1. **Command Pattern:**
   * **Các lớp liên quan:** Command, ConvertCtoF, ConvertFtoC. Command Là lớp trừu tượng đại diện cho một lệnh (command) cụ thể, có phương thức execute() để thực hiện hành động.

* **Ý nghĩa của** **Command Pattern:** Giúp tách rời yêu cầu của người dùng thành các đối tượng lệnh cụ thể (ConvertCtoF, ConvertFtoC), dễ dàng mở rộng và tái sử dụng chúng.

Ví dụ: ConvertCtoF và ConvertFtoC: là 2 lớp đại diện cho lệnh (command), cụ thể thông qua phương thức execute của lớp trừu tượng TemperatureCommand từ đó triển khai execute() cho từng đại diện lệnh (command) để thực hiện hành động chuyển đổi nhiệt độ từ độ C sang độ F và độ F sang độ C và đc TemperatureCommand\_processor của mẫu **Singleton** thực hiện và quản lý các lệnh (commands) để chuyển đổi nhiệt độ. điển hình như code bên dưới:

 public *void* ConvertFC(){

Doublef=Double.parseDouble(temperatureViewRemote.jTextFieldInputRemote2().getText());

TemperatureCommand commandRemote = **new** ConvertFtoC(f,temperatureModelRemote);

temperatureCommand\_processor.execute2(commandRemote);

                }

1. **Singleton Pattern:**
   * **Lớp liên quan:** TemperatureCommand\_processor.
   * **Ý Nghĩa của Singleton Pattern:** Sử dụng Singleton để đảm bảo rằng chỉ có một Đại diện duy nhất của lớp tồn tại
   * **Ví Dụ code điển hình:**

  private static TemperatureCommand\_processor temperatureCommand\_processor ;//Lưu trữ Đại diện duy nhất của lớp, chỉ có thể được truy cập từ bên trong lớp.

    private TemperatureCommand\_processor(){};// Constructor của lớp Singleton được khai báo là private, điều này ngăn chặn việc tạo Đại diện mới từ bên ngoài lớp(không thể sử dụng toán tử new để tạo một đối tượng mới của lớp Singleton).

    //Phương thức này kiểm tra xem Đại diện đã được tạo chưa. Nếu chưa, nó sẽ tạo mới Đại diện và trả về nó; nếu đã có Đại diện, nó sẽ đơn giản trả về Đại diện đó.Điều này giúp đảm bảo rằng không có nhiều hơn một phiên bản của TemperatureCommand\_processor được tạo ra, và tất cả các phần của ứng dụng sẽ sử dụng cùng một Đại diện duy nhất này.

  public static TemperatureCommand\_processor makTemperatureCommand\_processor(){

    if(temperatureCommand\_processor == null){

          temperatureCommand\_processor = **new** TemperatureCommand\_processor();

        }

        return temperatureCommand\_processor;

    }

Ví dụ, trong lớp Controller của ta:

* **viewRemote:** Là một Đại diện của lớp View, đại diện cho giao diện người dùng để hiển thị và tương tác với dữ liệu nhiệt độ.
* **modelRemote:** Là một Đại diện của lớp Model, chứa dữ liệu nhiệt độ và logic xử lý chuyển đổi nhiệt độ.
* **command\_processor**:Là một Đại diện của lớp Command\_processor, được quản lý bởi mẫu Singleton để thực hiện và quản lý các lệnh **(commands)** để chuyển đổi nhiệt độ. **điển hình như code bên dưới thể hiện** lớp Command\_processor thuộc singelton **pattern quản lí các lệnh trong command pattern:**

 public *void* ConvertFC(){

Doublef=Double.parseDouble(temperatureViewRemote.jTextFieldInputRemote2().getText());

TemperatureCommand commandRemote = **new** ConvertFtoC(f,temperatureModelRemote);

temperatureCommand\_processor.execute2(commandRemote);

                }

1. **Observer Pattern:**
   * **Các lớp liên quan:** Model, View.
   * **Mối quan hệ:**
     + Model: Là nhà cung cấp thông tin (Publisher ) cho Observer Pattern. Nó cập nhật trạng thái của nhiệt độ và thông báo cho các đối tượng đã đăng ký khi trạng thái thay đổi.Publisher là một phần của cơ chế thông báo, nơi nó chịu trách nhiệm gửi thông báo đến các subscribers (người đăng ký là view).
     + View: Là một trong các đối tượng đã đăng ký(subscribers ), nhận thông tin từ Model và cập nhật giao diện người dùng tương ứng.

**Ý nghĩa của Observer Pattern:** Cho phép View tự động cập nhật khi Model thay đổi, giúp đồng bộ hóa dữ liệu và giao diện người dùng.

1. **MVC Pattern:**
   * **Các lớp liên quan:** Model, View, Controller.
   * **Mối quan hệ:**
     + Model: Lưu trữ dữ liệu và quản lý logic nghiệp vụ, cập nhật trạng thái và thông báo cho TemperatureView.
     + View: Đại diện cho giao diện người dùng và hiển thị dữ liệu từ Model. Nó nhận yêu cầu từ người dùng và chuyển tiếp đến Controller để xử lý.
     + Controller: Xử lý các yêu cầu từ người dùng, tương tác với Model để thực hiện các chức năng nghiệp vụ và cập nhật View.

* **Ý nghĩa của** **MVC Pattern:** Phân tách rõ ràng các thành phần trong dự án (dữ liệu, giao diện, logic điều khiển), làm giảm sự phụ thuộc giữa chúng và dễ dàng bảo trì, mở rộng.

**Ý nghĩa của từng mẫu thiết kế trong dự án của ta:**

* **Command Pattern:** Giúp tách rời yêu cầu của người dùng thành các đối tượng lệnh cụ thể (ConvertCtoF, ConvertFtoC), dễ dàng mở rộng và tái sử dụng chúng.
* **Singleton Pattern:** Đảm bảo rằng chỉ có một đối tượng TemperatureCommand\_processor tồn tại, giúp quản lý và thực thi lệnh một cách hiệu quả.
* **Observer Pattern:** Cho phép TemperatureView tự động cập nhật khi TemperatureModel thay đổi, giúp đồng bộ hóa dữ liệu và giao diện người dùng.
* **MVC Pattern:** Phân tách rõ ràng các thành phần trong dự án (dữ liệu, giao diện, logic điều khiển), làm giảm sự phụ thuộc giữa chúng và dễ dàng bảo trì, mở rộng.

**## Đặc điểm mối quan hệ của từng class trong mẫu thiết kế của ứng dụng chuyển đổi nhiệt độ:**

**Lưu ý biến đối tượng đc lưu ở bộ nhớ heap nằm ở ram máy tính**

* + - **Controller** vs **View:** Có muốn quan hệ Association vì Controller và view không có tính thành phần vs nhau và tồn tại độc lập, Controller lưu trữ remote của View ở biến đối tượng View phải đc khởi tạo trước Controller và truyền remote của view vào cho controller.
    - **View** vs **Controller :** Có muốn quan hệ Association vì View và Controller không có tính thành phần vs nhau và tồn tại độc lập, View lưu trữ remote của Controller ở biến đối tượng Controller phải đc khởi tạo trước View và truyền remote của Conttroller vào cho View.
    - **Controller** vs **Model:** Có muốn quan hệ Association vì Controller và Model không có tính thành phần vs nhau và tồn tại độc lập, Controller lưu trữ remote của Model ở biến đối tượng Model phải đc khởi tạo trước Controller và truyền remote của model vào cho controller.
    - **View** vs **Model:** Có muốn quan hệ Association vì View và Model không có tính thành phần vs nhau và tồn tại độc lập, View lưu trữ remote của Model ở biến đối tượng Model phải đc khởi tạo trước View và truyền remote của model vào cho View.
    - **Command** vs **Model:** Có muốn quan hệ Association vì Command và Model không có tính thành phần vs nhau và tồn tại độc lập, Command lưu trữ remote của Model ở biến đối tượng, Model phải đc khởi tạo trước Command và truyền remote của model vào cho Command.
    - **Command\_processor** vs **Command:** Có muốn quan hệ depency

Vì Command\_processor lưu trữ ngắn hạn remote của command trong biến local stack

* + - Hai class **ConvertF2C** và **ConvertC2F** vs **Command:** Hai class ConvertF2Cvà ConvertC2F có mối quan hệ kế thừa với Command vì Hai class ConvertF2Cvà ConvertC2F sử dụng lại phương thức excute của lớp trừu tượng Command để triển khai hành động chuyển đổi nhiệt độ của các đại diện lệnh lớp Command
    - **Publisher** vs **Subscriber**: Có mối quan hệ không xác định
    - **View** vs **Subscriber:** Là view sử dụng và triển khai các phương thức của lớp đa hình **Subscriber**
    - **Model vs Publisher:** Model vs Publisher có mối quan hệ kế thừa Model sẽ kế thừa lại các phương thức và thuộc tính có sẵn của lớp cha Publisher