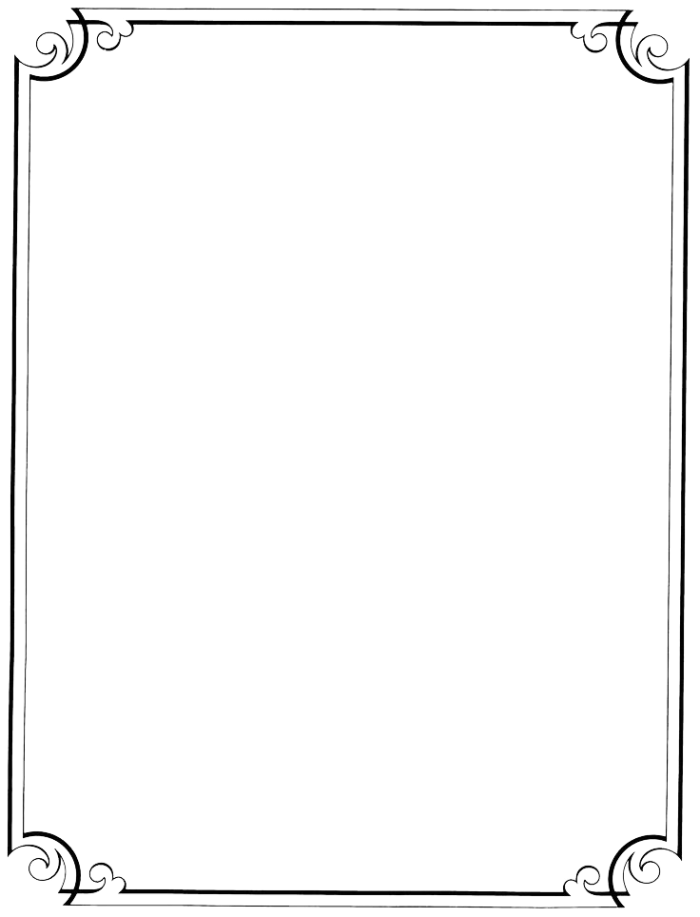
**BỘ GIÁO DỤC VÀ ĐÀO TẠO**

**TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT TP.HCM**

**KHOA ĐÀO TẠO CHẤT LƯỢNG CAO**

 ****

**BÁO CÁO MÔN MẪU THIẾT KẾ PHẦN MỀM**

**ĐỀ TÀI: ỨNG DỤNG QUẢN LÝ SINH VIÊN**

|  |  |
| --- | --- |
| **GVHD** | **Lê Văn Vinh** |
| **Lớp** | **Sáng thứ 5, tiết 3-4** |
| **Thực hiện** | **Nhóm** |
| **Thành viên** | **19110290 Nguyễn Mai Phương Thảo** |

Thành phố Hồ Chí Minh, ngày 17 tháng 6 năm 2022

CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM

 Độc lập – Tự do – Hạnh phúc

----\*\*\*----

**NHIỆM VỤ ĐỒ ÁN CUỐI KÌ**

**Họ và tên sinh viên**:

Nguyễn Mai Phương Thảo MSSV: 19110290

Vũ Đức Hậu MSSV:

Võ Thành Đạt MSSV:

**Ngành**: Công nghệ thông tin

**Giảng viên hướng dẫn**: Lê Văn Vinh

**Ngày nộp đề tài:** 19/06/2022

**Tên đồ án:** Phầm mềm Quản lý sinh viên có ứng dụng 5 mẫu thiết kế phần mềm

TRƯỞNG NGÀNH GIẢNG VIÊN HƯỚNG DẪN

CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM

Độc lập – Tự do – Hạnh phúc

----\*\*\*----

**PHIẾU NHẬN XÉT CỦA GIÁO VIÊN HƯỚNG DẪN**

**Họ và tên sinh viên**:

Nguyễn Mai Phương Thảo MSSV: 19110290

Vũ Đức Hậu MSSV:

Võ Thành Đạt MSSV:

**Ngành**: Công nghệ thông tin

**Tên đồ án:** Phầm mềm Quản lý sinh viên có ứng dụng 5 mẫu thiết kế phần mềm

**Giảng viên hướng dẫn**: Lê Văn Vinh

**NHẬN XÉT**

1. **Về nội dung đề tài và khối lượng thực hiện:**

**………………………………………………………………………………………………………………………………………………………………………………………………**

1. **Ưu điểm:**

**………………………………………………………………………………………………………………………………………………………………………………………………**

1. **Khuyết điểm:**

**………………………………………………………………………………………………………………………………………………………………………………………………**

1. **Điểm: ……………………………………………………………………………………**

Tp*. Hồ Chí Minh, ngày tháng năm 2022*

Giáo viên hướng dẫn

*(Ký & ghi rõ họ tên)*

# LỜI CẢM ƠN

Trong lời đầu tiên của báo cáo đề tài môn Mẫu thiết kế phần mềm, chúng em muốn gửi lời cảm ơn và biết ơn chân thành nhất của mình tới tất cả những người đã hỗ trợ, giúp đỡ chúng em về kiến thức vào tinh thần trong quá trình thực hiện báo cáo này.

Chúng em xin chân thành gửi lời cảm ơn đến các thầy cô giáo trong đại học Sư Phạm Kỹ Thuật TP.HCM nói chung và các thầy cô trong ngành Công Nghệ Thông Tin nói riêng đã tận tình giảng dạy, truyền đạt cho chúng em những kiến thức cũng như kinh nghiệm quý báu trong suốt quá trình học tập.

Đặc biệt, chúng em xin gửi lời cảm ơn đến giảng viên Lê Văn Vinh, giảng viên Khoa Công nghệ thông tin – trường đại học Sư Phạm Kỹ Thuật TP.HCM. Thầy đã tận tình theo sát giúp đỡ, trực tiếp chỉ bảo trong suốt quá trình nghiên cứu và học tập của chúng em. Trong thời gian học tập với thầy, chúng em không những tiếp thu thêm nhiều kiến thức bổ ích mà còn học tập được tinh thần làm việc, thái độ nghiên cứu, học tập nghiêm túc và hiệu quả. Đây là những yếu tố rất cần thiết cho chúng em trong quá trình học và làm việc sau này.

Do thời gian thực hiện có hạn cùng kiến thức còn nhiều hạn chế nên bài làm của chúng em chắc chắn sẽ không tránh khỏi những thiếu sót. Chúng em rất mong nhận được những ý kiến đóng góp quý báu của thầy và các bạn để có thêm kinh nghiệm và tiếp tục hoàn thành đồ án của mình.

Nhóm em xin chân thành cảm ơn!

**MỤC LỤC**

[LỜI CẢM ƠN 4](#_Toc106653037)

[I. DESIGN PATTERN LÀ GÌ? 1](#_Toc106653038)

[1. Tổng quan 1](#_Toc106653039)

[2. Sử dụng Design Pattern 1](#_Toc106653040)

[a. Vì sao nên sử dụng? 1](#_Toc106653041)

[b. Khi nào nên sử dụng? 2](#_Toc106653042)

[3. Phân loại Design Pattern 2](#_Toc106653043)

[II. CÁC DẠNG DESIGN PATTERN ĐƯỢC ÁP DỤNG 3](#_Toc106653044)

[1. Singleton Pattern 3](#_Toc106653045)

[a. Khái niệm 3](#_Toc106653046)

[b. Cấu trúc và cài đặt Singleton Pattern 3](#_Toc106653047)

[c. Ưu và nhược điểm 4](#_Toc106653048)

[d. Ứng dụng của Singleton Pattern 5](#_Toc106653049)

[e. Code mẫu về Singleton 5](#_Toc106653050)

[2. Iterator Pattern 6](#_Toc106653051)

[a. Khái niệm 7](#_Toc106653052)

[b. Cấu trúc của Iterator 7](#_Toc106653053)

[c. Ưu và nhược điểm 8](#_Toc106653054)

[d. Ứng dụng Iterator Pattern 9](#_Toc106653055)

[e. Code mẫu về Iterator Pattern 9](#_Toc106653056)

[3. Observer Pattern 12](#_Toc106653057)

[a. Khái niệm 12](#_Toc106653058)

[b. Cấu trúc của Observer Pattern. 13](#_Toc106653059)

[c. Ưu và nhược điểm Observer Pattern 14](#_Toc106653060)

[d. Ứng dụng Observer Pattern 15](#_Toc106653061)

[e. Code mẫu về Observer Pattern 15](#_Toc106653062)

[4. Template Method Pattern 19](#_Toc106653063)

[a. Khái niệm 19](#_Toc106653064)

[b. Cấu trúc của Template Method Pattern 19](#_Toc106653065)

[c. Ưu và nhược điểm Template Method Pattern 20](#_Toc106653066)

[d. Ứng dụng của Template Method Pattern 21](#_Toc106653067)

[e. Code mẫu về Template Method Pattern 21](#_Toc106653068)

[III. DESIGN PATTERN VÀ PHẦN MỀM 28](#_Toc106653069)

[1. Giới thiệu 28](#_Toc106653070)

[2. Chương trình quản lý trường học 28](#_Toc106653071)

[a. Mô tả chương trình 28](#_Toc106653072)

[b. Danh sách các chức năng 28](#_Toc106653073)

[3. Áp dụng Design Pattern vào bài toán 29](#_Toc106653074)

# DESIGN PATTERN LÀ GÌ?

## Tổng quan

Trong công nghệ phần mềm, người ta định nghĩ khái niệm về Design Pattern là một giải pháp tổng thể, tối ưu và được tái sử dụng cho các vấn đề chung trong thiết kế phần mềm. Design Pattern được coi là một tập hợp các giải pháp đã được suy nghĩ, và dùng để giải quyết trong một tình huống cụ thể nào đó.

Vậy các Design Pattern nó có thực sự quan trọng đối với quá trình phát triển một ứng dụng phần mềm hay không? Câu trả lời chắc chắc sẽ là có, vì khi áp dụng thuần phục được nó trong quá trình phát triển nó sẽ giúp sản phẩm của chúng ta linh hoạt, dễ dàng thay đổi và bảo trì.

## Sử dụng Design Pattern

### Vì sao nên sử dụng?

Những Design Pattern này có cấu trúc rõ ràng rành mạch, khi áp dụng vào chúng ta sẽ mang lại rất nhiều lợi ích cho việc thiết kế phần mềm. Và đó là những mẫu phổ biến và được sử dụng rất nhiều trong lập trình, và bắt buộc người lập trình viên nào cũng nên biết đến nó. Những lợi ích mà nó mang lại có thể kể đến như:

* Nó sẽ giúp sản phẩm của chúng ta trở nên linh hoạt, dễ dàng thay đổi và bảo trì hơn.
* Các giải pháp này giúp ta tăng tốc độ phát triển phần mềm bằng cách đưa ra các mô hình test, mô hình phát triển đã qua kiểm nghiệm.
* Design Pattern là hướng đi giúp cho lập trình viên gặp một vấn đề nào đó khó giải quyết thay vì tự tìm kiếm giải pháp tốn kém thời gian
* Vì là một quy chuẩn chung, nên các lập trình viên sẽ dễ dàng làm việc chung với nhau, hiểu code cùa người khác một cách nhanh chóng. Các thành viên trong một team có thể dễ dàng trao đổi với nhau để cùng xây dựng dự án mà không tốn kém quá nhiều thời gian.

### Khi nào nên sử dụng?

Sử dụng Design Pattern một cách hiệu quả giúp bạn có thể tiết kiệm được một khối thời gian và công sức để phải suy nghĩ ra một cách giải quyết cho những vấn đề mà nó ngay từ đầu đã có lời giải.

Design Pattern khi được áp dụng đúng cách vào quá trình phát triển phần mềm sẽ giúp cho chương trình đó trở nên uyển chuyển hơn, dễ dàng quản lý tiến trình hoạt động, dễ nâng bảo trì…

Tuy nhiên cũng không nên quá lạm dùng vào nó đối với những vấn đề khá đơn giản. Ví dụ một vấn đề đơn giản chúng ta có thể giải quyết trong 3-4 dòng code, khi chúng ta áp dụng design pattern, nó sẽ trở nên phức tạp hơn quá mức cần thiết.

Nhìn chung, chúng ta nên sử dụng các mẫu thiết kết này một cách đúng cách, vào những trường hợp cần thiết.

## Phân loại Design Pattern

Trong hệ thống các mẫu Design Pattern có đến 23 mẫu được định nghĩa trong cuốn “Design patterns Elements of Reusable Object Oriented Software” và được chia thành 3 nhóm:

* Nhóm khởi tạo (Creational Pattern)
* Nhóm cấu trúc (Structunal Pattern)
* Nhóm hành vi (Behavior Pattern)

Nhóm đầu tiên là nhóm Creational Pattern, nhóm này cung cấp cho chúng ta các giải pháp về việc khỏi tạo một object một cách gián tiếp, và trách thực hiện việc khởi tạo một cách trực tiếp như sử dụng method new. Điều này giúp cho chương trình trở nên mềm dẻo hơn trong việc quyết định object nào cần được tạo ra trong những tình huống cụ thể.

Nhóm thứ hai là nhóm Structunal Pattern, nhóm này sẽ liên quan đến các đối tượng và các thành phần của object. Nó dùng để thiết lập, định nghĩa quan hệ giữa các đối tượng.

Nhóm cuối cùng trong bộ ba nhóm là Behavioral Pattern thường dùng trong các hành vi của đối tượng, sự giao tiếp giữa các object với nhau.

# CÁC DẠNG DESIGN PATTERN ĐƯỢC ÁP DỤNG

## Singleton Pattern

Trong quá trình phân tích và phát triển của một hệ thống ứng dụng. Chắc chắn sẽ có những lúc bài toán yêu cầu chúng ta trong hệ thống chỉ sở hữu một đối tượng tồn tại duy nhất. Và đối tượng duy nhất này có thể truy xuất nó bất kì nơi nào trong hệ thống. Từ đây mẫu Singleton Pattern ra đời để giải quyết vấn đề này.

### Khái niệm

Singleton Pattern là một trong những mẫu thiết kế thuộc nhóm khỏi tạo Creational pattern. Singleton được định nghĩa là một lớp, và nó chỉ tồn tại đúng duy nhất một thể hiện **(instance)** trong chương trình của chúng ta.

Và những lớp khác trong hệ thống muốn truy xuất đến nó phải sử dụng một hàm mà Singleton này cung cấp để trả về một **instance** duy nhất.

### Cấu trúc và cài đặt Singleton Pattern

* **Cấu trúc**

Để hiểu rõ hơn về mẫu Singleton này, chúng ta hãy xem xét một cấu trúc của một lớp Singleton đơn giản nhất như hình bên dưới.

Text

Description automatically generated

Chúng ta có thể phân tích lớp Singleton này:

* Lớp này đã đảm bảo được chỉ có một instance duy nhất. Vì hàm constructor trong lớp này sử dụng là private. Điều này khiến cho những lớp khác không thể khởi tạo thêm một instance nào khác trong hệ thống nữa
* Lớp cung cấp cho chúng ta một phương thức getInstance để đảm bảo cho việc chỉ trả về đúng một instance duy nhất đã được tạo trước đó trong lớp.
* **Cài đặt**

Có rất nhiều cách để implement Singleton Pattern. Tuy nhiên dù là sử dụng cách gì đi nữa thì điều sẽ dựa vào những nguyên tắc như:

* Hàm constructor là private để hạn chế truy cập từ class bên ngoài, không cho tạo mới instance.
* Đặt private static final variable để đảm bảo biến chỉ được khởi tạo trong class
* Có một method public static dể return instance được khởi tạo của lớp.

### Ưu và nhược điểm

Singleton được sử dụng thông dụng trong các trường hợp chúng ta muốn có một đối tượng tồn tại duy nhất trong hệ thống, và nó cũng sẽ có các ưu và nhược điểm.

|  |  |
| --- | --- |
| **Ưu điểm** | **Nhược điểm** |
| * Chỉ tồn tại duy nhất một instance trong xuyên suốt chương trình * Có thể truy cập instance ở bất kì đâu và bất kì khi nào muốn trong hệ thống * Lớp Singleton chịu trách nhiệm cho việc kiểm soát sự khởi tạo của chính nó. | * Khó triển khai 1 cách hiệu quả để một class chỉ tồn tại đúng một đối tượng * Hạn chế số lượng instance của một class * Việc kiểm soát chỉ tồn tại duy nhất môn instance đôi khi không được như ý muốn |

### Ứng dụng của Singleton Pattern

Vì Singleton Pattern được định nghĩa là một class chỉ tồn tại duy nhất một thể hiện trong một chương trình, nên trong thực thế nó sẽ thường sử dụng trong các trường hợp như: Shared resource, Logger, Configuration, Caching, Thread pool

Hỗ trợ một số design pattern khác, khi những mẫu khác có áp dụng Singleton này như: Abstract Factory, Builder, Facade…

### Code mẫu về Singleton

* **Đặt vấn đề**

Khi triển một phần mềm ứng dụng, rất nhiều lúc chúng ta sẽ làm việc với API. Và nhiều lớp trong chương trình của chúng ta muốn truy xuất và sử dụng đến API này. Ở trường hợp này chúng ta có thể áp dụng mẫu Singleton để quản lý việc đó.

* **Giải quyết vấn đề**

Đầu tiên chúng ta khởi tạo một class APISingleton với một thể hiện duy nhất là instance và hàm constructor của lớp này sẽ là private để tránh trường hợp tạo mới.

public class APISingleton {  
 private static APISingleton *instance*;  
 private APISingleton(){};  
 public static APISingleton getInstance(){  
 if(*instance* == null) {  
 *instance* = new APISingleton();  
 }  
 return *instance*;  
 }  
 public String getAPI() {  
 return "https://randomuser.me/api/";  
 }  
  
}

Trong lớp APISingleton này chúng ta sẽ có hàm getInstance để trả về một đối tượng instance của lớp. Và một hàm để các get API.

Như vậy là chúng ta đã làm một ví dụ đơn giản về mẫu Singleton này. Bây giờ khi một lớp nào muốn sử dụng chỉ cần gọi đến lớp APISingleton này.

public class Main {  
 public static void main(String[] args) throws IOException {  
 String data;  
 HttpDataHandler httpDataHandler = new HttpDataHandler();  
 data = httpDataHandler.getHttpData(APISingleton.*getInstance*().getAPI());  
 System.*out*.println(data);  
 }  
}

## Iterator Pattern

Trong quá trình phát triển phần mềm, sẽ có rất nhiều cách để lưu một tập hợp như là: mảng, tập hợp, cây, bảng băm, stack, queue… Đối mới mỗi các lưu tập hợp, chúng ta lại có rất nhiều cách để truy xuất các phần tử trong các tập hợp này. Điều đó sẽ khiến chúng ta tạo ra nhiều phương thức để truy xuất tập hợp.

Iterator Pattern ra đời để cung cấp cho chúng ta một giải pháp chung để truy xuất các phần tử trong những tập hợp khác nhau, mà không cần quan tâm loại tập hợp đó là gì. Sử dụng phương pháp này, chúng ta có thể truy xuất tới các phần tử trong tập hợp theo một cách dễ dàng nhất.

### Khái niệm

Iterator Pattern là một trong những Pattern thuộc nhóm Behavior Pattern. Mẫu này sẽ cung cấp cho ta một phương thức truy cập tuần tự các phần tử của một đối tượng tập hợp, mà không cần tạo dựng riêng các phương pháp truy cập cho đối tượng tổng hợp này.

Mẫu này được thiết kế để chúng ta có thể xử lý với nhiều loại tập hợp khác nhau, mà không cần quan tâm tập hợp này thuộc loại gì. Chúng ta sẽ xử lý các tập hợp với cùng một phương pháp định sẵn.

Để thực hiện được điều này, Iterator Pattern định nghĩa cho chúng ta 2 phương thức sau:

* Hàm next(): dùng để trả về phần tử kế tiếp trong tập hợp
* Hàm hasNext(): dùng để kiểm tra phần tử cuối cùng trong tập hợp. Trả về true nếu như còn phần tử trong mảng, và trả về false nếu trong tập hợp không còn phần tử nào

Diagram

Description automatically generated

Có thể nói Interator là cầu nối giữa client code và tập hợp. Giúp cho về xử lý trên tập hợp trở nên đơn giản hơn.

### Cấu trúc của Iterator

Các thành phần cấu trúc lên một mẫu Iterator Pattern được thể hiện bên dưới

Diagram

Description automatically generated

* **Aggregate:** là một interface nó dùng để định nghĩa các phương thức để tạo Iterator object
* **ConcreteAggregate:** implements Aggreagte, cài đặt interface tạo Iterator để trả về một thể hiện ConcreteIterator thích hợp.
* **Iterator:** là một interface hoặc abstract class, định nghĩa các phương thức để truy cập, duyệt các phần tử.
* **ConcreteIterator:** implements Iterator, cài đặt các phương thức Iterator, giữ index khi duyệt qua các phần tử trong tập hợp.
* **Client:** đối tượng sử dụng Iterator Pattern

### Ưu và nhược điểm

|  |  |
| --- | --- |
| Ưu điểm | Nhược điểm |
| * Chúng ta có thể truy cập song song trên cùng một tập hợp vì mỗi đối tượng iterator có chứa trạng thái riêng của nó * Trong một số trường hợp, ta có thể trì hoãn việc lập lại và tiếp tục nó khi cần thiết | * Kém hiệu quả hơn so với việc duyệt qua các phần tử của tập hợp một cách trực tiếp * Không cần thiết nếu ứng dụng hoạt động chỉ với một tập hợp đơn giản |

### Ứng dụng Iterator Pattern

Vì mẫu này sẽ cung cấp cho chúng ta phương thức truy cập các phần tử nhiều tập hợp khác nhau, nên ứng dụng của nó sẽ thường được sử dụng trong các trường hợp dưới đây:

* Khi chúng ta không biết được nội dung cũng như cấu trúc của tập hợp mà chúng ta định truy xuất là gì.
* Sử dụng Iterator sẽ giúp chúng ta tránh trường hợp bị trùng lặp source code quá nhiều. Vì mỗi tập hợp, là sử dụng một phương thức khác nhau. Điều này cũng khiến code của chúng ta sẽ nhiều hơn, khó bảo trì và nâng cấp mở rộng sau này.
* Việc dùng Iterator để duyệt qua các phần tử của một tập hợp, khiến code của chúng ta dễ theo dỗi các cấu trúc dữ liệu khác nhau hoặc các loại cấu trúc chuỗi chưa được biết trước.

### Code mẫu về Iterator Pattern

* **Đặt vấn đề**

Giả sử chúng ta có một lớp để quản lý lưu trữ một nhóm học sinh trong một lớp học. Chúng ta sẽ áp dụng mẫu Iterator này để duyệt qua các phần tử trong tập hợp học sinh này.

* **Giải quyết vấn đề**

Đầu tiên ta sẽ tạo một interface chứa các phương thức để truy cập và duyệt qua các phần tử trong một tập hợp.

public interface StudentIterator<T>{  
 boolean hasNext();  
 T next();  
}

Tiếp theo ta tạo một interface thực hiện việc tạo ra một đối tượng Iterator

public interface Container {  
 public StudentIterator getIterator();  
}

Tạo một lớp học sinh với thông tin cơ bản là tên của học sinh đó

public class Student {  
 private String name;  
  
 public Student(String name) {  
 this.name = name;  
 }  
 @Override  
 public String toString() {  
 return name ;  
 }  
}

Tiếp tục ta sẽ tạo ra lớp để quản lý học sinh trong một nhóm, lớp này sẽ implement interface **Container** để tạo ra Iterator và trả về một lớp xử lý việc duyệt phần tử trong tập hợp tương ứng.

public class StudentGroup implements Container {  
 private List<Student> studentList = new ArrayList<>();  
 public void addStudent(Student student){  
 studentList.add(student);  
 }  
  
 @Override  
 public StudentIterator getIterator() {  
 return new StudentGroupIterator();  
 }  
 private class StudentGroupIterator implements StudentIterator {  
 private int index = 0;  
 @Override  
 public boolean hasNext() {  
 if(index < studentList.size()){  
 return true;  
 }  
 else return false;  
 }  
 @Override  
 public Object next() {  
 return studentList.get(index++);  
 }  
 }  
  
}

Ở đây ta có thể nhận thấy lớp **StudentGroupIterator** implement **StudentIterator** và override 2 hàm là hasNext() và next(). Hai hàm để chúng ta có thể thực hiện việc duyệt qua các học sinh trong một nhóm.

Sử dụng hàm main để test hoạt động của Iterator Pattern. Tạo một đối tượng mới là một studentGroup, và sử dụng hàm getIterator() để trả về một thể hiện Iterator.

Và dùng vòng lặp while, duyệt tới khi nào hết học sinh trong nhóm thì dừng lại.

public class Main {  
 public static void main(String[] args) {  
 StudentGroup studentGroup = new StudentGroup();  
 studentGroup.addStudent(new Student("Nguyen Trong Huy"));  
 studentGroup.addStudent(new Student("Pham Thanh Loi"));  
 studentGroup.addStudent(new Student("Nguyen Trong Tin"));  
  
 StudentIterator<Student> studentIterator = studentGroup.getIterator();  
 while (studentIterator.hasNext()){  
 Student sd = studentIterator.next();  
 System.*out*.println(sd);  
 }  
 }  
}

Chương trình thực thi và hiện kết quả như bên dưới

Text

Description automatically generated

Với cách sử dụng Iterator để duyệt qua nhóm học sinh, ta có thể thấy rằng, lập trình viên khi sử dụng tương tác với một tập hợp, họ không cần phải quan tâm kiểu dữ liệu cũng như cấu trúc của tập hợp đó là gì. Họ chỉ cần dùng Iterator Pattern, sử dụng hai hàm cơ bản là next() và hasNext() đã được định nghĩa từ trước để duyệt qua nhóm học sinh.

## Observer Pattern

### Khái niệm

Observer Pattern là một trong những Pattern thuộc nhóm Behavior Pattern. Nó sẽ định nghĩa một mối quan hệ một nhiều giữa các đối tượng trong chương trình. Khi mà một đối tượng nhất định cần được thông báo thường xuyên về những thay đổi xảy ra trong một đối tượng khác trong chương trình.

Từ đó mẫu Observer này được sử dụng để quản lý một đối tượng về bất kỳ sự thay đổi trạng thái nào, và thông báo cho cái đối tượng phụ thuộc khác đã đăng ký từ trước một cách tự động.

Trong mẫu Observer có 2 thành phần quan trọng chính đó là Subject và Observer:

* Subject: được gọi cho đối tượng chứa các nguồn sự kiện, tin tức
* Observer: được gọi cho các đối tượng phụ thuộc vào Subject, tiếp nhận các sự kiện, tin tức và xử lý.

Mối quan hệ giữa Subject và Observer được thể hiện rõ qua việc khi chúng ta đăng ký một kênh Youtube.

Diagram

Description automatically generated

Ở đây kênh Youtube sẽ là Subject và sẽ cho các User đăng ký hay còn gọi là Observer. Với việc đăng ký này, khi kênh Youtube này đăng một video mới, nó sẽ gửi thông báo đến các Observer đã được đăng ký nó từ trước. Sơ đồ trên thể hiện rõ quy tắc hoạt động của một mẫu Observer Pattern.

### Cấu trúc của Observer Pattern.

Các thành phần cấu trúc lên một mẫu Observer Pattern được thể hiện bên dưới

A picture containing diagram

Description automatically generated

* **Subject :** chứa danh sách các observer, cung cấp các phương thức có thể thêm hoặc loại bỏ observer.
* **Observer :** định nghĩa một phương thức update cho các observer nhận được thông báo khi có sự thay đổi của subject.
* **ConcreteSubject :** cài đặt các phương thức của Subject, lưu trữ trạng thái danh sách các ConcreteObserver, và gửi thông báo đến Observer khi có bất kì thay đổi nào.
* **ConcreteObserver :** cài đặt các phương thức của Observer, lưu trữ trạng thái của Subject, thực thi việc cập nhật khi subject thông báo đến

### Ưu và nhược điểm Observer Pattern

|  |  |
| --- | --- |
| Ưu điểm | Nhược điểm |
| * Dễ dàng mở rộng và ít sự thay đổi vì mẫu này cho phép ta có thể thay đổi Subject và Observer một cách độc lập * Thêm Observer mà không phải sửa đổi Subject hoặc Observer khác * Khi một đối tượng thay đổi sẽ được thông báo đến các đối tượng khác mà chúng không cần phải liên kết chặt chẽ * Có thể thông báo không giới hạn Observer | * Không kiểm soát được thứ tự các Observer nào nhận được thông báo |

### Ứng dụng Observer Pattern

Ứng dụng của mẫu Observer Pattern khá là phổ biến trong các ứng dụng thường sử dụng mối quan hệ một nhiều đối với các đối tượng với nhau. Khi một đối tượng cần thông báo cho những đối tượng khác những thay đổi của mình.

Thường được sử dụng nhiều trong quản lý sự kiện, và mô hình MVC.

### Code mẫu về Observer Pattern

* **Đặt vấn đề**

Giả sử chúng ta mô phỏng việc một kênh Youtube sẽ có những user khác vào đăng ký, khi kênh Youtube này hoạt động và thực hiện một việc thay đổi nào đó, ví dụ như thêm một video mới vào kênh. Những người đã đăng ký kênh Youtube này sẽ nhận được thông báo.

Ở đây ta có thể nhận thấy kênh Youtube là một Subject và các User còn lại đăng ký kênh Youtube này được gọi là Observer

* **Giải quyết vấn đề**

Đầu tiên ta tạo interface **Subscriber** định nghĩa phương thức notify, để cho các Observer có thể nhận được thông báo khi có sự thay đổi của kênh Youtube

public interface Subscriber {  
 void notify(YoutubeChannel youtubeChannel);  
}

Tiếp theo là tạo một interface khác là IyoutubeSubscriber định nghĩa các phương thức như đăng ký, hủy đăng ký, và gửi thông báo. Và lớp YoutubeChannel sẽ implement interface này.

public interface IYoutubeSubscriber {  
 void subscribe (Subscriber subscriber);  
 void unsubscribe (Subscriber subscriber);  
 void notifyAllSubscriber();  
}

public class YoutubeChannel implements IYoutubeSubscriber {  
 private String name;  
  
 public String getName() {  
 return name;  
 }  
 private List<Video> videos = new ArrayList<>();  
 private List<Subscriber> subscribers = new ArrayList<>();  
  
  
 public YoutubeChannel(String name) {  
 this.name = name;  
 }  
  
 @Override  
 public void subscribe(Subscriber subscriber) {  
 if(!subscribers.contains(subscriber)){  
 subscribers.add(subscriber);  
 }  
 }  
  
 @Override  
 public void unsubscribe(Subscriber subscriber) {  
 if(subscribers.contains(subscriber)){  
 subscribers.remove(subscriber);  
 }  
 }  
  
 @Override  
 public void notifyAllSubscriber() {  
 for (Subscriber subscriber : subscribers){  
 subscriber.notify(this);  
 }  
 }  
  
 public void uploadVideo(String title, String url){  
 Video video = new Video(title,url);  
 videos.add(video);  
 this.notifyAllSubscriber();  
 }  
  
 public Video getNewVideo(){  
 return videos.get(videos.size() - 1);  
 }  
}

Ở lớp YoutubeChannel ta sẽ quản lý một list các Subscriber đã đăng ký, và sẽ thông báo cho họ khi chúng ta thực hiện việc thay đổi nào đó, ví dụ như thực hiện phương thức uploadVideo, để thêm một video mới vào List Video của kênh Youtube đó.

Tạo một lớp User và implement interface Subscriber và thực hiện override phương thức notify, thông báo ra nội dung video mới mà kênh Youtube (Subject) đó vừa đăng tải.

public class User implements Subscriber{  
 private String name;  
  
 public User(String name) {  
 this.name = name;  
 }  
 @Override  
 public void notify(YoutubeChannel youtubeChannel) {  
 System.*out*.println( name + " nhan duoc thong bao | " + "Youtuber " + youtubeChannel.getName() +  
 " tai len video : " + youtubeChannel.getNewVideo().getTitle() + " - " + youtubeChannel.getNewVideo().getUrl());  
 }  
}

Dùng hàm main để test mẫu Observer Pattern này. Ta tạo ra 3 user, và 2 channel Youtube. Trong đó User 1 sẽ đăng ký kênh Youtube 1, User 3 đăng ký kênh Youtube 2 và User 2 đăng ký cả 2 kênh Youtube.

public class Main {  
 public static void main(String[] args) {  
 User user1 = new User("User 1");  
 User user2 = new User("User 2");  
 User user3 = new User("User 3");  
  
 YoutubeChannel channel = new YoutubeChannel("Huy");  
 channel.subscribe(user1);  
 channel.subscribe(user2);  
 channel.uploadVideo("Vlog 1","youtube.com/vlog1");  
  
  
 YoutubeChannel channel2 = new YoutubeChannel("Huy2");  
 channel2.subscribe(user2);  
 channel2.subscribe(user3);  
 channel2.uploadVideo("Vlog 2","youtube.com/vlog2");  
 }  
}

Ta có thể nhận thấy ở kết quả dưới đây, những user vào đăng ký kênh Youtube nào mới được nhận thông báo từ kênh đó. Trường hợp User 2 đăng ký cả 2 kênh, nên nhận một lúc 2 thông báo từ 2 kênh khác nhau.

Text

Description automatically generated with low confidence

Vậy là ta đã áp dụng được mẫu Observer Pattern trong mối quan hệ một nhiều thành công, một kênh Youtube với bất kì thay đổi nào nó sẽ thông báo cho các Subscriber khác đã đăng ký trước đó biết một cách tự động.

## Template Method Pattern

### Khái niệm

Template Method Pattern là một trong những Pattern thuộc nhóm Behavior Pattern. Nó sẽ định nghĩa cho chúng ta một bộ khung của một thuật toán trong một chức năng, và chuyển giao việc thực hiện nó cho các lớp con. Và các lớp con có thể định nghĩa lại cách thực hiện của một thuật toán mà không phải thay đổi cấu trúc của thuật toán đó.

Khi trong quá trình phát triển ứng dụng, các đối tượng cụ thể đôi khi sẽ có cùng một cách thực hiện một chức năng nào đó, nhưng các bước thực hiện có thể khác nhau ở một vài bước. Nếu như chúng ta tách riêng việc thực hiện chức năng đấy ra, thì sẽ bị dư thừa code khá nhiều, vì sẽ có những bước giống nhau. Vì thế ta nên sử dụng mẫu Template Method này, để tạo nên một cách thức truy cập giống nhau nhưng hành động và kết quả sẽ khác nhau.

Trong mẫu Template Method, thì abstract class sẽ định nghĩa một template method để thực hiện một chức năng nào đó. Khi muốn sử dụng chúng ta sẽ gọi template method này để tạo dựng lên một bộ khung.

### Cấu trúc của Template Method Pattern

Các thành phần cấu trúc lên một mẫu Template Method Pattern được thể hiện bên dưới

Diagram

Description automatically generated

* **AbstractClass:** trong lớp này nó sẽ định nghĩa các phương thức cho từng bước thực hiện một chức năng cần thiết và các lớp con có thể điều chỉnh các phương thức này. Và một phương thức templateMethod() để gọi các bước riêng lẻ đã được cài đặt ở lớp con.
* **ConcreteClass:** extends AbstractClass, lớp con có thể ghi đè lên các phương thức trừu tượng của lớp cha để cung cấp các khai triển thực sự. Và không thể ghi đè lên phương thức templateMethod();

### Ưu và nhược điểm Template Method Pattern

|  |  |
| --- | --- |
| **Ưu điểm** | **Nhược điểm** |
| * Tái sử dụng code, tránh việc duplicate code. * Đưa những phần trùng lặp vào một lớp cha * Cho phép lớp con có thể điều chỉnh một số phần nhất định của thuật toán. | * Template method có càng nhiều bước để override càng khó trong việc bảo trì |

### Ứng dụng của Template Method Pattern

Việc sử dụng mẫu Template Method cũng khá là phổ biến, nếu chúng ta cần giải quyết một thuật toán mà nó có quá nhiều bước trùng lặp ở những lớp con. Điều này chúng ta sẽ khai triển phương thức trừu tượng cho các bước thực hiện thuật toán đó vào một lớp duy nhất, và các lớp con có thể override để điều chỉnh cho phù hợp.

### Code mẫu về Template Method Pattern

* **Đặt vấn đề**

Giả sử ứng dụng của chúng ta có 2 lớp là Coffee và Tea, hai lớp này điều có phương thức để thực hiện các bước tạo ra một ly Coffee hoặc ly Trà. Tuy nhiên ở chức năng pha chế này, ta nhận thấy có rất nhiều bước mà 2 phương thức này giống nhau. Ví dụ như là đun nước và cho nước vào cốc.

Vì vậy ta cần áp dụng mẫu Template này để định dạng một bộ khung, khi muốn qua chế một thức uống nóng nào đó.

* **Giải quyết vấn đề**

Đầu tiên ta sẽ tạo một lớp abstract PrepareTemplate gồm các phương thức bước thực hiện việc pha chế. Và hàm prepare() để thực hiện các bước pha chế

public abstract class PrepareTemplate {  
  
 protected void boilWater(){  
 System.*out*.println("Boiling water");  
 };  
 protected void pourInCup(){  
 System.*out*.println("Pouring into cup");  
 }  
 abstract void brew();  
 abstract void addCondiments();  
 public final void prepare(){  
 boilWater();  
 brew();  
 pourInCup();  
 addCondiments();  
 System.*out*.println("\nComplete !!!");  
 };  
}

Ta nhận thấy các bước riêng rẻ, để các lớp con có thể điều chỉnh là 2 phương thức abstract brew và addCondiments. Vì đối với mỗi loại thức uống khác nhau, ta lại cho màu những loại gia vị khác nhau.

Tạo 2 lớp Tea và Coffer extend lớp Template này, và override lại 2 phương thức abstract. Ở riêng mỗi 2 lớp ta điều chỉnh lại 2 phương thức này theo cách riêng của từng lớp.

public class Coffee extends PrepareTemplate {  
 private Scanner scanner = new Scanner(System.*in*);  
 private Integer choose = null;  
 @Override  
 void brew() {  
 menuCoffee();  
 choose = Integer.*parseInt*(scanner.nextLine());  
 switch (choose){  
 case 1:  
 System.*out*.println("Choose Espresso Coffee");  
 break;  
 case 2:  
 System.*out*.println("Choose Latte Coffee");  
 break;  
 case 3:  
 System.*out*.println("Choose Americano Coffee");  
 break;  
 }  
 }  
 @Override  
 void addCondiments() {  
 menuCondiments();  
 choose = Integer.*parseInt*(scanner.nextLine());  
 switch (choose){  
 case 1:  
 System.*out*.println("Put Sugar in Cup");  
 break;  
 case 2:  
 System.*out*.println("Put Milk in Cup");  
 break;  
 case 3:  
 System.*out*.println(" ");  
 break;  
  
 }  
 }  
 private void menuCoffee(){  
 System.*out*.println("\n-----------Choose Coffee------------");  
 System.*out*.println("1. Espresso Coffee");  
 System.*out*.println("2. Latte Coffee");  
 System.*out*.println("3. Americano Coffee");  
 System.*out*.println("---------------------------");  
 System.*out*.print("Please choose: ");  
 }  
 private void menuCondiments(){  
 System.*out*.println("\n-----------Choose Condiments------------");  
 System.*out*.println("1. Sugar");  
 System.*out*.println("2. Milk");  
 System.*out*.println("3. No Condiments");  
 System.*out*.println("---------------------------");  
 System.*out*.print("Please choose: ");  
 }  
}

import java.util.Scanner;  
  
public class Tea extends PrepareTemplate{  
 private Scanner scanner = new Scanner(System.*in*);  
 private Integer choose = null;  
 @Override  
 void brew() {  
 menuTea();  
 choose = Integer.*parseInt*(scanner.nextLine());  
 switch (choose){  
 case 1:  
 System.*out*.println("Choose Green Tea");  
 break;  
 case 2:  
 System.*out*.println("Choose Black Tea");  
 break;  
 case 3:  
 System.*out*.println("Choose Oolong Tea");  
 break;  
 }  
 }  
 @Override  
 void addCondiments() {  
 menuTopping();  
 choose = Integer.*parseInt*(scanner.nextLine());  
 switch (choose){  
 case 1:  
 System.*out*.println("Put Sugar in Cup");  
 break;  
 case 2:  
 System.*out*.println("Put Honey in Cup");  
 break;  
 case 3:  
 System.*out*.println("Put Ginger in Cup");  
 break;  
 case 4:  
 System.*out*.println(" ");  
 break;  
  
 }  
 }  
 private void menuTea(){  
 System.*out*.println("\n-----------Choose Tea------------");  
 System.*out*.println("1. Green Tea");  
 System.*out*.println("2. Black Tea");  
 System.*out*.println("3. Oolong Tea");  
 System.*out*.println("---------------------------");  
 System.*out*.print("Please choose: ");  
 }  
 private void menuTopping(){  
 System.*out*.println("\n-----------Choose Condiments------------");  
 System.*out*.println("1. Sugar");  
 System.*out*.println("2. Honey");  
 System.*out*.println("3. Ginger");  
 System.*out*.println("4. No Condiments");  
 System.*out*.println("---------------------------");  
 System.*out*.print("Please choose: ");  
 }  
}

Tiếp theo ta sẽ dùng hàm main để test chương trình, ta tạo lần lượt 2 đối tượng coffee và tea và thực hiện phương thức prepare();

public class Main {  
 public static void main(String[] args) {  
 PrepareTemplate coffee = new Coffee();  
 PrepareTemplate tea = new Tea();  
 coffee.prepare();  
 System.*out*.println("\n--------------------------");  
 tea.prepare();  
 }  
}

Và 2 lớp con là Coffee và Tea đã đinh nghĩa lại 2 phương thức brew và addCondiments, nên tuy là cả 2 đều sử dụng chung 1 lớp cha Template, nhưng kết quả trả về thì lại khác nhau, tùy thuộc vào lớp con điều chỉnh.

Text

Description automatically generated

Table

Description automatically generated

Ta có thể nhận thấy ở 2 bước brew và addCondiments, với lớp Coffee người dùng sẽ chọn loại Coffee và gia vị riêng biệt, khác với lớp Tea.

Ở đây ta có thể thấy ứng dụng của mẫu Template Method Pattern, khi chúng ta có nhiều lớp con thực hiện một phương thức chức năng nào đó giống nhau ở nhiều bước thực hiện, ta nên định nghĩa một bộ khung, thực hiện những bước chung ở lớp cha, và những.

# DESIGN PATTERN VÀ PHẦN MỀM

## Giới thiệu

Các mẫu Design Pattern được ứng dụng khá nhiều trong quá trình phát triển phần mềm. Với việc áp dụng tốt các mẫu sẽ giúp chương trình của chúng ta phát triển nhanh hơn, khả năng bảo trì, tái sử dụng và đảm bảo tính đúng đắn của chương trình.

Chương trình được chọn để áp dụng các mẫu Design Pattern trong quá trình phát triển chương trình là chương trình quản lý trường học. Trong quá trình phát triển chương trình sẽ áp dụng 4 mẫu Design Pattern đã được học để thực hiện.

## Chương trình quản lý trường học

### Mô tả chương trình

Chương trình quản lý trường học sẽ ứng dụng trong môi trường giáo dục, với nhu cầu quản lý sinh viên, giảng viên, phòng học, môn học và điểm số.

Trong quá trình phát triển mà xây dựng ứng dụng quản lý môn học, sẽ áp dụng các mẫu Design Pattern để giải quyết các vấn đề trong quản lý. Điều này sẽ khiến ứng dụng chúng ta code trở nên dễ hiểu, dễ bảo trì và nhân cấp sau này.

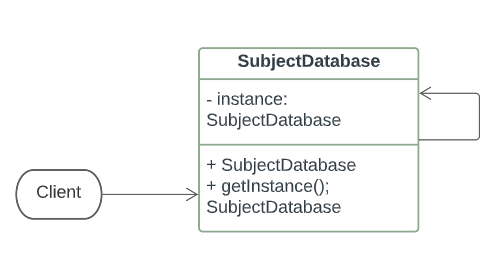
Chương trình quản lý 4 đối tượng chính đó là sinh viên, môn học và giảng viên và lớp học.

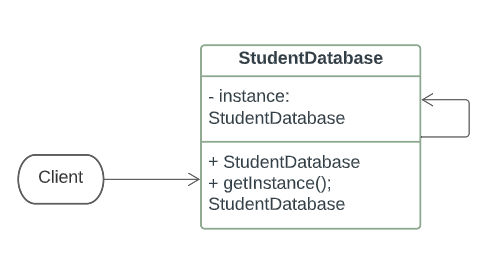
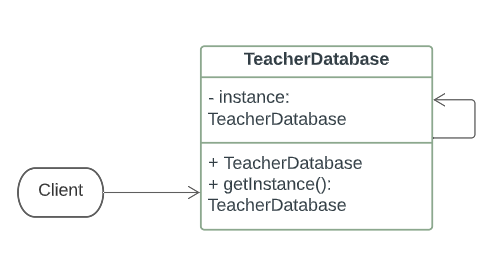
### Danh sách các chức năng

Chương trình có những chức năng cơ bản cho một chương trình quản lý lớp học

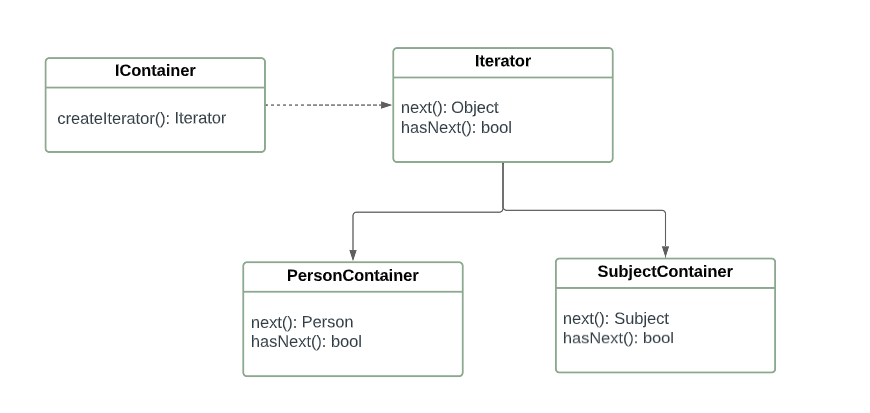
* Quản lý sinh viên
* Quản lý giảng viên
* Quản lý môn học
* Quản lý lớp học
* Quản lý điểm số

## Áp dụng Design Pattern vào bài toán

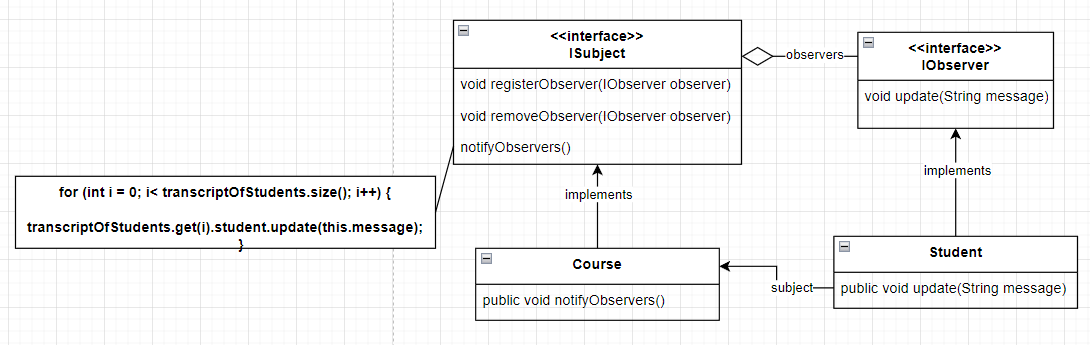
* + 1. Singleton Pattern:



* + 1. Iterator Pattern:



* + 1. Observer Pattern:



* + 1. Template Method Pattern:

