhận diện các trạng thái tách biệt của Context, mỗi trạng thái sẽ đóng gói trong một lớp ConcreteState, cung cấp hành vi chỉ định tương ứng với trạng thái đó. Context được thiết kế theo nguyên tắc OCP.
- Theo sơ đồ chuyển trạng thái, doAction() sẽ chuyển Context đến trạng thái kế tiếp. Trạng thái khởi đầu sẽ được Client cấu hình cho Context.
- Khi cài đặt phương thức cho lớp Context, tùy trạng thái hiện hành ta ủy nhiệm cho đối tượng ConcreteState tương ứng thực hiện tác vụ. Client tương tác với Context và không nhận biết sự có mặt của các State.

Ví dụ, việc phân phối hàng lần lượt trải qua ba trạng thái: đang xử lý (Processing), đang chuyển (OnRoute) và đã đến người nhận (AtDestination). Hai phương thức phụ thuộc trạng thái là getCurrentLocation() (báo vị trí) và goNext() (chuyển sang trạng thái tiếp sau).



```

// State
interface DeliveryState {
 void goNext(Delivery delivery);
 String getLocation();
}

// ConcreteState
class Processing implements DeliveryState {
 @Override public void goNext(Delivery delivery) {
 delivery.setState(new OnRoute());
 }

 @Override public String getLocation() { return "Warehouse"; }
}

```

```

class OnRoute implements DeliveryState {
 @Override public void goNext(Delivery delivery) {
 delivery.setState(new AtDestination());
 }

 @Override public String getLocation() { return "On the train"; }
} class AtDestination implements
DeliveryState {
 @Override public void goNext(Delivery delivery) {
 delivery.setState(this);
 }

 @Override public String getLocation() { return "Final destination"; } }

// Context class Delivery { private
DeliveryState state; public
Delivery(DeliveryState state) {
 this.state = state;
 System.out.println(getCurrentLocation());
}
public void setState(DeliveryState state) {
 this.state = state;
} public Delivery
goNext() {
 state.goNext(this);
 System.out.println(getCurrentLocation());
 return this;
}
private String getCurrentLocation() {
 return state.getLocation();
}
}
}

public class Client { public static void
main(String[] args) { System.out.println("--- State
Pattern ---"); Delivery delivery = new Delivery(new
Processing()); delivery.goNext().goNext();
}
}

```

Cài đặt ví dụ trên, dùng enum:

```

enum State {
 PROCESSING {
 @Override void goNext(Delivery delivery) { delivery.setState(ON_ROUTE); }
 @Override String getLocation() { return "Warehouse"; }
 },
 ON_ROUTE {
 @Override void goNext(Delivery delivery) { delivery.setState(AT_DESTINATION); }
 @Override String getLocation() { return "On the train"; }
 },
 AT_DESTINATION {
 @Override void goNext(Delivery delivery) { delivery.setState(this); }
 @Override String getLocation() { return "Final destination"; }
 };
 abstract void goNext(Delivery delivery);
 abstract String getLocation();
} class Delivery { private
State state; public
Delivery(State state) {
 this.state = state;
 System.out.println(state.getLocation());
}

public void setState(State state) {
 this.state = state;
} public Delivery
goNext() {
 state.goNext(this);
 System.out.println(state.getLocation());
 return this;
}
}

```

```

}
public class Client { public static void
main(String[] args) { System.out.println("---- State
Pattern ---");
Delivery delivery = new
Delivery(State.PROCESSING);
delivery.goNext().goNext();
}
}

```

## 2. Liên quan

- Flyweight: áp dụng Flyweight để các State dùng chung thuộc tính khi đó State chỉ đóng gói hành vi mà không có biến thực thể.
- Singleton: các đối tượng State thường là các Singleton để tránh tạo mới đối tượng State khi chuyển trạng thái.
- Strategy: State và Strategy có cùng cấu trúc tĩnh (sơ đồ lớp) nhưng khác nhau về mục đích. Với State, Client có ít hoặc không có kiến thức của các đối tượng State cụ thể. Trạng thái nội của Context quyết định các đối tượng State ban đầu và chuyển tiếp. Với Strategy, Client thường nhận thức của các đối tượng Strategy khác nhau và chịu trách nhiệm khởi tạo một Strategy cụ thể.

## 3. Java API

Trong JSF, FacesServlet sẽ triệu gọi và thực thi các phương thức của LifeCycle. LifeCycle lần lượt cộng tác với nhiều "phase" (JSF có 6 phase) để thực hiện yêu cầu JSF. Các "phase" này thể hiện các State trong mẫu thiết kế State.

## 4. Sử dụng Ta có:

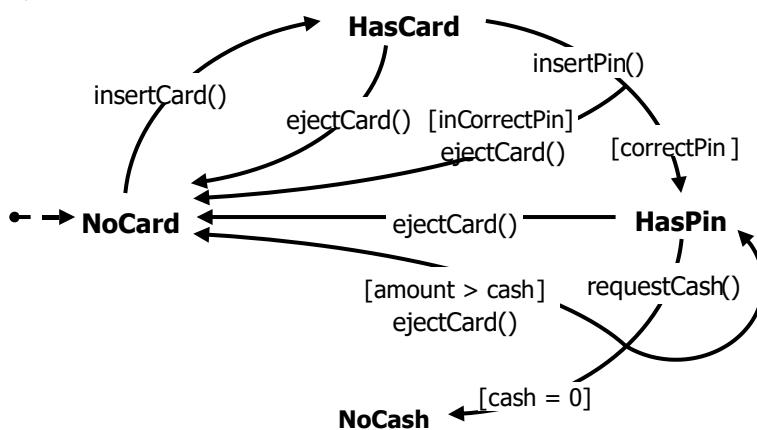
- Các đối tượng sẽ thay đổi hành vi của chúng trong thời gian chạy, dựa trên một số ngữ cảnh.
- Các đối tượng đang trở nên phức tạp với nhiều nhánh điều kiện.

Ta muốn:

- Thay đổi tập xử lý cho các yêu cầu động đến một đối tượng.
- Giữ sự linh hoạt trong việc gán các yêu cầu để xử lý.

## 5. Bài tập

- a) Khi ATM đang trong một trạng thái, người dùng có bốn tùy chọn ([insert card], [eject card], [insert PIN], [request cash]). Tùy theo trạng thái hiện hành, ATM sẽ có hành vi khác nhau.



Trạng thái NoCard: chưa đưa thẻ ATM vào máy.

- [insert card]:  $\square$  HasCard. Yêu cầu "Please enter a PIN".
- [eject card]: báo "Enter a card first".
- [insert PIN]: báo "Enter a card first".
- [request cash]: báo "Enter a card first".

Trạng thái HasCard: đã đưa thẻ ATM vào máy.

- [insert card]: báo "You can't enter more than one card".
- [eject card]: báo "Card ejected",  $\square$  NoCard.

[insert PIN]: nếu PIN đúng, báo "Correct PIN",  $\square$  HasPin. Nếu PIN sai, báo "Incorrect PIN", đẩy card ra và báo "Card ejected",  $\square$  NoCard.

[request cash]: báo "Enter PIN first" Trạng

thái HasPin: mã PIN nhập hợp lệ.

- [insert card]: báo "You can't enter more than one card".
- [eject card]: báo "Card ejected",  $\square$  NoCard.
- [insert PIN]: báo "Already entered PIN".

[request cash]: nếu số tiền yêu cầu (amount) lớn hơn tiền mặt (cash) hiện có trong máy, báo "Dont have that cash", đẩy card ra và báo "Card ejected",  $\square$  NoCard. Nếu số tiền yêu cầu hợp lệ (amount  $\square$  cash), thanh toán và vẫn ở trạng thái HasPin; mỗi lần thanh toán, kiểm tra nếu tiền mặt đã hết (cash = 0),  $\square$  NoCash. Trạng thái NoCash:

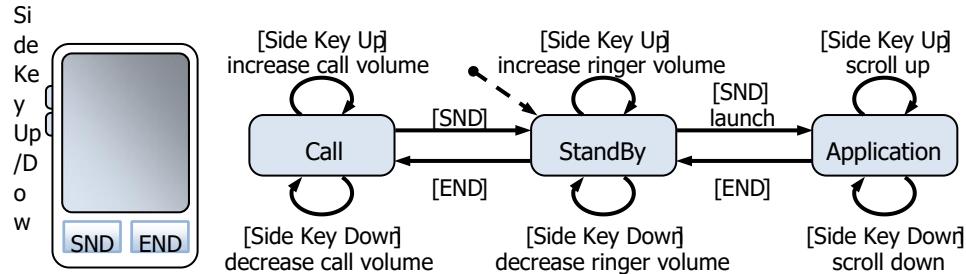
- [insert card]: báo "We don't have money".
- [eject card]: báo "We don't have money".

[insert PIN]: báo "We don't have money".

[request cash]: báo "We don't have money".

Hãy áp dụng mẫu thiết kế State để thay đổi hành vi của máy ATM khi trạng thái của nó thay đổi. Sơ đồ chuyển trạng thái sau mô tả bốn trạng thái của một máy ATM (tô đậm).

b) Thiết kế một cell phone. Do kích thước giới hạn nên chỉ có 4 phím: 2 phím bề mặt là SND và END, 2 phím trên cạnh là Side Key Up và Side Key Down. Ngoài ra, cùng một phím có thể ánh xạ những chức năng khác nhau tùy theo chế độ (trạng thái) hiện tại của thiết bị. Sơ đồ chuyển trạng thái như sau:



Chế độ StandBy: thiết bị đang trong chế độ chờ.

[SND]: chuyển qua chế độ Call.

[END]: chuyển sang chế độ Application.

[Side Key Up/Down]: tăng/giảm âm lượng chuông.

Chế độ Call: thiết bị đang trong chế độ thoại.

[SND]: báo lỗi.

[END]: chuyển sang chế độ StandBy.

[Side Key Up/Down]: tăng/giảm âm lượng thoại.

Chế độ Application: thiết bị đang trong chế độ chạy ứng dụng.

[SND]: báo lỗi.

[END]: chuyển sang chế độ StandBy.

[Side Key Up/Down]: cuộn màn hình lên/xuống.

Hãy áp dụng mẫu thiết kế State để thay đổi hành vi của cell phone khi chế độ của nó thay đổi.

Hướng dẫn:

- Cài đặt mặc định cho phương thức [SND] và [END] là báo lỗi.

- Các phương thức cho [SND] và [END] nhận tham số là đối tượng CellPhone để có thể gọi phương thức thay đổi chế độ của nó.

## Strategy

Encapsulate an algorithm

Phần dễ thay đổi nhất trong chương trình là thuật toán, trong chương trình đôi khi bạn cần lựa chọn một trong nhiều thuật toán để giải quyết vấn đề. Lưu ý, các thuật toán này có đầu vào và đầu ra giống nhau, chỉ khác cách cài đặt. Ví dụ, để sắp xếp một danh sách, bạn có thể lựa chọn nhiều "chiến lược" sắp xếp: theo tên, theo thời điểm thay đổi, theo kích thước, .v.v...

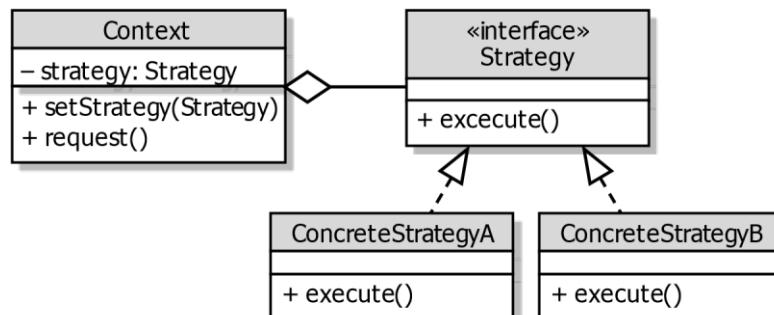
Mẫu thiết kế Strategy tách code thực thi thuật toán ra, đóng gói chúng, tạo thành một họ các thuật toán. Điều này cho phép chương trình có khả năng lựa chọn một số lượng lớn thuật toán và thay đổi động thuật toán trong thời gian chạy.

Ý tưởng là tách phần xử lý thuật toán ra khỏi đối tượng, từ đó cho phép thay đổi thuật toán độc lập với đối tượng dùng nó.

Tạo một số đối tượng thể hiện thuật toán và cho phép một đối tượng Context thay đổi đối tượng thuật toán của nó (thay đổi "chiến lược"), đối tượng thuật toán được áp dụng này sẽ thực hiện thuật toán cụ thể cho Context.

Ta dùng mẫu thiết kế này khi muốn tạo một đối tượng (Context) hỗ trợ linh hoạt nhiều thuật toán cùng một họ.

### 1. Cài đặt



- Strategy: giao diện cho họ thuật toán, đóng gói hành vi cho họ thuật toán đó. Context sử dụng giao diện này để thực thi thuật toán cụ thể được cài đặt bởi một ConcreteStrategy.

- ConcreteStrategy: cài đặt một thuật toán cụ thể, sử dụng giao diện Strategy.

- Context: lớp sử dụng nhiều biến thể khác nhau của thuật toán, bằng cách làm việc với các ConcreteStrategy thông qua giao diện Strategy. Trong lớp này có thể định nghĩa một giao diện giúp Strategy truy cập được dữ liệu của Context.

- Client tạo đối tượng Strategy chỉ định và truyền nó đến Context. Setter của Context dùng để thay thế Strategy liên kết với Context trong thời gian chạy. Cách sử dụng các đối tượng Strategy của Context theo nguyên tắc Dependency Injection, đối tượng phụ thuộc không phải tạo trước mà được "tiêm" vào khi cần.

### Các bước thực hiện

- Định nghĩa interface Strategy, mô tả "chiến lược" sẽ được chọn dùng trong Context.
- Cung cấp vài biến thể cài đặt khác nhau cho thuật toán bằng cách thực hiện các lớp ConcreteStrategy cài đặt các interface Strategy.
- Lớp Context cung cấp cách để cấu hình (lựa chọn) "chiến lược" sẽ được sử dụng, nó giúp cho code của Client gọi nó trở nên linh hoạt hơn. Context được thiết kế theo nguyên tắc OCP.

```
// Context class
Context {
 private Strategy strategy;
 public Context setStrategy(Strategy strategy) {
 this.strategy = strategy; return this;
 }
 public void request(String s) {
 System.out.println(strategy.algorithm(s));
 }
}

// Strategy
interface Strategy {
 String algorithm(String s);
}

// ConcreteStrategy
class UpperStrategy implements Strategy {
 @Override public String algorithm(String s) {
 return s.toUpperCase();
 }
}
class LowerStrategy implements Strategy {
 @Override public String algorithm(String s) {
 return s.toLowerCase();
 }
}
public class Client {
 public static void main(String[] args) {
 System.out.println("--- Strategy Pattern ---");
 Context context = new Context();
 String s = "Design Patterns";
 Strategy upperStrategy = new UpperStrategy();
 Strategy lowerStrategy = new LowerStrategy();
 context.setStrategy(lowerStrategy).request(s);
 context.setStrategy(upperStrategy).request(s);
 }
}
```

Context gọi request() với các đối tượng Strategy khác nhau, được "tiêm" vào nó. Tùy theo loại đối tượng Strategy được áp dụng, request() chạy các thuật toán tương ứng, đóng gói trong đối tượng đó.

Nhận xét, interface Strategy là SAM, nghĩa là bạn có thể đánh dấu nó là @FunctionalInterface. Hơn thế nữa, Strategy chính là một

Function. Vì thế, không nhất thiết phải cài đặt các lớp ConcreteStrategy rồi truyền chúng cho Context. Bạn có thể truyền một Function, hoặc truyền "hành vi cài đặt cho Strategy cụ thể" đến Context, dưới dạng biểu thức Lambda hoặc method reference.

```
// Context class
Context {
 private Strategy strategy;
 public Context setStrategy(Strategy strategy) {
 this.strategy = strategy; return this;
 }
 public void request(String s) {
 System.out.println(strategy.algorithm(s));
 }
}

// Strategy
@FunctionalInterface interface
Strategy {
 String algorithm(String s);
}
public class Client {
```

```

public static void main(String[] args) {
 System.out.println("--- Strategy Pattern ---");
 Context context = new Context();
 String s = "Design Patterns";
 Strategy f = str -> str.chars().mapToObj(c -> (char)c + " ").reduce("", String::concat);
 context.setStrategy(f).request(s);
 context.setStrategy(String::toUpperCase).request(s);
}
}

```

## 2. Liên quan

- Bridge: giống như Strategy, Bridge ủy nhiệm cho một đối tượng thực hiện tác vụ thực tế của nó.
- Flyweight: các đối tượng Strategy tạo từ các đối tượng Flyweight sẽ tốt hơn.
- Template Method: cũng tạo linh hoạt khi sử dụng thuật toán, nhưng Template Method dùng thừa kế trong lúc Strategy dùng Composition. Template Method làm việc ở cấp lớp nên nó static; Strategy làm việc ở cấp đối tượng, có thể thay đổi hành vi trong thời gian chạy.
- State: State và Strategy có cùng cấu trúc tĩnh (sơ đồ lớp) nhưng khác nhau về mục đích. Vẫn là một Strategy - một lớp và một State - một lớp, nhưng các Strategy không cần biết nhau, trong lúc một State có thể biết State kế tiếp.

## 3. Java API

Khi tạo GUI với một layout cụ thể, ta cung cấp một đối tượng LayoutManager cho container. Khi container cần thực hiện thuật toán bố trí các component ("chiến lược" layout), nó gọi phương thức của đối tượng LayoutManager tương ứng.

Trong trường hợp này: Context là Container; Strategy là LayoutManager; ConcreteStrategy là loại LayoutManager cụ thể (BorderLayout, GridLayout).

Một ví dụ khác là đối tượng java.util.Comparator được truyền đến phương thức sort() của Collection. Đối tượng này đóng gói thuật toán so sánh cụ thể:

```

import java.util.Collections;
import java.util.Comparator; //
...
Comparator comparator = new BookComparatorByName();
Collections.sort(books, comparator);

```

Trong đó: Context là Collections, Strategy là Comparator, ConcreteStrategy là loại Comparator cụ thể (BookComparatorByName).

## 4. Sử dụng Ta

có:

- Nhiều lớp liên quan nhau và chỉ khác nhau ở hành vi của chúng. Strategy cung cấp cách cấu hình một lớp có nhiều hành vi.
- Cần các biến thể khác nhau của thuật toán với mục đích nhất định.
- Các thuật toán dùng dữ liệu mà Client chưa biết đến. Dùng Strategy để tránh bộc lộ cấu trúc dữ liệu của thuật toán.
- Context định nghĩa nhiều hành vi, chúng xuất hiện theo các nhánh điều kiện, chuyển các hành vi này vào các Strategy.

## 5. Bài tập

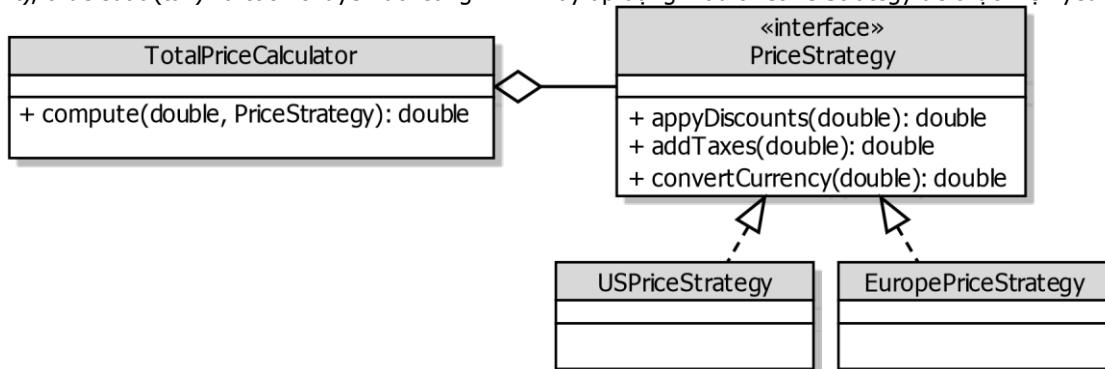
a) Một trò chơi cho phép thiết lập nhiều cấp độ chơi: EASY, MEDIUM, HARD. Cấu hình cho độ khó của trò chơi được đóng gói trong DifficultyStrategy, bao gồm những hạn chế của người chơi, số lượng và độ khó của các nhiệm vụ con phải hoàn thành, ... Tùy lựa chọn ban đầu của người chơi, chiến lược thích hợp sẽ được áp dụng. Hãy áp dụng mẫu thiết kế Strategy để thực hiện yêu cầu này.

b) Từ một cột dữ liệu liên tục trong một bảng dữ liệu, có hai cách để xuất thông tin thống kê.

- Tổng hợp số liệu (summary) bao gồm các thông tin: Min (trị nhỏ nhất), 1st Quartile (tứ phân vị thứ nhất), Median (trung vị), Mean (trung bình), 3rd Quartile (tứ phân vị thứ ba), Mode (yếu vị), Max (trị lớn nhất).
- Đồ thị phân phối tần suất (histogram).

Hãy áp dụng mẫu thiết kế Strategy để thực hiện yêu cầu này.

c) Có hai cách tính giá tiền khác nhau với loại ngoại tệ được sử dụng, đều trả về VND. Hai cách này khác nhau về chế độ giảm giá (discount), thuế suất (tax) và cách chuyển đổi sang VND. Hãy áp dụng mẫu thiết kế Strategy để thực hiện yêu cầu này.





## Template Method

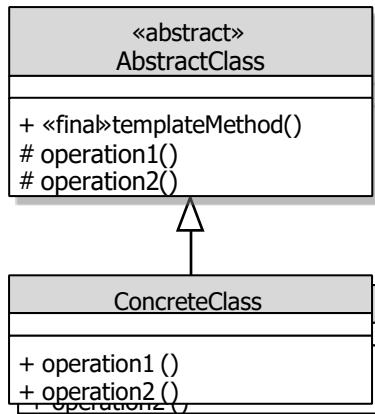
Algorithm skeleton

Mẫu thiết kế Template Method cung cấp khung sườn của một hành vi/thuật toán, được hình thành bởi một chuỗi thao tác có thứ tự hoặc không cần thứ tự. Các lớp con trong mẫu thiết kế này có thể định nghĩa lại các thao tác trong chuỗi thao tác, độc lập với cấu trúc khung sườn của hành vi/thuật toán đó. Khả năng này làm cho việc thực hiện hành vi/thuật toán trở nên linh động. Ngoài ra, mẫu thiết kế này cho phép chèn thêm một số phương thức "móc" (hook) vào chuỗi thao tác tại một số điểm đặc biệt, cho phép mở rộng hành vi/thuật toán tại các điểm đó.

Mẫu thiết kế Template Method rất phù hợp khi tạo ra các framework (khung công việc), trong đó ta cung cấp thuật toán (một kiểu framework) linh hoạt để thực hiện công việc.

Các đơn thể phần mềm có vòng đời (life cycle) bao gồm các phương thức callback, cũng được cài đặt theo mẫu thiết kế Template Method, thể hiện nguyên tắc thiết kế "inversion of control".

### 1. Cài đặt



#### - AbstractClass: chứa

- + Một phương thức cho thuật toán (templateMethod()), phương thức này thường là final để không bị viết lại (overridden), nghĩa là ngăn các lớp con thay đổi thuật toán.
- + Các phương thức trừu tượng cho các thao tác cơ bản tạo nên thuật toán. Có thể định nghĩa các thao tác cơ bản này để cung cấp hành vi mặc định. Phương thức cho thuật toán gọi các thao tác cơ bản này theo thứ tự phù hợp.

#### - ConcreteClass: lớp dẫn xuất từ AbstractClass, cài đặt cụ thể và tách biệt cho các thao tác cơ bản của thuật toán. Nói cách khác, mỗi ConcreteClass cài đặt cho một biến thể của thuật toán.

Ý tưởng là lớp cơ sở (AbstractClass) khai báo các placeholders (chỗ đặt trước) cho một thuật toán, lớp dẫn xuất (ConcreteClass) sẽ lựa chọn hiện thực cho các placeholders này.

Thường có ba kiểu tác vụ khác nhau được gọi từ templateMethod():

```
abstract class AbstractClass {
 public final void templateMethod() {
 operation1(); operation2();
 operation3();
 } protected abstract void
 operation1(); protected void
 operation2() { } protected final void
 operation3() { } }
```

+ operation1(): tác vụ trừu tượng (abstract step) được khai báo (và sử dụng) trong lớp cơ sở; lớp con sẽ định nghĩa cụ thể nó.

+ operation2(): tác vụ được định nghĩa trong lớp cơ sở, tác vụ này có thể cài đặt rỗng hoặc được cài đặt mặc định (optional step). Tùy nhu cầu mở rộng thuật toán, lớp con có thể định nghĩa lại hoặc không cần định nghĩa lại tác vụ này. Chúng gọi là các phương thức "hook", do lớp con có thể "móc" vào thuật toán tại một số điểm.

+ operation3(): tác vụ không thay đổi, định nghĩa bước bắt buộc phải có trong thuật toán. **Các bước thực hiện**

- Định nghĩa thuật toán cho phương thức templateMethod(). Chia thuật toán này thành nhiều bước, mỗi bước sẽ trở thành một phương thức abstract trong AbstractClass. Việc chia thuật toán thành nhiều bước giúp cho quá trình thực hiện thuật toán trở nên linh hoạt, dễ tùy biến.

- Cài đặt từng phương thức abstract này (từng bước của thuật toán) trong một hay nhiều lớp ConcreteClass, dẫn xuất từ Abstract Class.

```
import java.util.regex.Pattern;
```

```
class Message {
 String address;
 String subject;
 String content;

 public Message(String address, String subject, String content) {
 this.address = address; this.subject = subject;
 this.content = content;
```

```

}

// AbstractClass abstract
class MessageSender {
protected Message message;
public final void execute(Message message) {
this.message = message;
initialize().sendMessage().cleanUp();
} protected abstract MessageSender
initialize(); protected abstract MessageSender
sendMessage(); protected abstract MessageSender
cleanUp();
}

// ConcreteClass
class EmailMessageSender extends MessageSender {
private boolean status = false;
private String log = "Send email message failed";
private boolean isEmail(String address) {
 return Pattern.compile("^[A-Z0-9._%+-]+@[A-Z0-9.-]+\\.[A-Z]{2,6}$",
Pattern.CASE_INSENSITIVE).matcher(address).find();
}

@Override protected MessageSender initialize() {
status = isEmail(message.address); return
this;
}

@Override protected MessageSender sendMessage() {
if (status) {
 System.out.println("Sending by email...");
 log = "Send message to " + message.address + " successful";
} return
this;
}

@Override protected MessageSender cleanUp() {
status = false;
System.out.println("[LOG]: " + log);
return this;
}
}

class HttpPostMessageSender extends MessageSender {
private boolean status = false;
private String log = "Send HTTP message failed";
private boolean isURL(String address) {
 return Pattern.compile("\b(https?|ftp|file):\/\/[-a-zA-Z0-9+&@#/%?=~_|!:,.;]*[-a-zA-Z0-
9+&@#/%=~_|]", Pattern.CASE_INSENSITIVE).matcher(address).find();
}

@Override protected MessageSender initialize() {
status = isURL(message.address); return this;
}

@Override protected MessageSender sendMessage() {
if (status) {
 System.out.println("Sending by HTTP post...");
 log = "Send message to " + message.address + " successful";
}
return this;
}

@Override protected MessageSender cleanUp() {
status = false;
System.out.println("[LOG]: " + log);
return this;
}
}

```

```

}

public class Client {
 public static void main(String[] args) {
 System.out.println("--- Template Method Pattern ---");
 Message eMessage = new Message("billgates@microsoft.com", "to Bill", "Hello Bill!");
 Message wMessage = new Message("http://oracle.com/forum", "to James", "Hello James!");
 new EmailMessageSender().execute(eMessage); new
 HttpPostMessageSender().execute(wMessage);
 }
}

```

Quy trình gửi một thông điệp, bằng email hoặc bằng web, phải qua một số bước: khởi tạo (initialize), gửi (sendMessage) và dọn dẹp (cleanUp). Thao tác cho các bước này được cài đặt độc lập với quy trình gửi thông điệp, phù hợp với từng loại thông điệp. Template Method cho phép tiêm vào các lời gọi phương thức theo thứ tự do Template Method cung cấp. Các lời gọi này có thể truyền đơn giản bằng cách dùng Runnable. Viết lại ví dụ trên theo ý tưởng này:

```

import java.util.regex.Pattern;

class Message {
 String address, subject, content;

 public Message(String address, String subject, String content) {
 this.address = address; this.subject = subject;
 this.content = content;
 }
}

// AbstractClass abstract
class MessageSender {
 protected Message message;
 protected Runnable initialize, sendMessage, cleanUp;
 protected MessageSender() { }

 public final void execute(Message message) {
 this.message = message; initialize.run();
 sendMessage.run(); cleanUp.run();
 }
}

// ConcreteClass
class EmailMessageSender extends MessageSender {
 private boolean status = false;
 private String log = "Send email message failed";
 private boolean isEmail(String address) {
 return Pattern.compile("[A-Z0-9._%+-]+@[A-Z0-9.-]+\\.[A-Z]{2,6}$",
 Pattern.CASE_INSENSITIVE).matcher(address).find();
 }
 public EmailMessageSender() {
 initialize = () -> status = isEmail(message.address);
 sendMessage = () -> { if (status) {
 System.out.println("Sending by email...");
 log = "Send message to " + message.address + " successful";
 } };
 cleanUp = () -> {
 status = false;
 System.out.println("[LOG]: " + log);
 };
 }
}
class HttpPostMessageSender extends MessageSender {
 private boolean status = false;
 private String log = "Send HTTP message failed";
 private boolean isURL(String address) {
 return Pattern.compile("\\b(https?|ftp|file):\/\/[-a-zA-Z0-9+&#/%?=~_|!:,.;]*[-a-zA-Z0-9+&#/%=~_|]", Pattern.CASE_INSENSITIVE).matcher(address).find();
 }

 public HttpPostMessageSender() {
 initialize = () -> status = isURL(message.address);
 sendMessage = () -> { if (status) {
 System.out.println("Sending by HTTP post...");
 log = "Send message to " + message.address + " successful";
 } };
 }
}

```

```

 }
 }
}

public class Client {
 public static void main(String[] args) {
 System.out.println("--- Template Method Pattern ---");
 Message eMessage = new Message("billgates@microsoft.com", "to Bill", "Hello Bill!");
 Message wMessage = new Message("http://oracle.com/forum", "to James", "Hello James!");
 new EmailMessageSender().execute(eMessage); new
 HttpPostMessageSender().execute(wMessage);
 }
}

```

## 2. Liên quan

- Factory Method: một trong các bước của Template Method thường là tạo đối tượng, khi đó nó dùng Factory Method.
- Strategy: Template Method dùng thừa kế để thay đổi một phần của thuật toán, Strategy dùng ủy nhiệm để thay đổi hoàn toàn thuật toán.

- Observer: mẫu thiết kế Observer thường được dùng kết hợp với mẫu thiết kế Template Method như ví dụ sau.

```

abstract class ProcessManager extends Observable {
protected final void process() { try {
doProcess(); setChanged();
notifyObservers(); } catch (Throwable t) {
 Log.error("ProcessManager.process(): ", t);
}
abstract protected void doProcess(); }

```

Khi áp dụng mẫu thiết kế Template Method, sau khi doProcess() làm thay đổi dữ liệu thì các Observer đăng ký với ProcessManager sẽ được thông báo và thay đổi phần hiển thị tương ứng.

## 3. Java API

Tất cả các phương thức non-abstract của java.io.InputStream, java.io.OutputStream, java.io.Reader và java.io.Writer.

Tất cả các phương thức non-abstract của java.util.AbstractList, java.util.AbstractSet và java.util.AbstractMap. Các phương thức non-abstract trên giữ vai trò giống như templateMethod().

## 4. Sử dụng Ta muốn:

- Cài đặt những phần không thay đổi của một thuật toán trong một lớp đơn và những phần thay đổi của thuật toán trong các lớp dẫn xuất từ lớp đơn đó.
- Hành vi chung của các lớp dẫn xuất được chuyển đến một lớp đơn duy nhất để tránh trùng lặp.

## 5. Bài tập

a) Quy trình kiểm tra tính hợp lệ của ba loại credit card (Visa, MasterCard, Diners Club) đều có 6 bước, trong đó có một số bước giống nhau (1, 4, 5) và một số bước khác nhau (2, 3, 6).

| Bước | Kiểm tra                                        | Visa            | MasterCard         | Diners Club         |
|------|-------------------------------------------------|-----------------|--------------------|---------------------|
| 1    | Hạn sử dụng (Expiration date)                   | > today         | > today            | > today             |
| 2    | Chiều dài dãy số (Length)                       | 13, 16          | 16                 | 14                  |
| 3    | Dãy số đầu (Prefix)                             | 4               | 51 – 55            | 30, 36, 38          |
| 4    | Ký tự hợp lệ (Valid characters)                 | 0 – 9           | 0 – 9              | 0 – 9               |
| 5    | Thuật toán kiểm tra (Check digit algorithm)     | thuật toán Luhn | thuật toán Luhn    | thuật toán Luhn     |
| 6    | Trạng thái tài khoản (Account in good standing) | gọi Visa API    | gọi MasterCard API | gọi Diners Club API |

Hãy áp dụng mẫu thiết kế Template Method cho việc kiểm tra tính hợp lệ cả ba loại card trên. Thuật toán kiểm tra (Check digit algorithm) gọi là thuật toán Luhn:

- Duyệt dãy số từ phải sang trái, số tại vị trí có thứ tự chẵn thì nhân đôi.

```

1 9 4 7 7 4 9 1 5
1 9x2 4 7x2 7 4x2 9 1x2 5
1 18 4 14 7 8 9 2 5

```

- Nếu kết quả nhân đôi này có hai chữ số thì lấy tổng hai chữ số đó làm kết quả cuối.

```

1 1+8 4 1+4 7 8 9 2 5
1 9 4 5 7 8 9 2 5

```

- Cộng các số cuối cùng này lại với nhau. Nếu kết quả chia hết cho 10 thì dãy số kiểm tra là hợp lệ.

$1 + 9 + 4 + 5 + 7 + 8 + 9 + 2 + 5 = 50 \square \text{valid}$

```

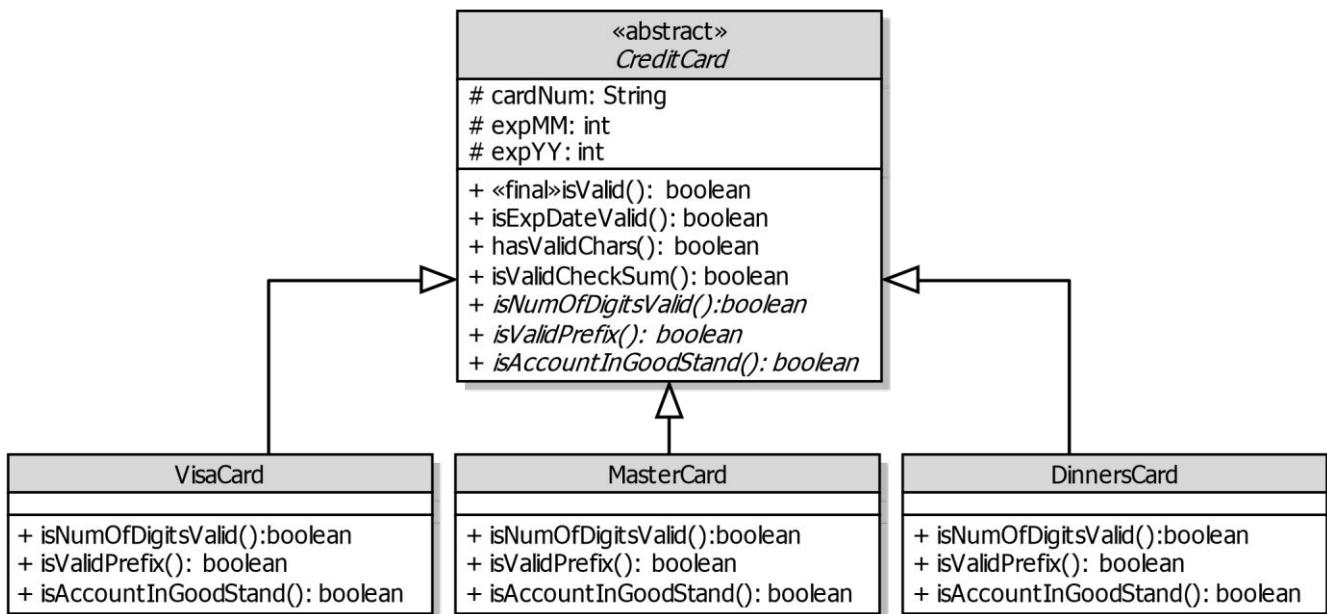
public boolean isLuhn(String cardNo) {
return IntStream.range(0, cardNo.length())

```

```

 .map(i -> (i % 2 != 0 ? 2 : 1) * (cardNo.charAt(i) - '0'))
 .map(i -> i > 9 ? i - 9 : i)
 .sum() % 10 == 0;
}

```



b) Thông tin một đơn hàng (Order) bao gồm (id, date, items, total); trong đó items là một Map chứa danh sách các mục đơn hàng. Quy trình in đơn hàng nhiều bước, định dạng các phần khác nhau của đơn hàng: - header

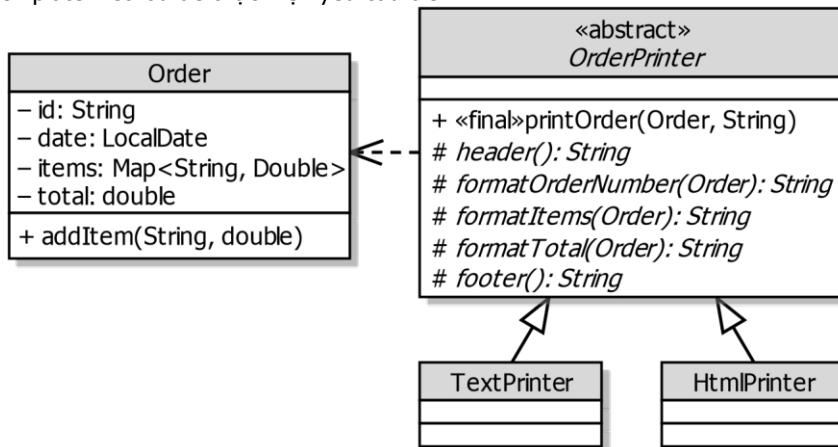
- mã số đơn hàng

- danh sách các mục đơn hàng

- tổng giá trị đơn hàng

- footer

Sau các bước định dạng, đơn hàng sẽ xuất ra tập tin văn bản hoặc tập tin HTML. Định dạng cho hai tập tin này khác nhau. Hãy áp dụng mẫu thiết kế Template Method để thực hiện yêu cầu trên.



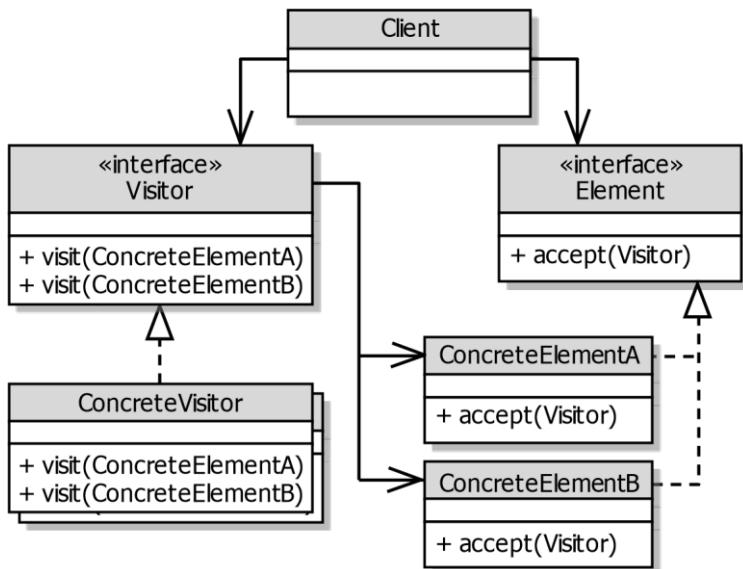
## Visitor

[Visit tree node](#)

Một cách để thực hiện một tác vụ lên các đối tượng khác nhau trong một cấu trúc phức hợp là cung cấp thao tác đó riêng cho từng lớp của chúng. Thay vì tiếp cận từ bên ngoài vào như trên, có thể đảo ngược cách tiếp cận: gửi một đối tượng vào trong cấu trúc phức hợp và để cho đối tượng đó làm việc, mà vẫn không làm thay đổi các lớp của cấu trúc phức hợp.

Mẫu thiết kế Visitor đóng gói tất cả những thao tác cần thiết cho tác vụ "viếng thăm" (print, upgrade, render, display, sum, ...) vào một lớp riêng. Đối tượng thuộc lớp này, gọi là `Visitor`, chứa tác vụ "viếng thăm" phù hợp với lớp đối tượng tiếp nhận nó. Các đối tượng `Element`, tức các đối tượng sẽ được "viếng thăm" trong cấu trúc phức hợp, phải chấp nhận "tiêm" đối tượng `Visitor` vào chúng, bằng phương thức `accept()`, để đối tượng `Visitor` đó có thể thực hiện tác vụ "viếng thăm" trên `Element` được. Như vậy, cách sử dụng các đối tượng `Visitor` tuân theo nguyên tắc Dependency Injection.

### 1. Cài đặt



- Visitor: định nghĩa giao diện chứa các phương thức visit() cho từng lớp ConcreteElement trong cấu trúc phức hợp. Tùy loại Element, Visitor sẽ "viếng thăm" bằng phương thức visit() tương ứng.
- ConcreteVisitor: cài đặt các tác vụ khai báo trong Visitor, thể hiện thuật toán "viếng thăm" cụ thể ứng với Element cụ thể: đếm số Element, hiển thị nội dung Element, tính toán tích lũy, ... số các tác vụ này có thể mở rộng.
- Element: khai báo giao diện chung cho Element, quan trọng là phương thức accept(), nhận Visitor như đối số. Chính phương thức accept() này sẽ gọi các phương thức visit() của Visitor mà nó tiếp nhận.
- ConcreteElement: các loại Element cụ thể, thường tạo nên cấu trúc phức hợp. Số ConcreteElement là cố định.
- Client thường làm việc với một đối tượng phức hợp (như cây Composite). Client không nhận thấy các lớp Element cụ thể do nó làm việc với đối tượng phức hợp thông qua interface. **Các bước thực hiện**
- Tạo interface Visitor định nghĩa tập hợp các phương thức visit() cho từng lớp ConcreteElement muốn được hỗ trợ. Do Visitor có thể làm việc với các đối tượng không cần có cha chung, nên interface Element là tùy chọn.
- Cài đặt phương thức accept(Visitor) cho các lớp ConcreteElement muốn Visitor "viếng thăm", để chấp nhận "tiêm" Visitor vào nó. Trong phương thức accept(), gọi phương thức của Visitor để "viếng thăm" lớp đó.
- Cài đặt giao diện Visitor cho các ConcreteVisitor. Lưu ý phương thức visit() sẽ truy cập đến trạng thái nội của ConcreteElement tương ứng. Nếu phương thức visit() liên quan đến tích lũy, Các ConcreteVisitor có thể cần có trạng thái nội để lưu kết quả tích lũy.

```

import java.util.ArrayList;
import java.util.Arrays; import
java.util.List;

// Element interface
FileSystemNode { void
accept(Visitor visitor);
}

// ConcreteElement
class FileNode implements FileSystemNode {
 String name;

 public FileNode(String name) {
this.name = name;
 }

 @Override public void accept(Visitor visitor) {
visitor.visit(this);
 }
}

class FolderNode implements FileSystemNode {
 String name;

 List<FileSystemNode> list = new ArrayList<>();
 public FolderNode(String name, FileSystemNode... children) {
this.name = name;
 Arrays.stream(children).forEach(list::add);
}
}

```

```

@Override public void accept(Visitor visitor) {
 visitor.visit(this);
}
public FolderNode add(FileSystemNode node) {
 list.add(node); return this;
}
}

// Visitor interface Visitor {
void visit(FileNode element);
void visit(FolderNode element);
}

// ConcreteVisitor
class PrintVisitor implements Visitor {
@Override public void visit(FileNode node) {
 System.out.println(node.name);
}

@Override public void visit(FolderNode node) {
 System.out.println("[" + node.name + "]");
 node.list.forEach(e -> {
 if (e.getClass() == FileNode.class) visit((FileNode)e);
 else if (e.getClass() == FolderNode.class) visit((FolderNode)e);
 });
}
}

public class Client {
 public static void main(String[] args) {
 System.out.println("--- Visitor Pattern ---");
 FolderNode root = new FolderNode("java",
 new FileNode("readme.txt"),
 new FolderNode("javaSE",
 new FileNode("code.java"),
 new FolderNode("tutorial")),
 new FolderNode("javaEE",
 new FileNode("web.pdf"),
 new FileNode("ejb.pdf")));
 root.accept(new PrintVisitor());
 }
}

```

Chú ý constructor của FolderNode, phương thức này cho phép tạo thành một cấu trúc phức hợp tương tự cây thư mục, bao gồm FileNode và FolderNode; FolderNode có thể chứa các FileNode và FolderNode khác. Client chọn một PrintVisitor cài đặt các phương thức visit(), hiển thị thông tin tùy FileNode hoặc FolderNode, trong đó phương thức visit() của FolderNode là đệ quy. Client gửi PrintVisitor này đến cấu trúc phức hợp bằng cách truyền PrintVisitor cho phương thức accept().

## 2. Liên quan

- Composite: thường được dùng để định nghĩa cấu trúc đối tượng phức hợp mà Visitor áp dụng tác vụ "viếng thăm" lên đó.
- Interpreter: Visitor cũng có thể hỗ trợ Interpreter thực hiện tác vụ diễn dịch của nó.

## 3. Java API

javax.lang.model.element.AnnotationValue  
e và AnnotationValueVisitor.  
javax.lang.model.element.Element và ElementVisitor.  
javax.lang.model.type.TypeMirror và TypeVisitor.  
javax.lang.model.element.Element and ElementVisitor.  
java.nio.file.FileVisitor và SimpleFileVisitor.

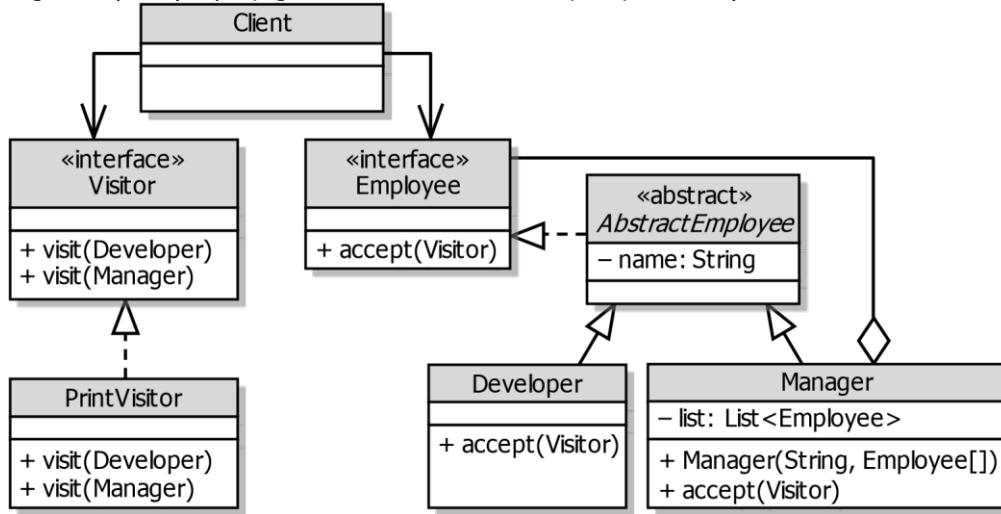
## 4. Sử dụng Ta muốn:

- Duyệt qua một hệ thống phân lớp phức tạp được bảo vệ chặt chẽ, khó thay đổi.
- Một hệ thống cần nhiều tác vụ khác nhau thực hiện trên nó, có thể mở rộng số tác vụ này. Dùng như một parser.
- Các tác vụ gắn liền với các loại đối tượng trong hệ thống phân cấp.
- Có một phương thức tích lũy thông tin từ các lớp khác nhau. Chuyển tác vụ tích lũy này đến một Visitor, nó có thể viếng thăm từng lớp để tích lũy thông tin.

## 5. Bài tập

- a) Developer và Manager đều là các Employee, nhưng Manager quản lý một danh sách (có thể rỗng) các Employee khác, bao gồm các Developer và các Manager khác dưới quyền. Cung cấp một Visitor "tiêm" vào các Employee để Employee xuất

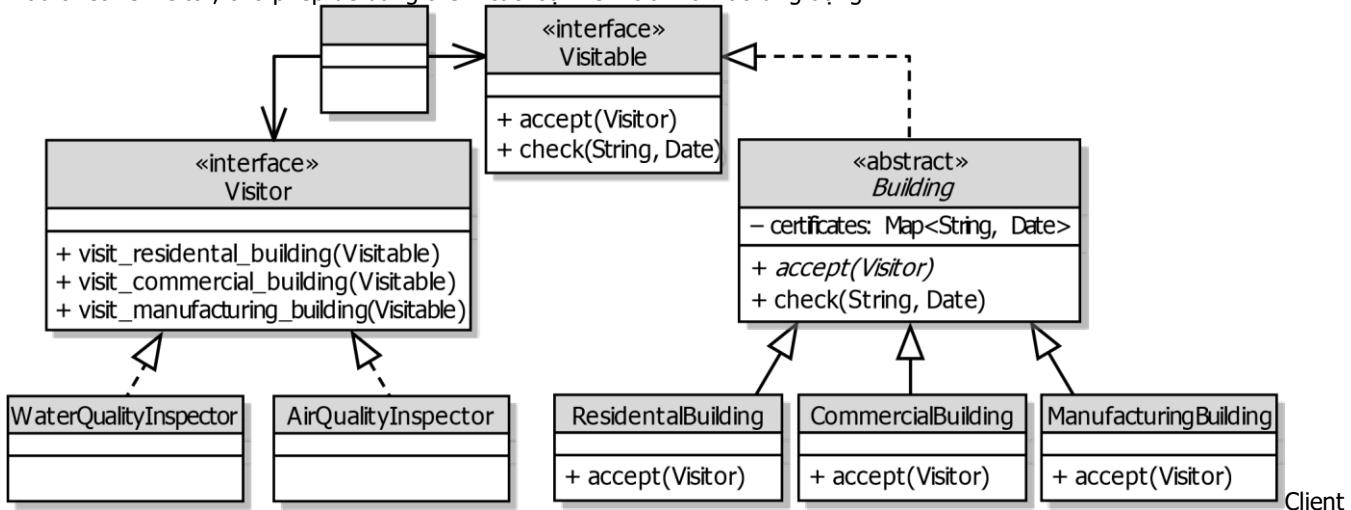
thông tin khi duyệt đến. Với Developer, chỉ in ra thông tin cơ bản. Với Manager, ngoài thông tin cơ bản, còn in thông tin về các Employee do Manager đó quản lý. Áp dụng mẫu thiết kế Visitor để thực hiện điều này.



b) Cục Công trình công cộng của thành phố Galveston, New York thực hiện một chính sách an toàn đổi mới toàn diện, theo đó tất cả các tòa nhà trong phạm vi thành phố sẽ được kiểm tra 5 năm một lần. Một cơ sở dữ liệu trung tâm lưu giữ địa chỉ của mỗi tòa nhà cùng với giấy chứng nhận kiểm định và ngày cấp cho từng loại kiểm tra được thực hiện cho tòa nhà. Lúc đầu, các loại kiểm tra được thực hiện chỉ giới hạn ở kiểm định chất lượng nước (water quality) và chất lượng không khí (air quality). Các loại tòa nhà được kiểm tra được chia thành ba loại: dân cư, thương mại và sản xuất. Mỗi loại tòa nhà đều có yêu cầu riêng để kiểm tra. Việc kiểm tra tòa nhà dân cư thường mất một giờ; kiểm tra tòa nhà thương mại cần ba giờ để hoàn thành; và kiểm tra tòa nhà sản xuất kéo dài tám giờ.

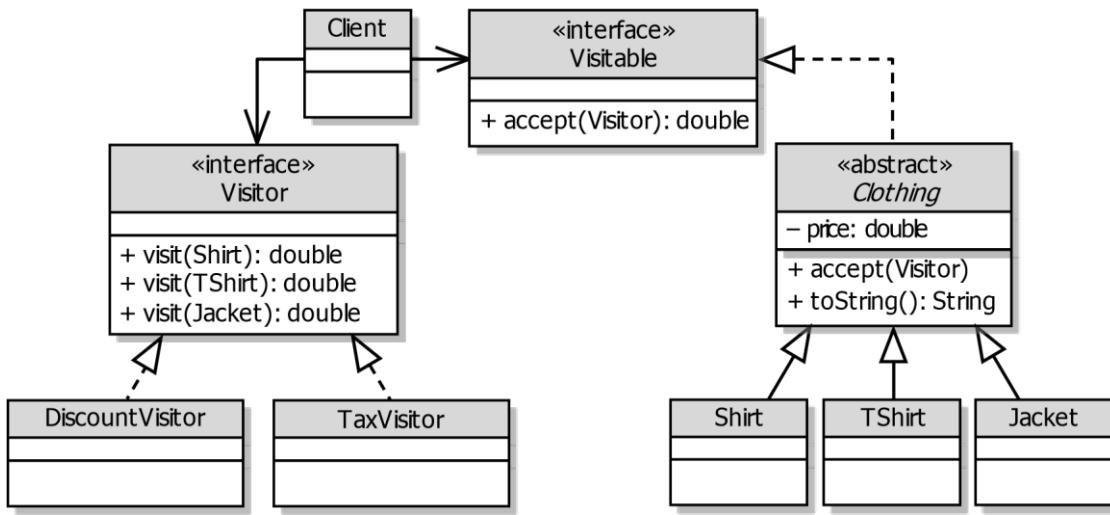
Tuy nhiên, sau khi Hội đồng thành phố mới được bầu, họ quyết định mở rộng chính sách kiểm tra: kiểm tra xâm nhiễm côn trùng (insect infestations) và kiểm tra an toàn hỏa hoạn (fire safety).

Để tránh những thay đổi có thể có trong tương lai làm vi phạm nguyên tắc OCP, phần mềm quản lý được thiết kế lại, sử dụng mẫu thiết kế Visitor, cho phép dễ dàng thêm các loại kiểm tra mới vào ứng dụng.

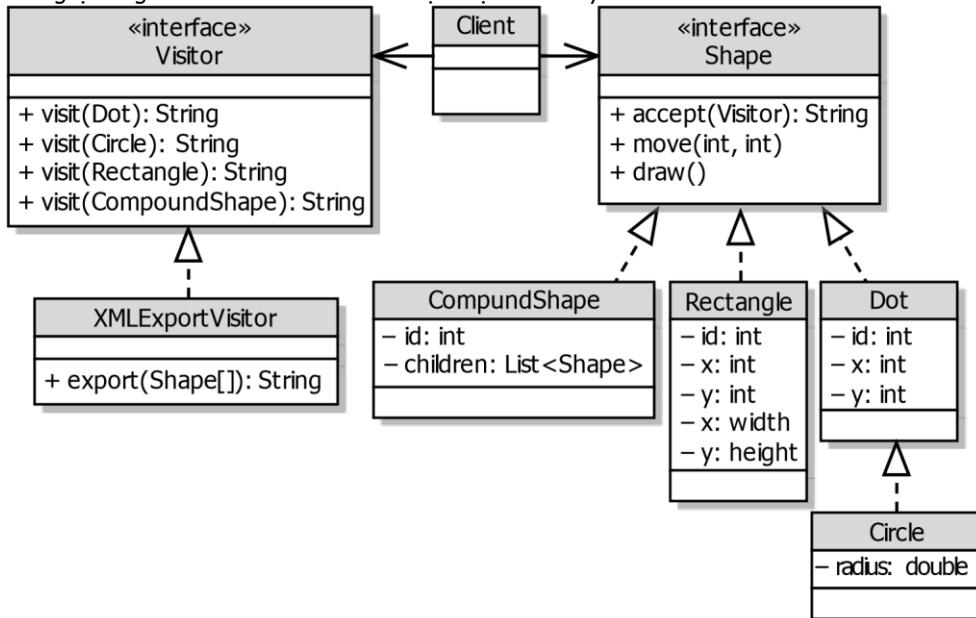


c) Cung cấp một Visitor truy cập hệ thống các tập tin và thư mục từ một vị trí chỉ định, in ra tên của các tập tin và thư mục so trùng với một mẫu biểu thức chính quy (regular expression) chỉ định. Giải quyết vấn đề nếu hệ thống hỗ trợ cả link. Link được dùng trên Unix, tương tự shortcut trên Windows, là một bản sao ảo của tập tin hoặc thư mục, chứa tên link và đường dẫn của tập tin hoặc thư mục mà nó đại diện. Link có các tác vụ giống như tập tin và thư mục, ngoại trừ việc ta có thể xóa nó mà không ảnh hưởng đến tập tin hoặc thư mục mà nó đại diện. Hướng dẫn: áp dụng mẫu thiết kế Proxy cho link.

d) Cửa hàng bán trang phục chỉ có ba mặt hàng với giá gốc khác nhau: Shirt, Tshirt và Jacket. Mỗi mặt hàng có chiết khấu (discount) và thuế (tax) khác nhau. Áp dụng mẫu thiết kế Visitor để tính giá cho một danh sách các mặt hàng trang phục. Lưu ý, chiết khấu được tính trước, sau đó mới tính thuế.



e) Chúng ta muốn kết xuất (export) một tập các đối tượng hình học (Shape) thành XML. Mỗi đối tượng vào một tag với các tag con là các thuộc tính của đối tượng hình học đó. Để cho phép thêm bất kỳ hành vi nào vào nhánh Shape mà không thay đổi việc kết xuất, đề nghị dùng mẫu thiết kế Visitor để thực hiện điều này.



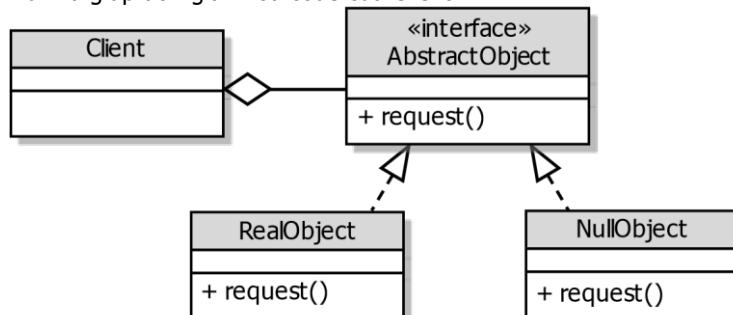
## Các mẫu thiết kế mở rộng

### Null Object

Provides "do nothing" behavior

Trong hầu hết các ngôn ngữ hướng đối tượng, các tham chiếu có thể là null. Các tham chiếu này cần phải được kiểm tra để đảm bảo chúng không phải là null trước khi gọi bất kỳ phương thức nào, bởi vì tham chiếu null không chỉ đến đối tượng nào nên không thể gọi các phương thức trên chúng được.

Ý tưởng là thay trả về null để mô tả sự vắng mặt của một đối tượng, trả về một đối tượng Null có cùng giao diện với đối tượng cần kiểm tra null. Đối tượng Null cung cấp hành vi "không làm gì" cho trường hợp thiếu đối tượng. Mẫu thiết kế Null Object loại bỏ sự cần thiết phải kiểm tra null và giúp đơn giản hóa code của Client.



Client sử dụng giao diện AbstractObject để tương tác với các lớp công tác. Nếu đối tượng nhận yêu cầu là một RealObject, thì yêu cầu được xử lý bằng cách cung cấp hành vi thực sự. Nếu đối tượng nhận là một NullObject, yêu cầu được xử lý bằng cách cung cấp hành vi "không làm gì" hoặc ít nhất là cung cấp một kết quả null.

Do Client đối xử nhất quán với RealObject và NullObject nên code của Client trở nên đơn giản. NullObject không chứa trạng thái nên thường được cài đặt như một Singleton.

#### Ví dụ minh họa

Cho cây nhị phân, cấu trúc như hình kèm theo. Các node "thật" được ghi chú với tên node và kích thước của cây con mà node đó là node gốc. Các node "null" được tô màu xám.

Phương thức duyệt cây traversal() hoạt động dựa trên phương thức getTreeSize(). Trong phương thức getTreeSize(), do áp dụng mẫu thiết kế Null Object, đối xử với các node là nhất quán, nên không cần phải kiểm tra null.

```
interface Node { String getName(); int getTreeSize();
 Node getLeft(); Node getRight(); void traversal();}
}
class RealNode implements Node {
 String name;
 Node left, right;

 public RealNode(String name, Node left, Node right) {
 this.name = name; this.left = left; this.right = right;
 }

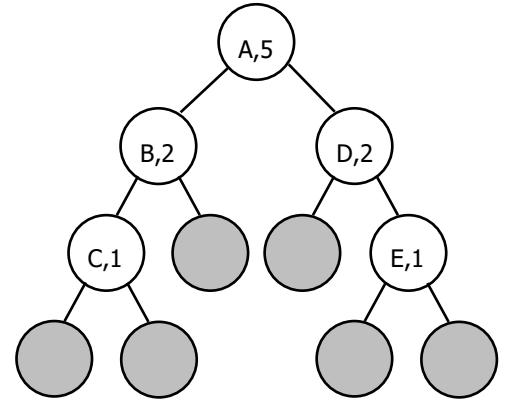
 @Override public int getTreeSize() {
 return 1 + left.getTreeSize() + right.getTreeSize();
 }

 @Override public Node getLeft() { return left; }
 @Override public Node getRight() { return right; }
 @Override public String getName() { return name; }

 @Override public void traversal() {
 System.out.print("[" + name + ", " + getTreeSize() + "] ");
 if (left.getTreeSize() > 0) left.traversal(); if
(right.getTreeSize() > 0) right.traversal();
 }
}
class NullNode implements Node {
 static NullNode instance = new NullNode();

 private NullNode() {}
 public static NullNode getInstance() { return instance; }

 @Override public int getTreeSize() { return 0; }
 @Override public Node getLeft() { return null; }
 @Override public Node getRight() { return null; }
 @Override public String getName() { return null; }
 @Override public void traversal() {}
}
public class Client {
 public static void main(String[] args) {
 Node root = new RealNode("A",
 new RealNode("B", new RealNode("C", NullNode.getInstance(), NullNode.getInstance()),
 NullNode.getInstance()),
 new RealNode("D", NullNode.getInstance(),
 new RealNode("E", NullNode.getInstance(), NullNode.getInstance())));
 root.traversal(); // [A, 5] [B, 2] [C, 1] [D, 2] [E, 1]
 }
}
```



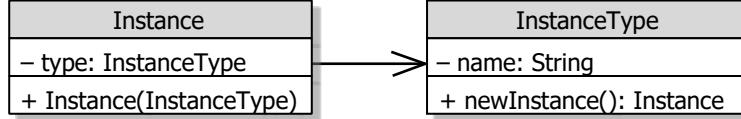
## Type Object

Types as objects

Khi bạn cần định nghĩa một số lượng lớn các "kiểu" khác nhau của một lớp chỉ định, cách tiếp cận kinh điển tạo cây thừa kế các "kiểu" mới dẫn xuất từ lớp chỉ định sẽ không hiệu quả vì tạo ra cây thừa kế lớn. Ngoài ra:

- Bạn không biết "kiểu" mới nào sẽ cần trong tương lai.
- Bạn muốn sửa đổi hoặc thêm các "kiểu" mới mà không phải biên dịch lại hoặc thay đổi mã.

Gọi lớp chỉ định là Instance, ý tưởng là đóng gói thông tin về "kiểu" của lớp Instance vào một lớp đơn InstanceType, mỗi thể hiện (instance) của lớp này đại diện cho một "kiểu" khác nhau. Đối tượng của lớp Instance chứa tham chiếu đến đối tượng lớp InstanceType mô tả "kiểu" của nó. Các đối tượng lớp Instance tham chiếu đến cùng một đối tượng "kiểu" InstanceType sẽ hoạt động như thể chúng cùng "kiểu". Dữ liệu đặc thù của đối tượng lưu trên lớp Instance, dữ liệu và hành vi chung cho một "kiểu" lưu trên lớp InstanceType.



Ví dụ minh họa

Mỗi quái vật (Monster) trong trò chơi đều có trị sức khỏe hiện tại (health). Trị này ban đầu được khởi tạo, và mỗi khi quái vật bị thương, nó giảm đi. Quái vật cũng có một chuỗi tấn công (attack). Khi quái vật tấn công, chuỗi này sẽ được hiển thị.

Bộ phận thiết kế cho chúng ta biết quái vật có rất nhiều chủng loại (breed) khác nhau, như Dragon. Mỗi chủng loại mô tả một loại quái vật tồn tại trong trò chơi, có thể có nhiều quái vật cùng một chủng loại tồn tại cùng lúc. Chủng loại xác định sức khỏe (health) ban đầu của quái vật và xác định chuỗi tấn công, tất cả quái vật cùng một chủng loại tấn công theo cùng một cách.

Theo thiết kế OOP điển hình, đầu tiên ta viết lớp cơ sở Monster:

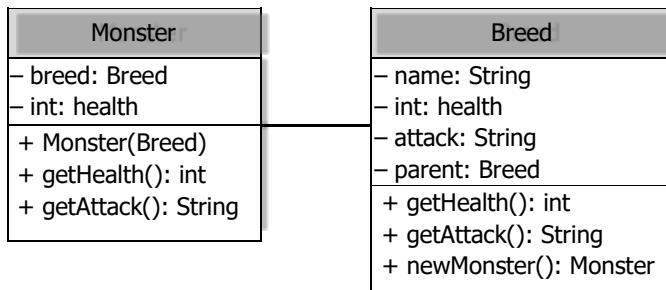
```
abstract class Monster {
 private int health;
 protected Monster(int startingHealth) {
 this.health = startingHealth;
 }
 abstract public String getAttack();
}
```

Sau đó, dẫn xuất ra mỗi lớp con cho một chủng loại mới:

```
class Dragon extends Monster {
 public Dragon() {
 super(100);
 }
 @Override String getAttack() { return "spray fire on you!"; }
}
```

Vấn đề là có đến hàng trăm chủng loại được thiết kế, ngoài ra bộ phận thiết kế muốn điều chỉnh thiết kế trong tương lai, nghĩa là phải thay đổi các lớp con đã code.

Giải pháp



Thay vì dẫn xuất từ Monster ra các lớp con cho mỗi chủng loại, ta có hai lớp:

- Lớp Monster, mỗi quái vật trong trò chơi là một thể hiện của lớp Monster.
- Lớp Breed, chứa thông tin dùng chung giữa các quái vật có cùng chủng loại (trị health ban đầu và chuỗi attack). Lớp Breed định nghĩa "kiểu" (type) của một đối tượng lớp Monster. Mỗi thể hiện của lớp Breed đại diện một kiểu quái vật cụ thể. Vì vậy mẫu thiết kế gọi là Type Object.

Chúng ta có thể tạo ra hàng trăm chủng loại khác nhau bằng cách tạo ra hàng trăm đối tượng "kiểu" thuộc lớp Breed với các giá trị khác nhau. Nếu bạn khởi tạo chúng từ dữ liệu đọc từ tập tin cấu hình, chúng ta có khả năng định nghĩa các chủng loại hoàn toàn mới và dễ dàng định nghĩa lại "kiểu" đã có.

Để liên kết Monster với chủng loại của chúng, mỗi đối tượng lớp Monster chứa một tham chiếu đến một đối tượng lớp Breed.

```
import java.util.stream.Stream;
// Instance class
Monster { int
 health; Breed
 breed;
```

```

 Monster(Breed breed) {
this.health = breed.health;
this.breed = breed;
}
int getHealth() {
 return breed.getHealth();
}
}

String getAttack() {
 return breed.name + " " + breed.getAttack();
}
}

// InstanceType
class Breed {
Breed parent;
String name;
int health;
String attack;

public Breed(Breed parent, String name, int health, String attack) {
this.parent = parent; this.name = name; this.health = health;
this.attack = attack;
}
int getHealth() { return (health != 0 || parent == null) ?
health // Override
 : parent.getHealth(); // Inherit
}
String getAttack() { return (attack != null || parent == null)
? attack // Override
 : parent.getAttack(); // Inherit
}

Monster newMonster() {
return new Monster(this);
}
}
public class Client { public static void
main(String[] args) {
System.out.println("--- Type Object ---");
 Breed troll = new Breed(null, "Troll", 25, "hits you!");
 Breed trollArcher = new Breed(troll, "Troll Archer", 0, "fires an arrow!");
 Breed trollWizard = new Breed(troll, "Troll Wizard", 0, "casts a spell on you!");
 Breed dragon = new Breed(null, "Dragon", 100, "spray fire on you!");
Monster[] monsters = {
 troll.newMonster(), trollArcher.newMonster(),
new Monster(trollWizard), new Monster(dragon) };
 Stream.of(monsters).map(Monster::getAttack).forEach(System.out::println);
}
}
}

```

## Extension Object

Additional interfaces are defined by extension objects

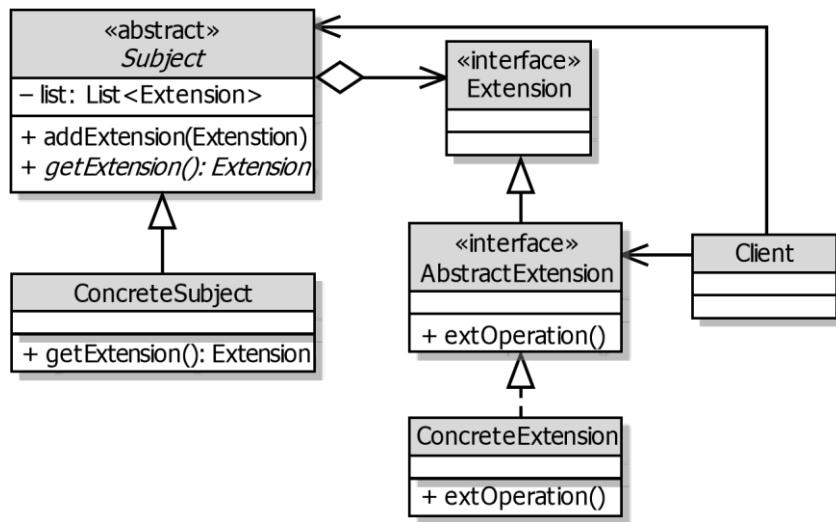
Giao diện của một đối tượng (Subject) có thể cần được mở rộng trong tương lai. Tuy nhiên đối tượng này có thể là một đối tượng phức hợp, có cấu trúc phân cấp phức tạp. Nếu bạn muốn tránh thay đổi trực tiếp các đối tượng trong cấu trúc phân cấp khi mở rộng thêm chức năng, bạn có thể cung cấp giao diện bổ sung bằng các đối tượng mở rộng (Extension Object). Mỗi đối tượng trong cấu trúc phân cấp:

- Duy trì một danh sách các đối tượng mở rộng đặc thù cho nó.
- Cung cấp phương thức cho phép lấy các đối tượng mở rộng theo tên.

Các phương thức bổ sung của đối tượng mở rộng có thể thao tác lên đối tượng phân cấp gốc.

Bạn nên sử dụng mẫu thiết kế Extension Object khi:

- Cần bổ sung giao diện mới hoặc không lường trước được vào giao diện hiện tại cho các lớp và bạn không muốn tác động đến những Client không cần giao diện mới này. Extension Object cho phép bạn nhóm các chức năng mở rộng liên quan với nhau bằng cách định nghĩa chúng trong một lớp riêng.
- Một lớp thể hiện cho một khái niệm trừu tượng chính đóng vai trò khác nhau với các Client khác nhau. Lớp cần mở rộng nhưng vẫn phải giữ khái niệm trừu tượng chính.
- Mở rộng hành vi mới cho một lớp mà không cần thừa kế nó.



Như vậy, ngoài mẫu thiết kế Visitor, một cách khác để thêm chức năng vào một cấu trúc phân cấp mà không thay đổi cấu trúc phân cấp là sử dụng mẫu thiết kế Extension Object.

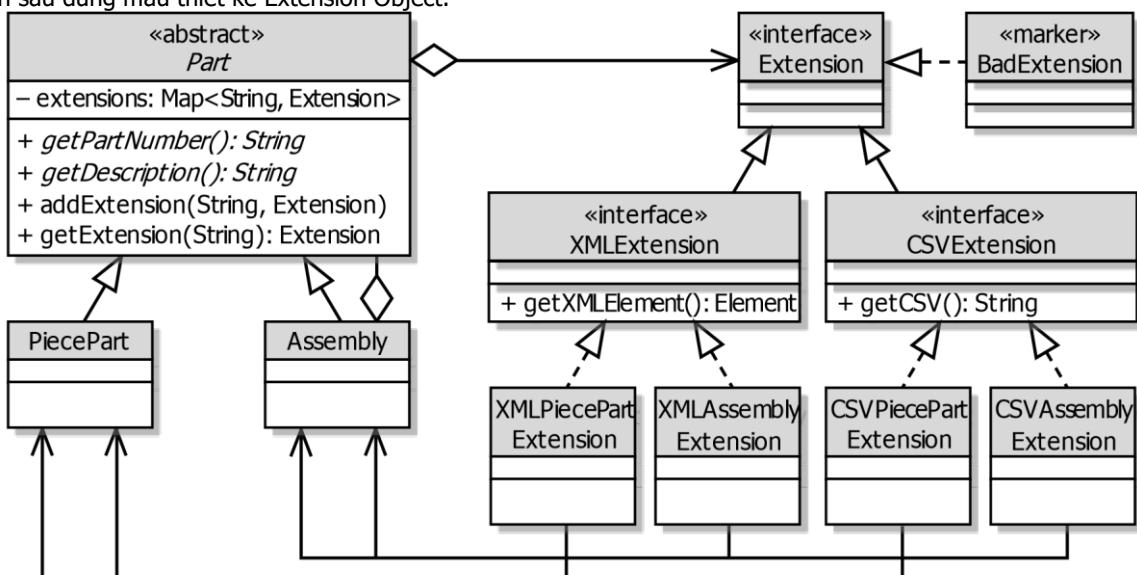
Ví dụ minh họa

Giả sử hóa đơn (bill) là một đối tượng phức hợp gồm nhiều Part, mỗi Part có thể là một PiecePart (partNo, cost, description) hoặc một Assembly (partNo, description, parts) với parts danh sách các Part hợp thành Assembly đó.

Chúng ta cần mở rộng khả năng các đối tượng (PiecePart, Assembly) trong cấu trúc phân cấp này để tạo ra biểu diễn XML (getXMLElement) hoặc CSV (getCSV) cho từng loại đối tượng. Nếu đặt các phương thức mở rộng này vào cấu trúc phân cấp, sẽ làm thay đổi cấu trúc phân cấp và có thể vi phạm nguyên lý SRP.

Nếu dùng mẫu thiết kế Visitor để xử lý các đối tượng trong cấu trúc phân cấp, tất cả code tạo XML (hoặc CSV) cho các đối tượng khác nhau trong cấu trúc phân cấp sẽ nằm trong cùng một đối tượng Visitor.

Phương án sau dùng mẫu thiết kế Extension Object.



Ví dụ có sử dụng thư viện JDOM ([www.jdom.org/](http://www.jdom.org/))

```

import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import org.jdom2.Element;
abstract class Part {
 HashMap<String, Extension> extensions = new HashMap<>();
 public abstract String getPartNumber(); public abstract
 String getDescription();

 public void addExtension(String type, Extension extension) {
 extensions.put(type, extension);
 }

 Extension getExtension(String type) {
 Extension pe = extensions.get(type); return
 pe == null ? new BadExtension() : pe;
 }
}

```

```

interface Extension { }

class BadExtension implements Extension { }

class PiecePart extends Part {
 String partNo;
 String description;
 double cost;

 PiecePart(String partNo, String description, double cost) {
 this.partNo = partNo; this.description = description;
 this.cost = cost; super.addExtension("CSV", new
CSVPiecePartExtension(this)); super.addExtension("XML", new
XMPiecePartExtension(this));
 }

 @Override public String getPartNumber() {
 return partNo;
 }

 @Override public String getDescription() {
 return description;
 }

 @Override public Extension getExtension(String type) {
 if (type.equals("XML")) return new XMPiecePartExtension(this);
 else if (type.equals("CSV")) return new CSVPiecePartExtension(this);
 return new BadExtension();
 }
}

class Assembly extends Part {
 List<Part> parts = new LinkedList<>();
 String partNo;
 String description;

 public Assembly(String partNo, String description) {
 this.partNo = partNo; this.description =
description;
 super.addExtension("CSV", new CSVAssemblyExtension(this));
 super.addExtension("XML", new XMLAssemblyExtension(this));
 }

 @Override public String getPartNumber() {
 return partNo;
 }

 @Override public String getDescription() {
 return description;
 }

 void add(Part part) {
 parts.add(part);
 }

 List<Part> getParts() {
 return parts;
 }
}

interface XMLExtension extends Extension {
 Element getXMLElement();
}

class XMPiecePartExtension implements XMLExtension {
 PiecePart piecePart = null;

 XMPiecePartExtension(PiecePart part) {
 piecePart = part;
 }

 @Override public Element getXMLElement() {
 return new Element("PiecePart")
 }
}

```

```

 .addContent(new Element("PartNumber").setText(piecePart.partNo))
 .addContent(new Element("Description").setText(piecePart.description))
 .addContent(new Element("Cost").setText(Double.toString(piecePart.cost)));
 }
}
class XMLAssemblyExtension implements XMLExtension {
 Assembly assembly = null;
 XMLAssemblyExtension(Assembly assembly) {
this.assembly = assembly;
 }

@Override public Element getXMLElement() {
 Element parts = new Element("Parts");
 Element e = new Element("Assembly")
 .addContent(new Element("PartNumber").setText(assembly.partNo))
 .addContent(new Element("Description").setText(assembly.description))
 .addContent(parts);
assembly.getParts().stream()
 .map(p -> (XMLExtension)p.getExtension("XML"))
.forEach(x -> parts.addContent(x.getXMLElement())); return
e;
}
}
interface CSVExtension extends Extension {
 String getCSV();
}
class CSVPiecePartExtension implements CSVExtension {
 PiecePart piecePart = null;
 CSVPiecePartExtension(PiecePart part) {
piecePart = part;
 }

@Override public String getCSV() {
return new StringBuffer("PiecePart, ")
 .append(piecePart.getPartNumber()).append(", ")
 .append(piecePart.getDescription()).append(", ")
 .append(piecePart.cost)
 .toString();
}
}
class CSVAssemblyExtension implements CSVExtension {
 Assembly assembly;

 CSVAssemblyExtension(Assembly assembly) {
this.assembly = assembly;
 }

@Override public String getCSV() {
 StringBuffer b = new StringBuffer("Assembly, ")
 .append(assembly.getPartNumber()).append(", ")
 .append(assembly.getDescription());
assembly.getParts().stream()
 .map(p -> (CSVExtension)p.getExtension("CSV"))
 .forEach(x -> b.append(", {").append(x.getCSV()).append("}"));
return b.toString();
}
}
public class Client { public static void
main(String[] args) {
System.out.println("--- Type Object ---");
 PiecePart p1 = new PiecePart("997624", "MyPart", 3.20);
 PiecePart p2 = new PiecePart("7734", "Hell", 666);
 Assembly a = new Assembly("5879", "MyAssembly");
a.add(p1);
 a.add(p2);

Extension e = p1.getExtension("XML");
XMLExtension xe = (XMLExtension)e;

```

```
Element xml = xe.getXMLElement();
System.out.println(xml.getChild("PartNumber").getTextTrim()); // 997624
Extension e1 = a.getExtension("CSV");
CSVExtension ce = (CSVExtension)e1;
String csv = ce.getCSV();
System.out.println(csv);
// Assembly, 5879, MyAssembly, {PiecePart, 997624, MyPart, 3.2}, {PiecePart, 7734, Hell, 666.0}
}
}
```

## Dependency Injection

Instantiation from the external entity

Một lớp thường có quan hệ với nhiều lớp phụ thuộc khác, các lớp này gọi là các dependency của lớp sử dụng. Khi sử dụng các dependency, bạn có thể:

- Khởi tạo trực tiếp

Tạo thể hiện của dependency trong lớp sử dụng bằng cách dùng toán tử "new". Như vậy lớp sử dụng kết nối chặt với dependency, vì bạn không thể thay đổi giá trị tham số khởi tạo dependency, cũng không sử dụng một cài đặt khác của dependency, mà không phải thay đổi code của lớp sử dụng.

Bạn có thể giảm sự phụ thuộc bằng cách "tiêm" (truyền) các tham số khởi tạo dependency vào constructor của lớp sử dụng. Bây giờ, có thể thay đổi các tham số khởi tạo dependency mà không phải thay đổi code của lớp sử dụng. Tuy nhiên lớp sử dụng vẫn phụ thuộc vào giao diện của dependency, tức phụ thuộc các tham số khởi tạo cho dependency. Ý tưởng là "tiêm" dependency đã khởi tạo vào lớp sử dụng.

- Dependency injection

Dùng một đối tượng bên ngoài tạo thể hiện độc lập của dependency rồi "tiêm" (injection) dependency vào lớp sử dụng. Dependency injection là một kiểu cấu hình đối tượng trong đó các đối tượng phụ thuộc (dependency) được thiết lập bởi một thực thể bên ngoài đối tượng sử dụng. Đối tượng sử dụng không khởi tạo cho các đối tượng phụ thuộc bằng toán tử "new", mà "tiêm" chúng vào để sử dụng.

Mẫu thiết kế để thực hiện việc "tiêm" dependency vào lớp sử dụng gọi là Dependency Injection. Mẫu thiết kế này tạo nên kết nối lỏng (loose coupling) giữa lớp sử dụng và các lớp phụ thuộc (dependency).

Lưu ý rằng "đối tượng bên ngoài" do cung cấp dependency khởi tạo cho lớp sử dụng nên đến lượt nó phụ thuộc vào dependency.

Nói cách khác, sự phụ thuộc chuyển từ đối tượng sử dụng dependency sang đối tượng cung cấp dependency. Điều này tệ hơn do lớp cung cấp phụ thuộc vào dependency mà nó không sử dụng trực tiếp thông tin. Sự phụ thuộc dây chuyền như vậy có thể tiếp tục theo con đường hướng lên, từ các lớp sử dụng dependency ban đầu đến giao diện người dùng (nếu có). Vì vậy, "đối tượng bên ngoài" thường được xây dựng như một container chứa các dependency, gọi là DI Container. Nó chịu trách nhiệm đăng ký, khởi tạo, quản lý vòng đời, cấu hình cho các dependency, và "tiêm" chúng đến lớp sử dụng khi có yêu cầu.

Các DI Container hiện nay thường sử dụng một trong bốn cơ chế cấu hình sau để "tiêm" các dependency:

- Dùng code Java. Ví dụ: gọi phương thức, các factory, các provider
- Dùng annotation. Ví dụ: @Inject, @Wired
- Dùng XML. Ví dụ: beans.xml
- Ngôn ngữ đặc thù miền (DSL - Domain Specific Language)

Các phương pháp "tiêm" dependency vào lớp sử dụng:

- Constructor injection: dependency (bắt buộc) sẽ được truyền vào lớp sử dụng thông qua constructor của lớp đó.
- Setter injection: dependency (tùy chọn) sẽ được truyền vào lớp sử dụng thông qua các phương thức setter của lớp đó.
- Public fields injection: dependency sẽ được "tiêm" vào filed tương ứng của lớp sử dụng.

Lợi ích của Dependency injection

- Giảm sự phụ thuộc vào dependency. Lớp sử dụng dễ thay đổi, thêm vào và loại bỏ các dependency.
- Tái sử dụng code nhiều hơn. Việc giảm phụ thuộc vào các dependency thường giúp dễ dàng sử dụng lại code của lớp sử dụng trong một ngữ cảnh khác.
- Tăng khả năng kiểm thử. Do có thể "tiêm" các dependency nên cũng có thể tiêm các cài đặt giả lập (mock) của các dependency này. Chúng được sử dụng để thử nghiệm thay cho cài đặt thực tế của dependency.
- Code dễ đọc hơn. Lớp sử dụng làm việc với các dependency thông qua giao diện, điều này giúp dễ dàng hơn để xem dependency mang đến những gì.

Dependency injection có hiệu quả trong các tình huống sau:

- Cần "tiêm" dữ liệu cấu hình vào một hoặc nhiều thành phần sử dụng.
- Cần "tiêm" cùng một dependency vào nhiều thành phần sử dụng.
- Cần "tiêm" các cài đặt khác nhau của cùng một dependency.
- Cần "tiêm" cùng một cài đặt của dependency với các cấu hình khác nhau. Dependency injection không cần thiết trong các tình huống sau:
  - Không bao giờ cần triển khai khác.
  - Không bao giờ cần cấu hình khác.

### 1) DI Container đơn giản

Ví dụ minh họa

Chương trình dùng interface CustomerDao làm giao diện truy cập cơ sở dữ liệu. Chương trình cần kết nối lỏng để có thể thay đổi loại cơ sở dữ liệu khi cần:

- Áp dụng mẫu thiết kế DAO (Data Access Object) để tạo một lớp (layer) truy cập cơ sở dữ liệu trừu tượng.
- Đăng ký các loại CustomerDao với DI Container: DBCustomerDao hoặc InMemCustomerDao.
- Thể hiện của CustomerDao không được khởi tạo bằng toán tử "new". DIContainer sẽ khởi tạo và "tiêm" thể hiện đã đăng ký của CustomerDao vào lớp CustomerService thông qua constructor.

Khi gọi phương thức resolve(), DIContainer khởi tạo các thể hiện của các lớp đăng ký một cách thủ công bằng cách dùng

Java Reflection để xác định và gọi constructor của các lớp đăng ký. Bằng cách đó, thể hiện của CustomerDao (là InMemCustomerDao) được khởi tạo rồi "tiêm" vào CustomerService thông qua constructor.

Chương trình minh họa

```
import java.lang.reflect.Constructor; import
java.util.ArrayList;
import java.util.Arrays; import
java.util.HashMap; import
java.util.List; import
java.util.Map; import
java.util.Optional; import
java.util.stream.Stream;
class Customer
{ int id;
String firstName, lastName;

Customer(int id, String firstName, String lastName) {
this.id = id; this.firstName = firstName;
this.lastName = lastName;
}

@Override public String toString() {
return String.format("[%d] %s %s", id, firstName, lastName);
}
}
interface CustomerDao {
Stream<Customer> getAll() throws Exception;
Optional<Customer> getById(int id) throws Exception;
boolean add(Customer customer) throws Exception;
boolean update(Customer customer) throws Exception;
boolean delete(Customer customer) throws Exception;
}
class InMemCustomerDao implements CustomerDao {
Map<Integer, Customer> customers = new HashMap<>();

@Override public Stream<Customer> getAll() {
return customers.values().stream();
}

@Override public Optional<Customer> getById(int id) {
return Optional.ofNullable(customers.get(id));
}

@Override public boolean add(Customer customer) {
if (getById(customer.id).isPresent()) return false;
customers.put(customer.id, customer); return true;
}

@Override public boolean update(Customer customer) {
return customers.replace(customer.id, customer) != null;
}

@Override public boolean delete(final Customer customer) {
return customers.remove(customer.id) != null;
}
}

class DIContainer {
Map<Class<?>, Class<?>> types = new HashMap<>();
void register(Class<?> forInterface, Class<?> forUse) throws Exception {
if (types.containsKey(forInterface)) throw new Exception("Type already registered!");
types.put(forInterface, forUse);
}

Object resolve(Class<?> forInterface) throws Exception {
return getImpl(forInterface);
}

Object getImpl(Class<?> forInterface) throws Exception {
```

```

 if (!types.containsKey(forInterface)) throw new Exception("Type does not exist!");
 Class impl = types.get(forInterface);
 Constructor constructor = impl.getDeclaredConstructors()[0];
 List list = new ArrayList();
 for (Class t : constructor.getParameterTypes()) list.add(getImpl(t));
 return constructor.newInstance(list.toArray());
 }
}

class CustomerService {
 CustomerDao dao;

 public CustomerService(CustomerDao dao) {
 this.dao = dao;
 }

 List<Customer> customers = Arrays.asList(
 new Customer(1, "Bill", "Clinton"),
 new Customer(2, "Barack", "Obama"), new
 Customer(3, "Donald", "Trump"));
 void init() throws Exception {
 System.out.println("dao::add");
 for (Customer customer : customers) dao.add(customer);
 }
 void findAll() throws Exception {
 System.out.println("dao::getAll"); try
 (Stream<Customer> stream = dao.getAll()) {
 stream.forEach(System.out::println);
 }
 }
}
public class Client { public static void main(String[]
args) throws Exception {
 System.out.println("--- Dependency Injection ---");
 DIContainer container = new DIContainer();
 container.register(CustomerDao.class, InMemCustomerDao.class);
 container.register(CustomerService.class, CustomerService.class);
 CustomerService service = (CustomerService) container.resolve(CustomerService.class);
 service.init(); service.findAll();
}
}

```

## 2) DI Container của framework

Trong Spring và .NET framework, DI Container còn được gọi là IoC Container (IoC - Inverse of Control). Ví dụ minh họa dưới đây dùng DI Container của Google Guice (<https://github.com/google/guice>).

### Chương trình minh họa

Dùng Maven hoặc tải về và đặt các gói JAR cần thiết vào CLASSPATH:

Google Guice <https://mvnrepository.com/artifact/com.google.inject/guice/4.2.0>

AOP Alliance <https://mvnrepository.com/artifact/aopalliance/aopalliance/1.0>

Guava <https://mvnrepository.com/artifact/com.google.common/guava/23.0>

Javax Inject <https://mvnrepository.com/artifact/javax.inject/javax.inject/1>

```

Dependency được "tiêm" vào bằng annotation @Inject
import com.google.inject.AbstractModule;
import com.google.inject.Guice;
import com.google.inject.Inject;
import com.google.inject.Injector;
import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Optional;
import java.util.stream.Stream;

```

```

class Customer
{
 int id;
 String firstName, lastName;
```

```

 Customer(int id, String firstName, String lastName) {
 this.id = id;
 this.firstName = firstName;
 this.lastName = lastName;
 }

 @Override public String toString() {
 return String.format("[%d] %s %s", id, firstName, lastName);
 }
}
```

```

}

interface CustomerDao {
 Stream<Customer> getAll() throws Exception;
 Optional<Customer> getById(int id) throws Exception;
 boolean add(Customer customer) throws Exception;
 boolean update(Customer customer) throws Exception;
 boolean delete(Customer customer) throws Exception;
}

class InMemCustomerDao implements CustomerDao {
 Map<Integer, Customer> customers = new HashMap<>();

 @Override public Stream<Customer> getAll() {
 return customers.values().stream();
 }

 @Override public Optional<Customer> getById(int id) {
 return Optional.ofNullable(customers.get(id));
 }

 @Override public boolean add(Customer customer) {
 if (getById(customer.id).isPresent()) return false;
 customers.put(customer.id, customer); return true;
 }

 @Override public boolean update(Customer customer) {
 return customers.replace(customer.id, customer) != null;
 }

 @Override public boolean delete(final Customer customer) {
 return customers.remove(customer.id) != null;
 }
}

// Binding Module, có vai trò như DI Container
class DaoModule extends AbstractModule {
 @Override
 protected void configure() {
 bind(CustomerDao.class).to(InMemCustomerDao.class);
 }
}
class CustomerService {
 @Inject
 CustomerDao dao;

 List<Customer> customers = Arrays.asList(
 new Customer(1, "Bill", "Clinton"),
 new Customer(2, "Barack", "Obama"),
 new Customer(3, "Donald", "Trump"));
 void init() throws Exception {
 System.out.println("dao::add");
 for (Customer customer : customers) dao.add(customer);
 }
 void findAll() throws Exception {
 System.out.println("dao::getAll");
 try {
 (Stream<Customer> stream = dao.getAll()) {
 stream.forEach(System.out::println);
 }
 }
 }
}

public class Client {
 public static void main(String[] args) throws Exception {
 System.out.println("----");
 System.out.println("Dependency Injection ---");
 Injector injector = Guice.createInjector(new DaoModule());
 CustomerService service = injector.getInstance(CustomerService.class);
 service.init(); service.findAll();
 }
}

```



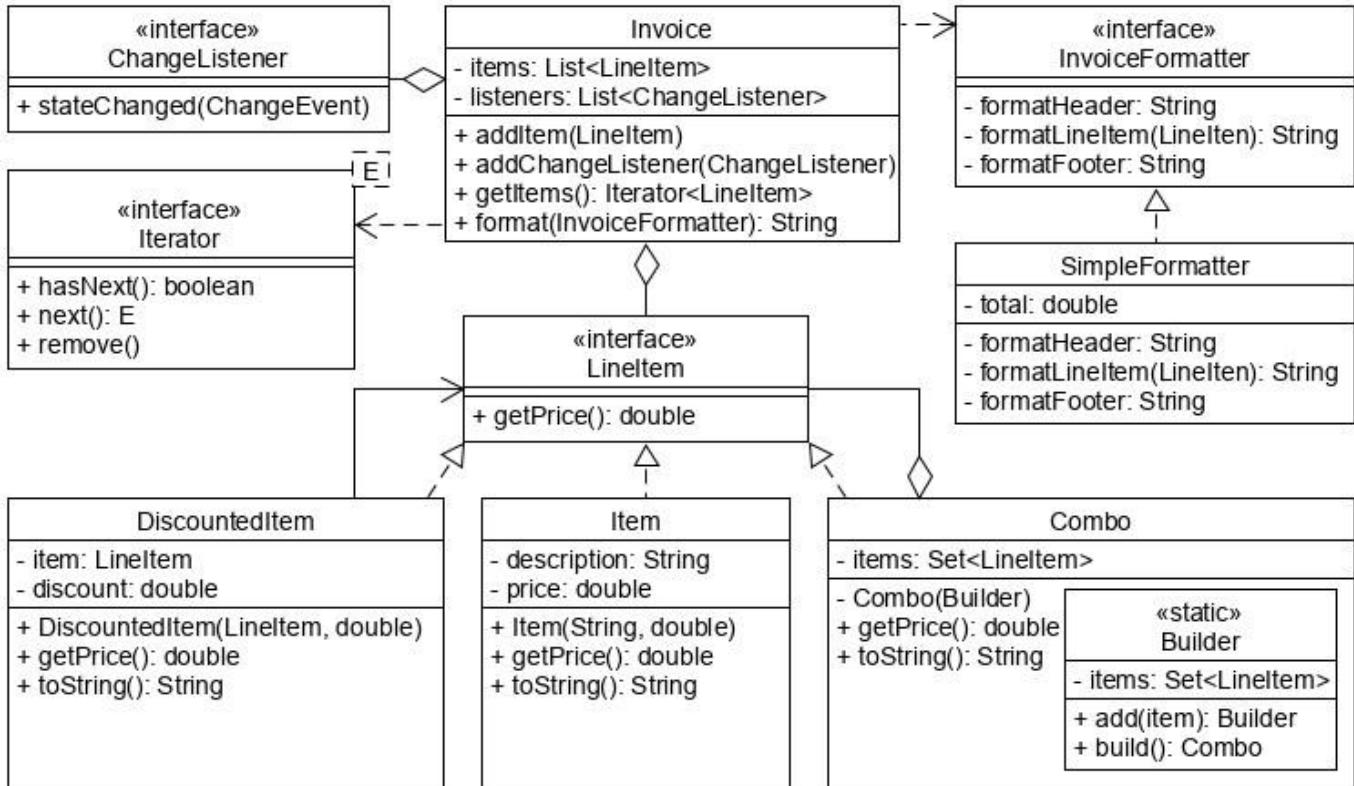
## Case study

### Case study – Invoice of POS

#### 1. Đặc tả

Module Invoice là một module trong ứng dụng POS hiển thị danh sách các mặt hàng (LineItem). Các mặt hàng được bán riêng (Item) hoặc bán chung theo gói (Combo) gồm vài mặt hàng. Các Item, Combo, thậm chí từng mặt hàng trong Combo có thể được giảm giá theo tỷ lệ phần trăm chỉ định (discount) khác nhau. Những mặt hàng được chọn sẽ cập nhật và hiển thị ngay trong một JTextArea thể hiện một Invoice. Module cho phép tùy biến định dạng văn bản hiển thị của Invoice.

#### 2. Sơ đồ lớp



#### 3. Các mẫu thiết kế

Bạn hãy so sánh các lớp (giao diện) trong sơ đồ lớp bên trên với sơ đồ lớp cài đặt cho từng mẫu thiết kế tương ứng để xác định được các lớp (giao diện) tham gia trong mỗi mẫu thiết kế và vai trò của chúng.

#### Composite

Một dòng trong đơn hàng (LineItem) có thể là mặt hàng (Item) được bán lẻ hoặc được bán theo gói (combo) tức một bộ nhiều mặt hàng. Mẫu thiết kế Composite được áp dụng để làm việc với đối tượng phức hợp này. - LineItem (Component): giao diện khai báo tác vụ chung `getPrice()`.

- Item (Leaf): đối tượng đơn.
- Combo (Composite): đối tượng phức hợp, phương thức `add()` để thêm đối tượng đơn vào nó, và phương thức `getPrice()` được viết lại để thực hiện tác vụ chung trên đối tượng phức hợp.

#### Builder

Combo là một danh sách nhiều mặt hàng phân biệt, mỗi mặt hàng có chế độ giảm giá khác nhau. Để việc tạo Combo phức hợp trở nên rõ ràng hơn, mẫu thiết kế Builder được áp dụng ở đây.

- Builder (ConcreteBuilder): phương thức `add()` tạo từng phần của đối tượng phức hợp và phương thức `build()` trả về đối tượng phức hợp.
- Combo (ConcreteProduct): đối tượng phức hợp, kết quả quá trình "xây dựng" đối tượng của Builder.

#### Decorator

Một mặt hàng giảm giá (DiscountedItem) là một mặt hàng (LineItem) có thêm thuộc tính `discount` (phần trăm chiết xuất) và phương thức `getPrice()` được viết lại để tính giá đã giảm. Thay vì dùng thừa kế để mô tả mặt hàng giảm giá, mẫu thiết kế Decorator được áp dụng để giải quyết vấn đề này.

- LineItem (Component): giao diện chung cho các đối tượng (có "trang trí" hoặc chưa "trang trí").
- Item (ConcreteComponent): đối tượng đơn giản chưa được "trang trí".
- DiscountedItem (ConcreteDecorator): đối tượng được "trang trí" thêm, với constructor nhận `LineItem` làm tham số và phương thức "trang trí" thêm là viết lại phương thức `getPrice()`.

## Observer

Trong ứng dụng, khi một mặt hàng được thêm vào Invoice, văn bản trong JTextArea của ứng sẽ được cập nhật, nhiệm vụ này có thể trao cho đối tượng xử lý sự kiện của nút [Add] thực thi. Nhưng ta có thể dùng mẫu thiết kế Observer để thực hiện điều này, tách biệt việc thêm mặt hàng với việc cập nhật hiển thị Invoice.

- ChangeListener (Subject): giao diện cho đối tượng dữ liệu, phương thức addChangeListener() dùng đăng ký các đối tượng theo dõi dữ liệu, một phần phương thức addItem() sẽ báo cho các đối tượng đăng ký với nó là dữ liệu đã thay đổi.
- Invoice (ConcreteSubject): lưu danh sách các đối tượng đăng ký theo dõi (listeners). Các đối tượng theo dõi này (Observer) là các ChangeListener của Swing, phương thức stateChanged() của nó sẽ cập nhật giao diện với dữ liệu mới.

## Iterator

Invoice được xem như một đối tượng chứa các đối tượng thành phần LineItem. Như vậy, có thể áp dụng mẫu thiết kế Iterator để duyệt các đối tượng thành phần này, chẳng hạn để lưu vào một cấu trúc dữ liệu khác hoặc lưu vào cơ sở dữ liệu.

- Iterator<E> (Iterator): giao diện cho đối tượng truy cập các đối tượng thành phần, phương thức remove() là tùy chọn.
- Iterator<LineItem> (ConcreteIterator): đối tượng vô danh cài đặt giao diện này là Iterator dùng truy cập các đối tượng thành phần của Invoice.
- Invoice (ConcreteAggregate): đối tượng chứa, phương thức getItems() trả về đối tượng Iterator dùng duyệt các đối tượng thành phần của nó.

## Strategy

Có nhiều "chiến lược" định dạng Invoice, mỗi chiến lược được đóng gói theo mẫu thiết kế Strategy để tùy biến linh hoạt "chiến lược" định dạng cho Invoice.

- InvoiceFormatter (Strategy): giao diện chung cho các "chiến lược" định dạng. Lưu ý là có thể áp dụng mẫu thiết kế Template Method tại đây.
- SimpleFormatter (ConcreteStrategy): một "chiến lược" định dạng cụ thể.
- Invoice (Context): phương thức format() thực thi "chiến lược" cụ thể được chọn truyền đến nó như tham số.

## 4. Chương trình

```
import java.awt.BorderLayout; import
java.awt.event.ActionEvent; import
java.awt.event.ActionListener; import
java.util.*;
import java.util.stream.Collectors;
import javax.swing.*; import
javax.swing.event.ChangeEvent; import
javax.swing.event.ChangeListener;
interface LineItem
{
 double
getPrice();
}
class Item implements LineItem {
 String description;
 double price;

 Item(String description, double price) {
 this.description = description;
 this.price = price;
 }

 @Override public double getPrice() {
 return price;
}

 @Override public String toString() {
 return description;
}
}
class Combo implements LineItem {
 Set<LineItem> items;

 private Combo(Builder builder) {
 this.items = builder.items;
 }

 @Override public double getPrice() {
 return items.stream().mapToDouble(LineItem::getPrice).sum();
 }

 @Override public String toString() {
```

```

 return "Combo: " + items.stream().map(LineItem::toString).collect(Collectors.joining(", "));
 }

 static class Builder {
 private final Set<LineItem> items = new HashSet<>();

 Builder add(LineItem item) {
 items.add(item); return
 this;
 }

 Combo build() {
 return new Combo(this);
 }
 }
}

class DiscountedItem implements LineItem {
 LineItem item; double discount;

 DiscountedItem(LineItem item, double discount) {
 this.item = item; this.discount = discount;
 }

 @Override public double getPrice() { return
 item.getPrice() * (1 - discount / 100);
 }

 @Override public String toString() {
 return String.format("%s (-%.1f%)", item, discount);
 }
}

interface InvoiceFormatter {
 String formatHeader();
 String formatLineItem(LineItem item);
 String formatFooter();
}

class SimpleFormatter implements InvoiceFormatter {
 double total;

 @Override public String formatHeader() {
 total = 0;
 return "--- I N V O I C E ---\n\n";
 }

 @Override public String formatLineItem(LineItem item) {
 total += item.getPrice();
 return String.format("%s: $%.2f\n", item, item.getPrice());
 }

 @Override public String formatFooter() { return
 String.format("\nTotal due: $%.2f\n", total);
 }
}

interface Iterator<E>
{
 boolean hasNext();
 E next(); void
 remove();
}

class Invoice {
 List<LineItem> items = new ArrayList<>();
 List<ChangeListener> listeners = new ArrayList<>();
 void addItem(LineItem
item) { items.add(item);
 ChangeEvent event = new ChangeEvent(this);
 listeners.forEach(l -> l.stateChanged(event)); }

 void addChangeListener(ChangeListener listener) {
 listeners.add(listener);
 }
}

```

```

 Iterator<LineItem> getItems() {
 return new Iterator<LineItem>() {
 int current = 0;
 @Override public boolean hasNext() { return current < items.size(); }
 @Override public LineItem next() { return items.get(current++); }
 @Override public void remove() { throw new UnsupportedOperationException(); }
 };
 }

 String format(InvoiceFormatter formatter) {
 String s = formatter.formatHeader();
 Iterator<LineItem> iterator = getItems();
 while (iterator.hasNext()) {
 LineItem item = iterator.next();
 s += formatter.formatLineItem(item);
 }
 return s + formatter.formatFooter();
 }
}

public class Client {
 public static void main(String[] args) {
 Invoice invoice = new Invoice();
 InvoiceFormatter formatter = new SimpleFormatter();

 JTextArea textArea = new JTextArea(20, 40);
 invoice.addChangeListener(new ChangeListener() {
 @Override public void stateChanged(ChangeEvent event) {
 textArea.setText(invoice.format(formatter));
 }
 });

 Item chips = new Item("French Fries", 19.95);
 Item coca = new Item("Soft Drink", 9.95);
 JComboBox listView = new JComboBox();
 listView.addItem(chips); listView.addItem(coca);
 listView.addItem(new Combo.Builder()
 .add(new DiscountedItem(chips, 5))
 .add(new DiscountedItem(coca, 10))
 .build());
 }

 JButton button = new JButton("Add");
 button.addActionListener(new ActionListener() {
 @Override public void actionPerformed(ActionEvent event) {
 LineItem item = (LineItem) listView.getSelectedItem();
 invoice.addItem(item);
 }
 });
 JPanel panel = new JPanel();
 panel.add(listView); panel.add(button);
 JFrame frame = new JFrame("Design Patterns Demo");
 frame.add(new JScrollPane(textArea), BorderLayout.CENTER);
 frame.add(panel, BorderLayout.SOUTH);
 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 frame.pack();
 frame.setLocationRelativeTo(null);
 frame.setVisible(true);
}
}

```

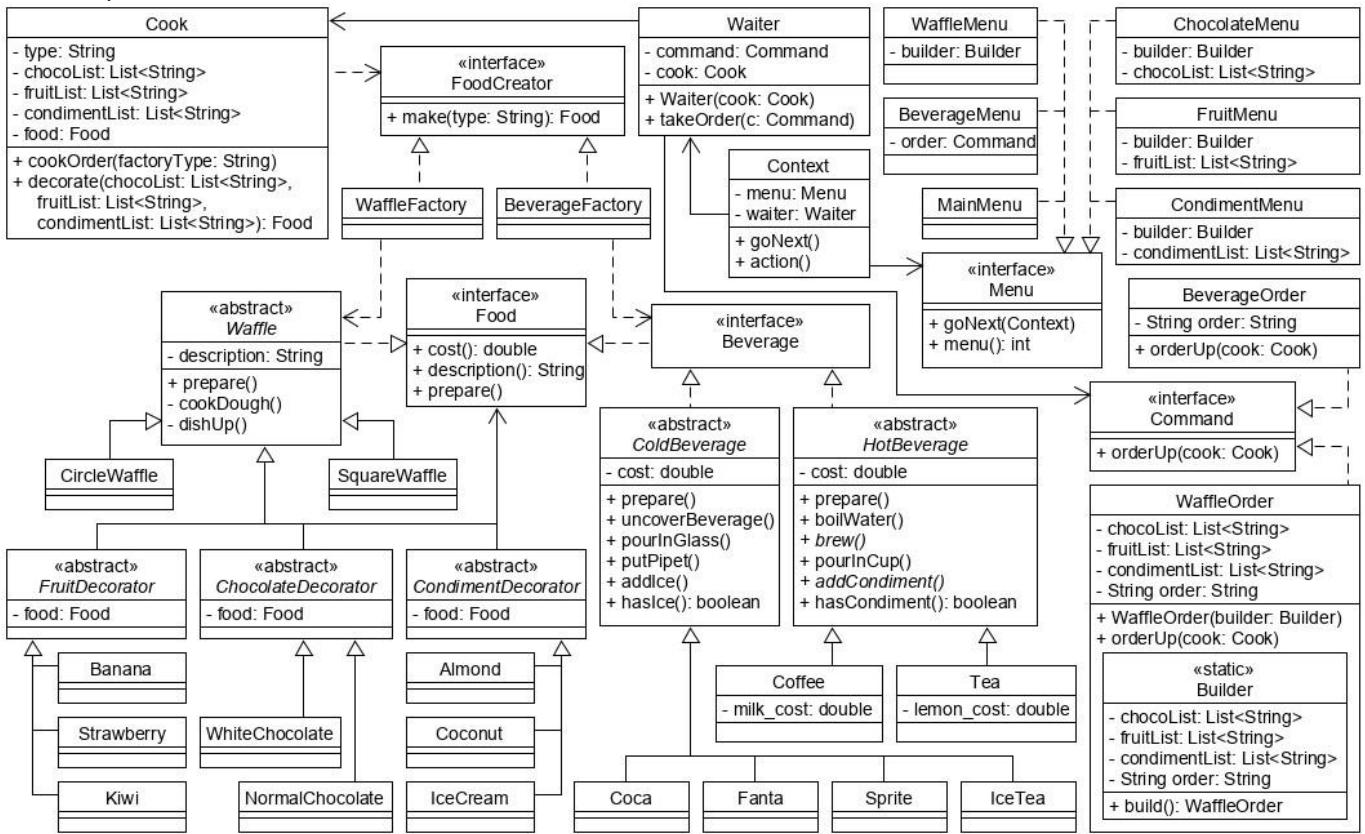
## Case study – Sweet Coffee

### 1. ĐẶC TÀ

Sweet Coffee là một quán cà phê bán bánh quế (Waffle) và đồ uống (Beverage). Khi khách hàng đến quán cà phê này, họ có thể thiết kế bánh quế họ muốn. Quy trình là chọn loại bánh quế (CircleWaffle hoặc SquareWaffle), sau đó chọn thêm chocolate (WhiteChocolate hoặc Normal Chocolate), trái cây (Banana, Strawberry hoặc Kiwi), phụ liệu (condiment: Almond, Coconut hoặc IceCream). Tùy thiết kế, chi phí sẽ được tính giá theo giá bánh và các phần thêm. Ngoài ra, khách hàng có thể đặt đồ uống nóng hoặc lạnh từ thực đơn. Phục vụ sẽ giao đơn hàng cho đầu bếp. Đầu bếp có trách nhiệm làm bánh và chuẩn bị đồ uống

theo quy trình. Chương trình hỗ trợ một menu nhiều cấp, mỗi màn hình menu có thể gọi nhiều lần, giúp khách hàng dễ dàng chọn món.

## 2. Sơ đồ lớp



## 3. Các mẫu thiết kế

### Factory Method

Cook dùng factory để tạo ra sản phẩm.

- Food (AbstractProduct) với các sản phẩm cụ thể (ConcreteProduct) là Waffle và Beverage.
- FoodCreator (Creator) dẫn xuất ra các factory cụ thể (ConcreteCreator): WaffleFactory và BeverageFactory. Phương thức factory là make().

### Decorator

Dùng "trang trí" thêm cho Waffle theo yêu cầu khách hàng, với ba vòng "trang trí" (ChocolateDecorator, FruitDecorator và CondimentDecorator).

- Food (Component): giao diện chung cho các Waffle (có "trang trí" hoặc chưa "trang trí").
- Waffle bao gồm CircleWaffle và SquareWaffle (ConcreteComponent): đối tượng đơn giản chưa được "trang trí".
- ChocolateDecorator, FruitDecorator và CondimentDecorator (Decorator): có thể gom thành một Decorator, nhưng chia thành nhiều abstract class để tách nhóm các lớp con.
- WhiteChocolate, NormalChocolate, Banana, Strawberry, Kiwi, Almond, Coconut, IceCream (ConcreteDecorator): là các lớp cho phép "trang trí" thêm.

### Template Method

Đồ uống nóng và lạnh có quy trình pha chế khác nhau, khách hàng cũng có thể yêu cầu thêm (sữa, chanh, đá lạnh). Mẫu thiết kế này giúp bảo đảm quy trình pha chế của hai loại đồ uống.

- HotBeverage và ColdBeverage (Abstract Class), lưu ý các phương thức của chúng:
- + Phương thức template prepare() mô tả các bước quy trình.
- + Các phương thức thành phần thực hiện các bước cố định trong quy trình.
- + Phương thức "hook" addCondiment() được định nghĩa hoặc không từ lớp con thực hiện các bước thêm chèn vào quy trình.

### Command

Đóng gói lệnh thực hiện các Order gửi từ Waiter đến Cook.

- Waiter (Invoker) đối tượng triệu gọi yêu cầu.
- Cook (Receiver) đối tượng nhận yêu cầu, phương thức cookOrder() của nó thực thi yêu cầu.
- Command (Command) đóng gói yêu cầu.
- BeverageOrder và WaffleOrder (Concrete Command) đóng gói yêu cầu cụ thể, yêu cầu được thực hiện khi gọi phương thức orderUp() của nó, Waiter gọi phương thức này nhưng Cook mới là đối tượng thực sự thực thi yêu cầu.

### State

Mỗi màn hình menu được xem như một trạng thái, phải có thể thực hiện hành động mong muốn tại mỗi màn hình menu và có thể chuyển sang màn hình menu khác tùy menu hiện hành.

- Context (Context): ngữ cảnh của trạng thái hiện tại (menu). Hai phương thức của nó:  
+ goNext(), ủy nhiệm cho đối tượng trạng thái hiện tại thực hiện, đối tượng này thiết lập lại ngữ cảnh chứa trạng thái kế tiếp rồi gọi goNext() của đối tượng ngữ cảnh để chuyển trạng thái.
- + action(), tại một trạng thái (ở đây là một menu), có thể chuyển sang menu khác hoặc chấm dứt chuyển menu và thực hiện yêu cầu. Đối tượng trạng thái cuối sẽ thực hiện yêu cầu bằng cách gọi action() của Context, thông tin cần cho việc thực hiện yêu cầu nhận từ đối tượng trạng thái.
- Menu (State): giao diện chung cho một trạng thái.
- MainMenu, WaffleMenu (tiếp là ChocolateMenu □ FruitMenu □ CondimentMenu) và BeverageMenu (ConcreteState): đóng gói các trạng thái cụ thể. Lưu ý là mỗi menu có thể lắp nhiều lần.

### **Builder**

Quy trình "lắp ráp" WaffleOrder phức tạp do khách hàng "thiết kế" bằng cách chọn qua nhiều cấp menu. Một Builder được truyền qua chuỗi các menu được chọn (chuỗi trạng thái), cấu hình dần WaffleOrder.

- WaffleOrder (Product): đối tượng có quá trình khởi tạo phức tạp.
- WaffleOrder.Builder (ConcreteBuilder): thu thập từng phần của yêu cầu để "cấu hình" WaffleOrder, phương thức build() của nó tạo ra WaffleOrder kết quả.

#### 4. Chương trình

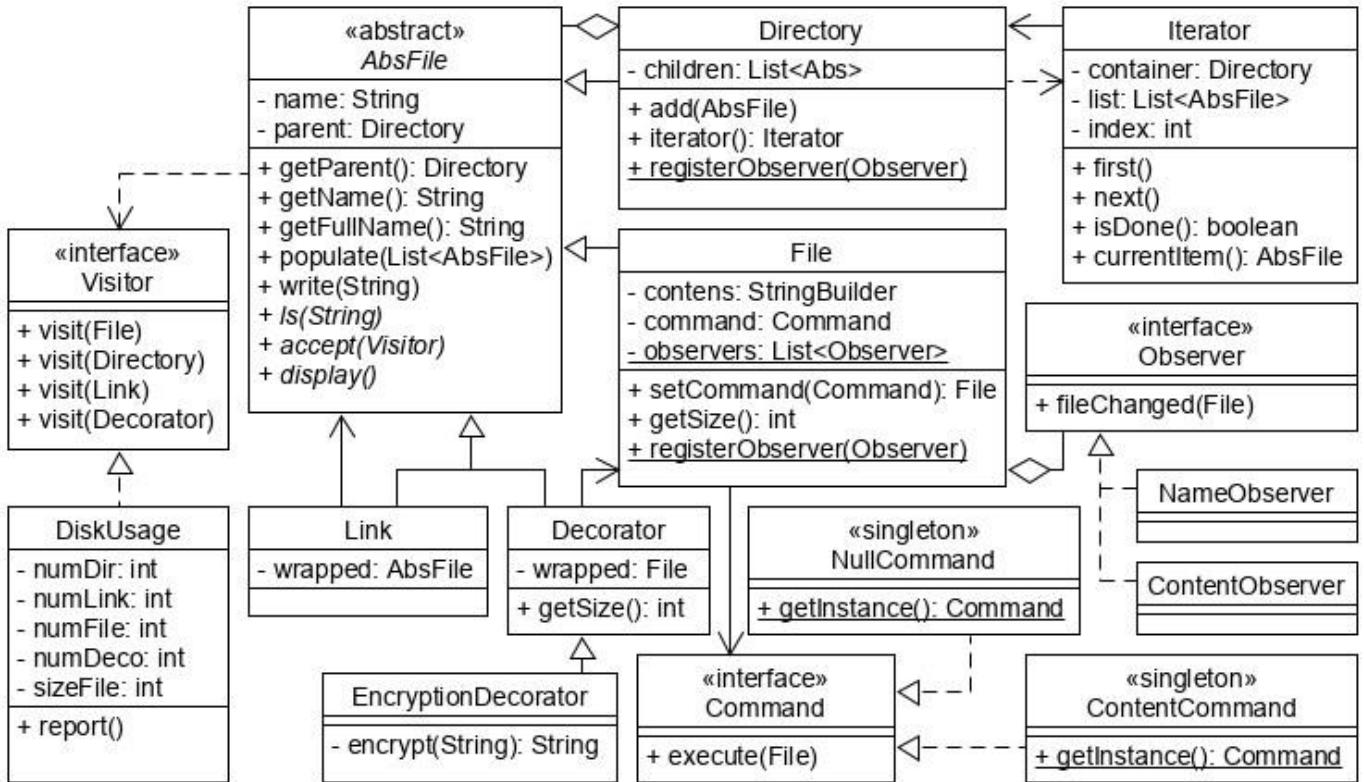
Xem mã nguồn kèm theo sách.

## Case study – File System

### 1. Đặc tả

Mô phỏng hoạt động của một hệ thống tập tin.

### 2. Sơ đồ lớp



### 3. Các mẫu thiết kế

#### Composite

Kiến trúc của hệ thống tập tin là kiến trúc phân cấp gồm File và Directory, Directory có thể rỗng hoặc chứa Directory con cùng các File khác. Cấu trúc này được mô phỏng bằng mẫu thiết kế Composite.

#### Proxy

Một Link trong hệ thống tập tin là một "đại diện" cho một File hoặc một Directory khác. Mẫu thiết kế Proxy được sử dụng để mô phỏng Link, "bao bọc" File hoặc Directory mà nó đại diện.

#### Chain of Responsibility

Chỉ ngược lại thư mục cha để tạo nên một "đường dẫn đầy đủ".

#### Iterator

Hệ thống tập tin được xem như một collection, mẫu thiết kế Iterator cung cấp một bộ lặp (iterator) dùng để duyệt các phần tử của kiến trúc phân cấp này.

#### Visitor

Một cách khác để duyệt kiến trúc phân cấp, thực hiện một thao tác tích lũy như đếm File, Directory, Link, tương tự lệnh du (disk usage) của Unix, mà không làm thay đổi kiến trúc phân cấp là sử dụng mẫu thiết kế Visitor.

#### Observer

Đăng ký các listener "one-to-many" để lắng nghe các sự kiện `write()` lên File như thay đổi tên hoặc thay đổi nội dung. Yêu cầu này được thực hiện bằng cách sử dụng mẫu thiết kế Observer.

#### Command

Đăng ký các đối tượng callback cho các sự kiện `write()` trên File, các đối tượng này được thực hiện bằng cách sử dụng mẫu thiết kế Command.

#### Decorator

Tăng cường chức năng `write()` trên File thành mã hóa File, là chức năng được "trang trí" thêm cho File. Yêu cầu này thích hợp cho việc áp dụng mẫu thiết kế Decorator.

#### Null Object

Khi không đăng ký đối tượng callback nào cho File, sử dụng mẫu thiết kế Null Object để tạo ra đối tượng callback "không làm gì".

## **Singleton**

Các đối tượng callback thường chỉ có một thể hiện duy nhất nên áp dụng mẫu thiết kế Singleton.

4. Chương trình Xem mã  
nguồn kèm theo sách.

## Tài liệu tham khảo

(Theo năm xuất bản)

- [1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides – **Design Patterns: Elements of Reusable Object-Oriented Software** – Addison-Wesley, 1995. ISBN 0201633612
- [2] Mark Grand – **Patterns in Java, Volume 1: A Catalog of Reusable Design Patterns Illustrated with UML, Second Edition** – John Wiley & Sons, 2002. ISBN 0471227293
- [3] Partha Kuchana – **Software Architecture Design Patterns in Java** – Auerbach Publications, 2004. ISBN 0849321425
- [4] Allen Holub – **Holub on Patterns: Learning Design Patterns by Looking at Code** – Apress, 2004. ISBN 978-1590593882
- [5] Joshua Kerievsky – **Refactoring to Patterns** – Addison-Wesley, 2004. ISBN 978-0321213358
- [6] Timothy C. Lethbridge, Robert Laganière – **Object-Oriented Software Engineering** – McGraw-Hill, 2005. ISBN 0077097610
- [7] Cay Horstmann – **Object-Oriented Design and Patterns, Second Edition** – John Wiley & Sons, 2006. ISBN 9780471744870
- [8] Steven John Metsker, William C. Wake – **Design Patterns in Java™** – Addison-Wesley, 2006. ISBN 0321333020
- [9] Paul R. Allen, Joseph J. Bambara – **SCEA Sun® Certified Enterprise Architect for Java™ EE Study Guide (Exam 310-051)** – McGraw-Hill, 2007. ISBN 0071510931
- [10] Christopher G. Lasater – **Design Patterns** – Wordware Publishing, 2007. ISBN 1598220314
- [11] Judith Bishop – **C# 3.0 Design Patterns** – O'Reilly, 2008. ISBN 978-0596527730
- [12] Joshua Bloch – **Effective Java™, Second Edition** – Addison-Wesley, 2008. ISBN 978-0321356683
- [13] Dale Skrien – **Object-Oriented Design using Java** – McGraw-Hill, 2009. ISBN 978-0072974164
- [14] Bernd Bruegge, Allen H. Dutoit – **Object-Oriented Software Engineering: Using UML, Patterns, and Java. Third Edition** – Prentice Hall, 2010. ISBN 13: 978-0-13-606125-0
- [15] Eddie Burris – **Programming in the Large with Design Pattern** – Pretty Print Press, 2012. ISBN 978-0072974164
- [16] Robert C. Martin – **Agile Software Development, Principles, Patterns, and Practices** – Pearson, 2014. ISBN 1292-02594-8
- [17] Craig Larman – **Applying UML & Patterns, Thrid Edition** – Pearson, 2015. ISBN 978-9332553941
- [18] Alexander Shvets – **Dive Into Design Patterns** – (e-book) 2018
- [19] Vaskaran Sarcar – **Java Design Patterns, Second Edition** – Apress, 2019. ISBN 978-1-4842-4077-9
- [20] Dmitri Nesteruk – **Design Patterns in .NET: Reusable Approaches in C# and F# for Object-Oriented Software Design** – Apress, 2019. ISBN 978-1-4842-4365-7