B The 1 Store

Course: CPIT-252
Supervisor: Dr. Rizwan Qureshi

Team Members

Section	ID	Name
IT 1	2135306	Essam Zare

Git Hub : https://github.com/EssamZarei/BThe1

Google Drive:
https://drive.google.com/drive/folders/1GH5svVKraqoryWI-nRq4k4E8KHRZO_ch?usp=sharing

Email:
EssamZarei@gmail.com
EAZarei@stu.kau.edu.sa

Table of Content

1. Introduction

This project implements a store management system designed to handle inventory, orders, and user customizations. By leveraging key design patterns—**Singleton**, **Observer**, and **Decorator**—along with the **MVC (Model-View-Controller)** architecture, the system achieves modularity, scalability, and maintainability. The store allows:

1. **Users** to create customized orders.
2. **Managers** to modify inventory.
3. **Synchronized database operations** to ensure consistent data access and integrity.

The integration of these patterns ensures the system is both robust and extensible, catering to real-world requirements.

2. Problem Definition

Managing a store that offers customizable products requires addressing several challenges:

1. **Centralized and Efficient Database Management**:
   - Prevent resource conflicts from multiple database connections.
   - Ensure consistent and synchronized data updates.
2. **Role-Based Operations**:
   - Managers need tools to dynamically modify the inventory.
   - Users need to interact with the store seamlessly.
3. **Dynamic Pricing and Notifications**:
   - Orders involve dynamic updates as users add or remove items.
   - Real-time feedback is essential for a good user experience.
4. **Product Customization**:
   - Users must have flexibility in adding custom features to products, such as names, logos, or characters.
5. **Maintainable and Scalable Architecture**:
   - Clear separation of concerns to simplify development and future enhancements.
   -

3. Suggested Solution

To address these challenges, the system combines the **Singleton**, **Observer**, and **Decorator** patterns with **MVC architecture**:

Singleton Pattern:

- Ensures a single, centralized database connection (`DBConnection` class).
- Optimizes resource usage and guarantees consistent data handling.

Observer Pattern:

- Keeps track of real-time changes, such as order totals, and notifies observers dynamically.

Decorator Pattern:

- Enables dynamic customization of drawings, allowing users to stack features like names, logos, and characters.

MVC Architecture:

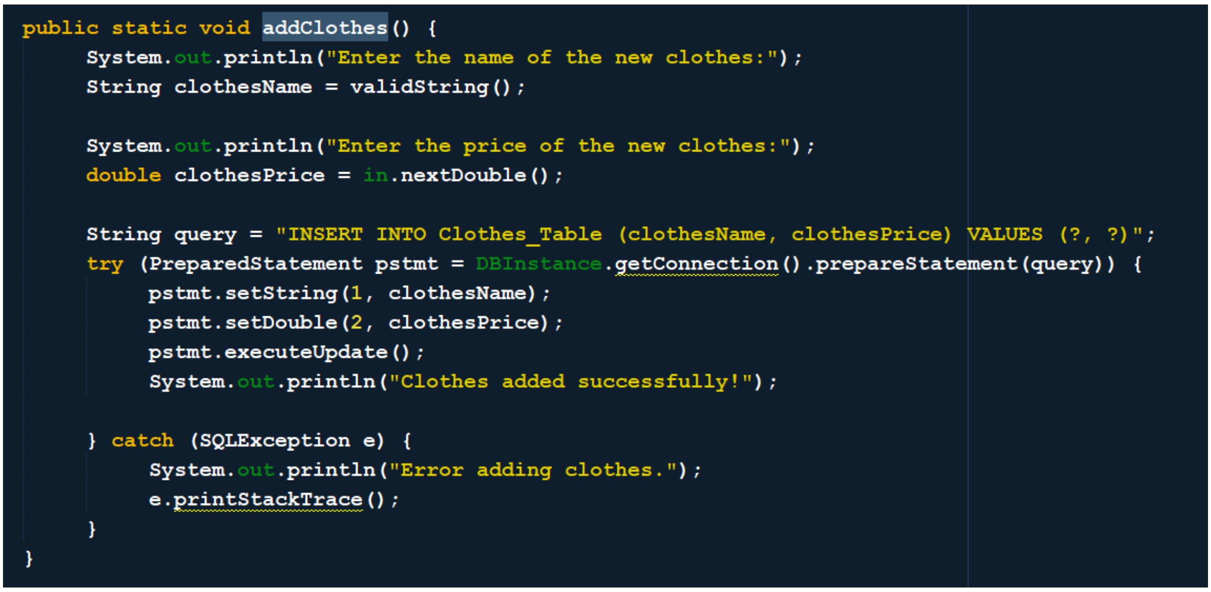
- Divides the system into:
  - **Model**: Handles database operations and business logic.
  - **View**: Manages user input and output.
  - **Controller**: Coordinates interactions between the Model and View.

4. Store Functions

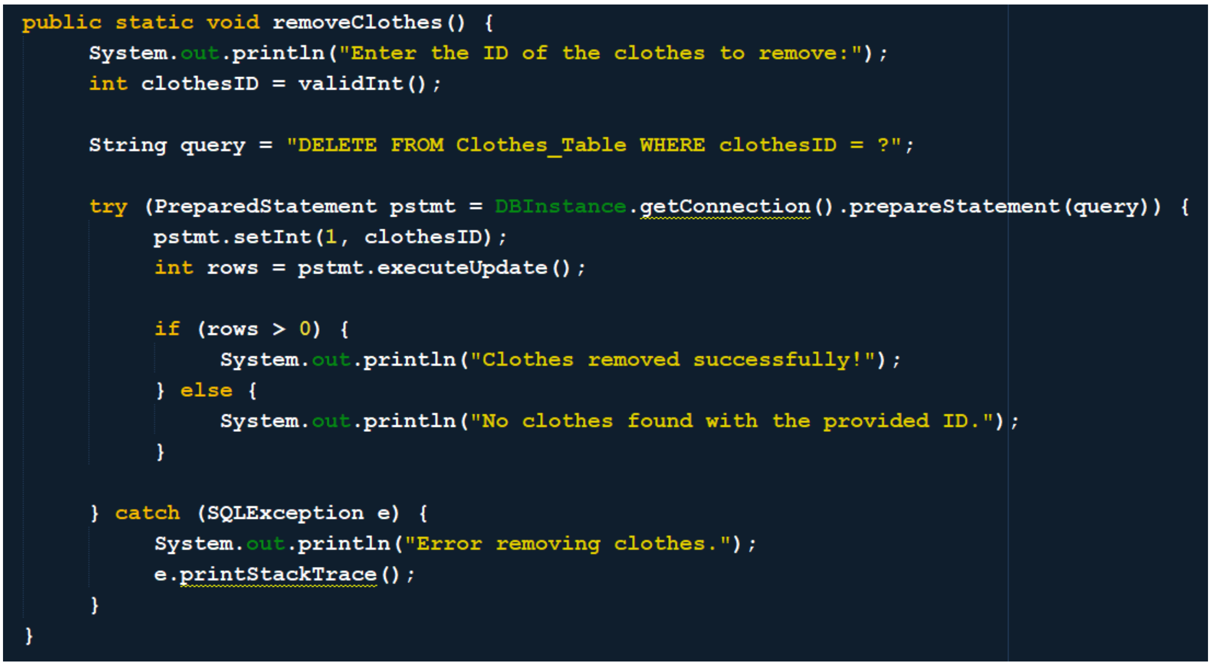
4.1 Inventory Management (Manager Role)

Managers can:

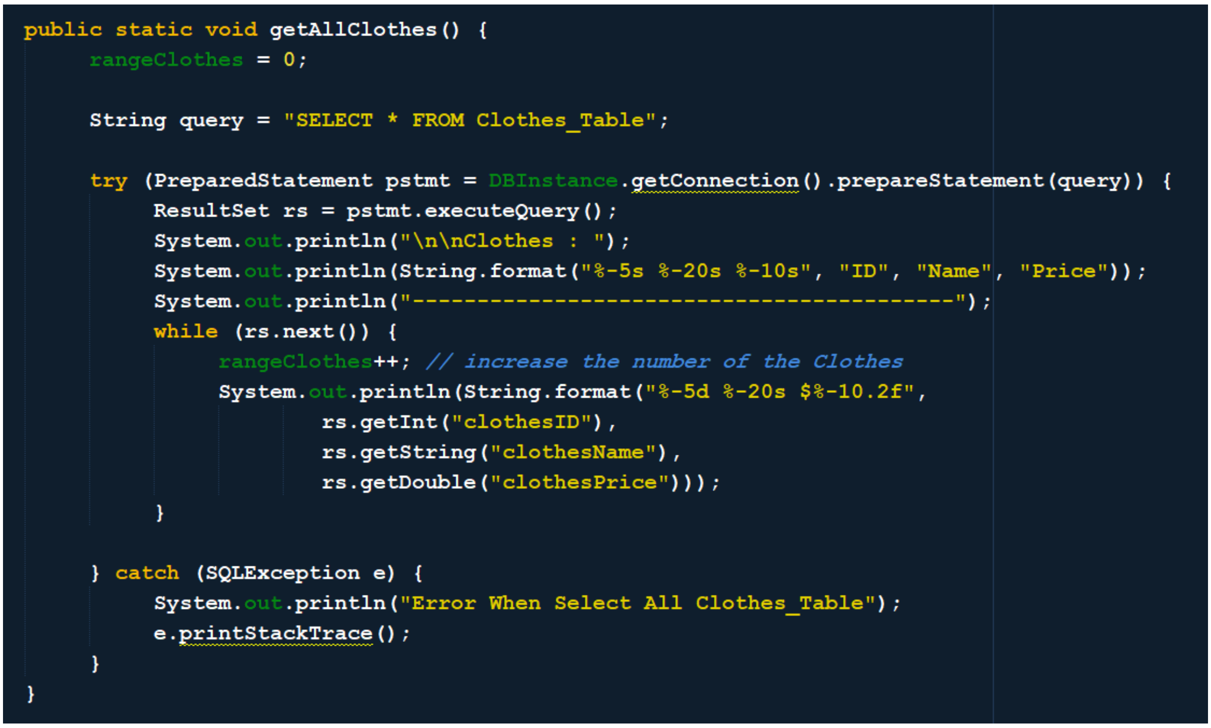
1. **Add Items**: Clothes, colors, and drawings can be added dynamically.
2. **Remove Items**: Inventory items can be removed using their unique IDs.
3. **View Inventory**: The current inventory is retrieved from the database and displayed in real-time.



```
public static void addClothes() {
     System.out.println("Enter the name of the new clothes:");
     String clothesName = validString();

     System.out.println("Enter the price of the new clothes:");
     double clothesPrice = in.nextDouble();

     String query = "INSERT INTO Clothes_Table (clothesName, clothesPrice) VALUES (?, ?)";
     try (PreparedStatement pstmt = DBInstance.getConnection().prepareStatement(query)) {
          pstmt.setString(1, clothesName);
          pstmt.setDouble(2, clothesPrice);
          pstmt.executeUpdate();
          System.out.println("Clothes added successfully!");

     } catch (SQLException e) {
          System.out.println("Error adding clothes.");
          e.printStackTrace();
     }
}
```



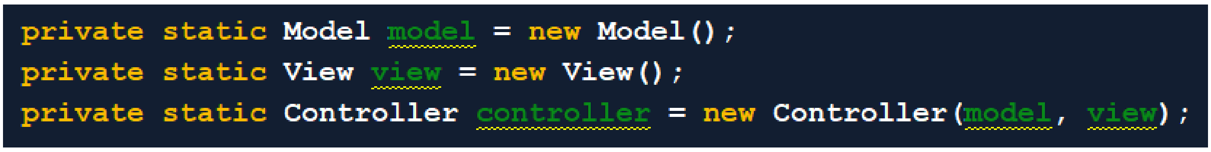
```java
public static void removeClothes() {
    System.out.println("Enter the ID of the clothes to remove:");
    int clothesID = validInt();

    String query = "DELETE FROM Clothes_Table WHERE clothesID = ?";

    try (PreparedStatement pstmt = DBInstance.getConnection().prepareStatement(query)) {
        pstmt.setInt(1, clothesID);
        int rows = pstmt.executeUpdate();

        if (rows > 0) {
            System.out.println("Clothes removed successfully!");
        } else {
            System.out.println("No clothes found with the provided ID.");
        }

    } catch (SQLException e) {
        System.out.println("Error removing clothes.");
        e.printStackTrace();
    }
}
```



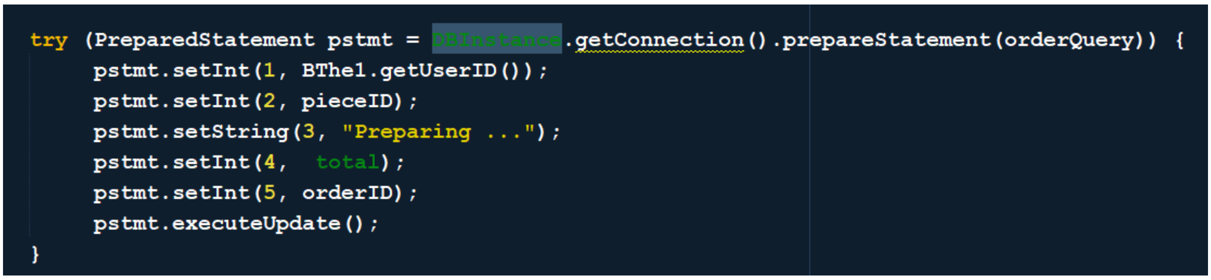
```java
public static void getAllClothes() {
    rangeClothes = 0;

    String query = "SELECT * FROM Clothes_Table";

    try (PreparedStatement pstmt = DBInstance.getConnection().prepareStatement(query)) {
        ResultSet rs = pstmt.executeQuery();
        System.out.println("\n\nClothes : ");
        System.out.println(String.format("%-5s %-20s %-10s", "ID", "Name", "Price"));
        System.out.println("------------------------------------------");
        while (rs.next()) {
            rangeClothes++; // increase the number of the Clothes
            System.out.println(String.format("%-5d %-20s $%-10.2f",
                    rs.getInt("clothesID"),
                    rs.getString("clothesName"),
                    rs.getDouble("clothesPrice")));
        }

    } catch (SQLException e) {
        System.out.println("Error When Select All Clothes_Table");
        e.printStackTrace();
    }
}
```

How MVC Works Here:

- **Model**: The `DBConnection` Singleton handles all database queries.
- **View**: Displays inventory prompts and operation results (e.g., success messages).
- **Controller**: Processes user inputs and calls the Model to update the database



```
private static Model model = new Model();
private static View view = new View();
private static Controller controller = new Controller(model, view);
```

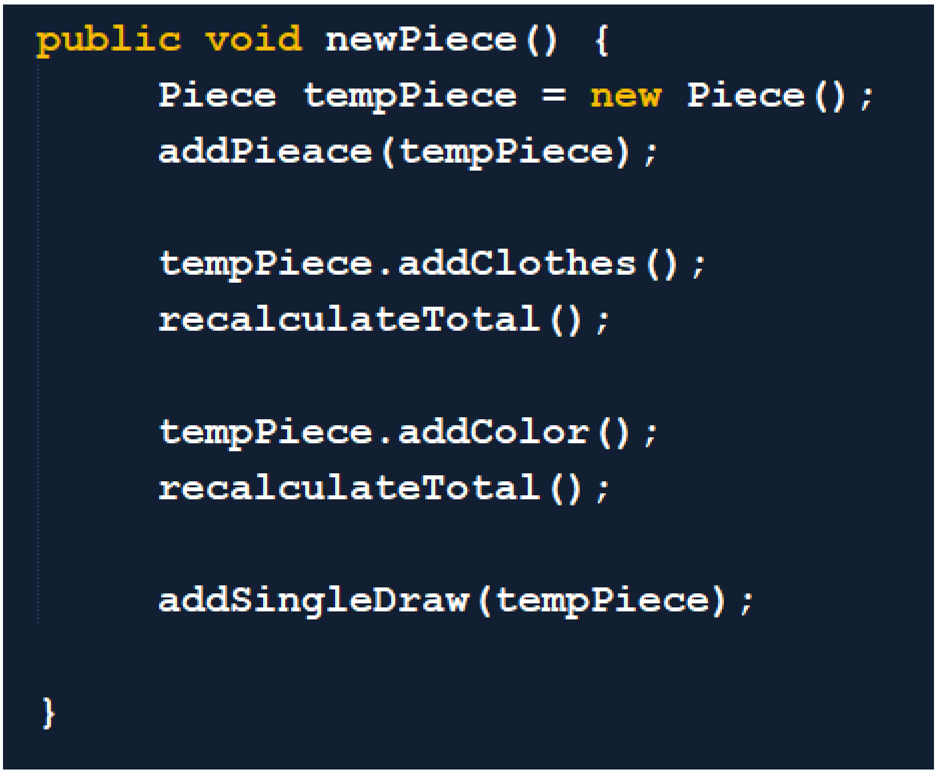


```
try (PreparedStatement pstmt = DBInstance.getConnection().prepareStatement(orderQuery)) {
        pstmt.setInt(1, BThe1.getUserID());
        pstmt.setInt(2, pieceID);
        pstmt.setString(3, "Preparing ...");
        pstmt.setInt(4,  total);
        pstmt.setInt(5, orderID);
        pstmt.executeUpdate();
}
```

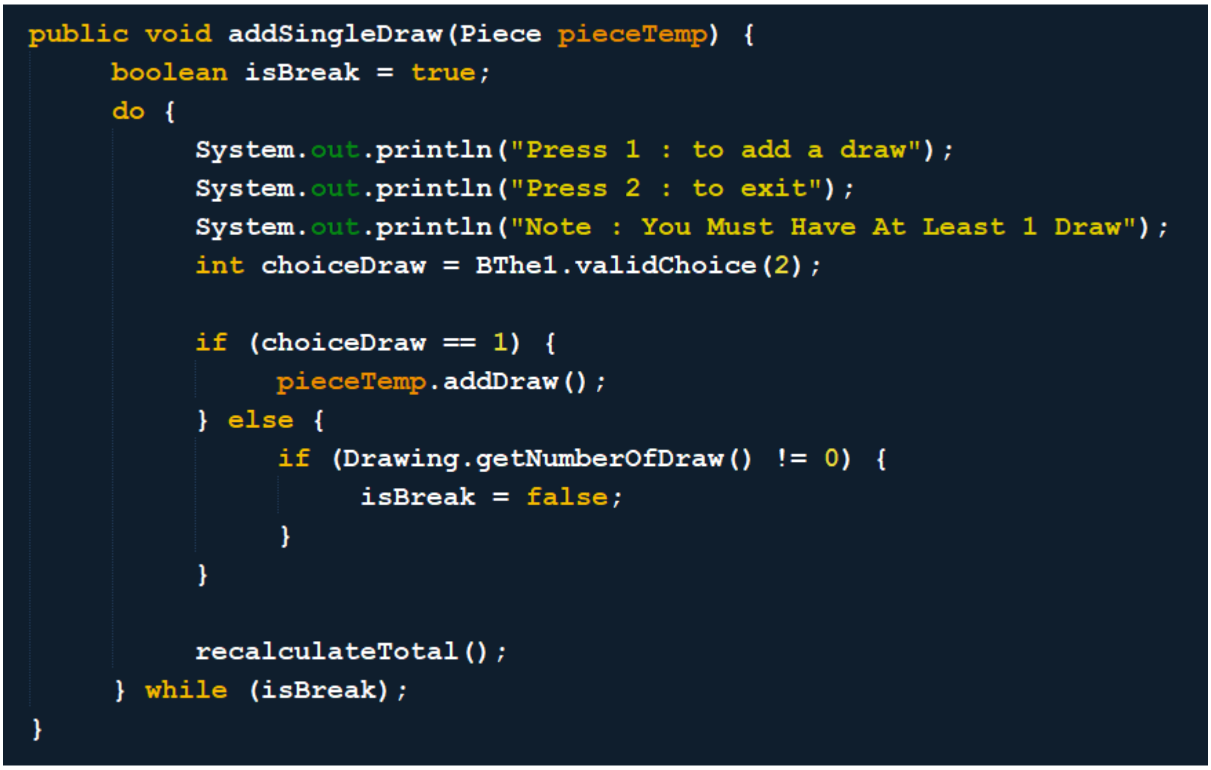
4.2 Order Management (User Role)

Users can:

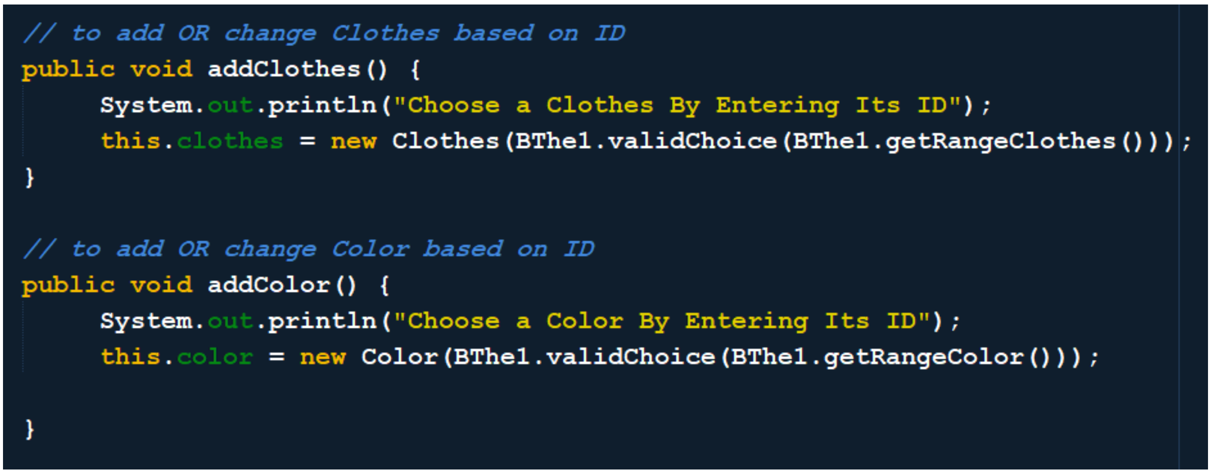
1. **Create New Orders**:
   - Add pieces consisting of clothes, colors, and drawings.
   - Customize drawings dynamically using the **Decorator Pattern**.



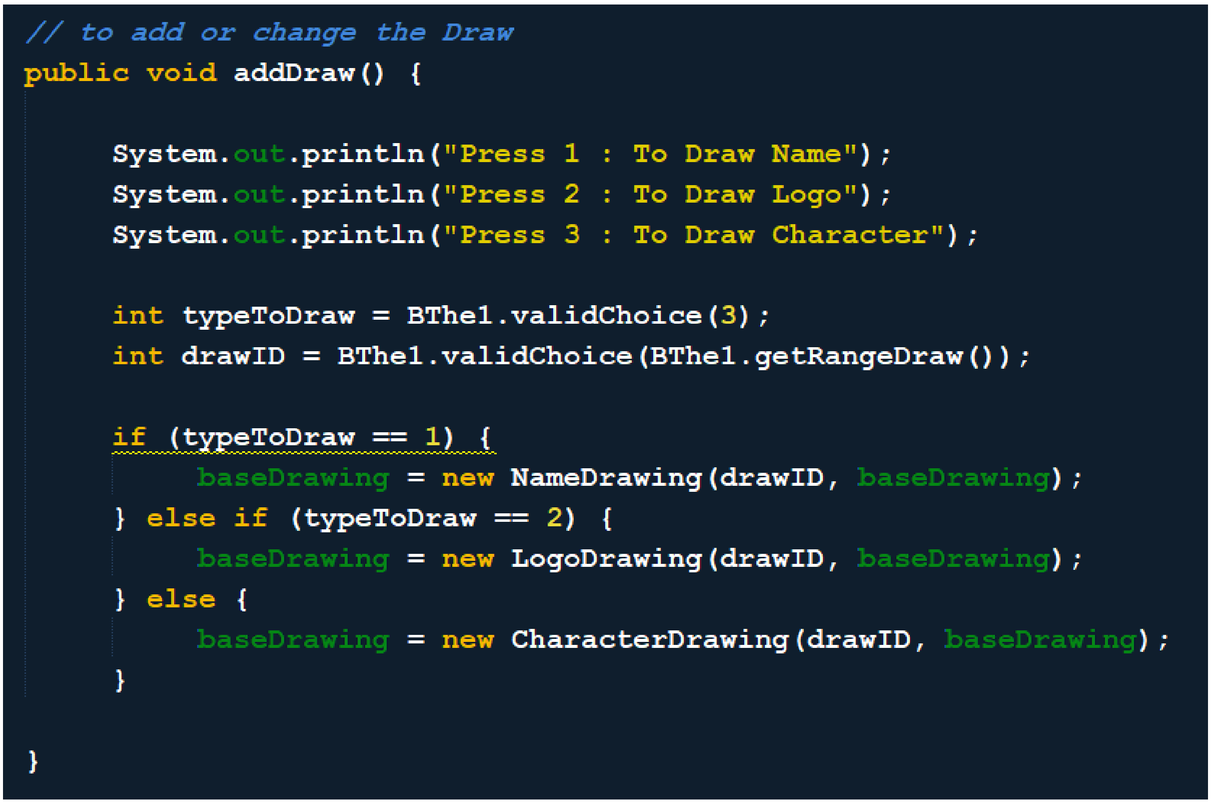
```
public void newPiece() {
        Piece tempPiece = new Piece();
        addPieace(tempPiece);

        tempPiece.addClothes();
        recalculateTotal();

        tempPiece.addColor();
        recalculateTotal();

        addSingleDraw(tempPiece);

}
```



```java
public void addSingleDraw(Piece pieceTemp) {
    boolean isBreak = true;
    do {
        System.out.println("Press 1 : to add a draw");
        System.out.println("Press 2 : to exit");
        System.out.println("Note : You Must Have At Least 1 Draw");
        int choiceDraw = BThe1.validChoice(2);

        if (choiceDraw == 1) {
            pieceTemp.addDraw();
        } else {
            if (Drawing.getNumberOfDraw() != 0) {
                isBreak = false;
            }
        }

        recalculateTotal();
    } while (isBreak);
}
```



```java
// to add OR change Clothes based on ID
public void addClothes() {
    System.out.println("Choose a Clothes By Entering Its ID");
    this.clothes = new Clothes(BThe1.validChoice(BThe1.getRangeClothes()));
}

// to add OR change Color based on ID
public void addColor() {
    System.out.println("Choose a Color By Entering Its ID");
    this.color = new Color(BThe1.validChoice(BThe1.getRangeColor()));

}
```



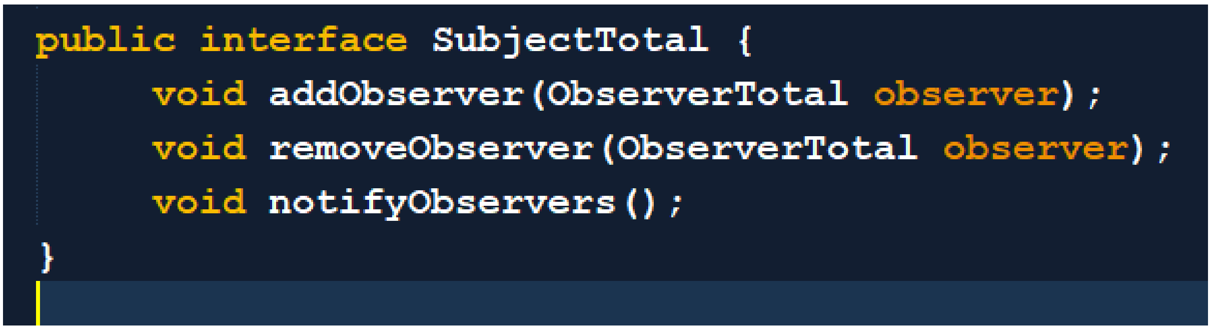
```
// to add or change the Draw
public void addDraw() {

    System.out.println("Press 1 : To Draw Name");
    System.out.println("Press 2 : To Draw Logo");
    System.out.println("Press 3 : To Draw Character");

    int typeToDraw = BThe1.validChoice(3);
    int drawID = BThe1.validChoice(BThe1.getRangeDraw());

    if (typeToDraw == 1) {
        baseDrawing = new NameDrawing(drawID, baseDrawing);
    } else if (typeToDraw == 2) {
        baseDrawing = new LogoDrawing(drawID, baseDrawing);
    } else {
        baseDrawing = new CharacterDrawing(drawID, baseDrawing);
    }

}
```

2. **Modify or Remove Pieces**:
    - Users can update attributes like clothes or colors or remove items entirely.
3. **Dynamic Pricing**:
    - The **Observer Pattern** ensures prices are recalculated and displayed in real-time as users modify their orders.

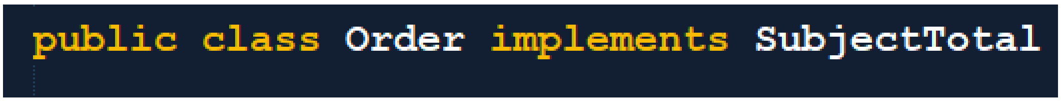
How MVC Works Here:

- **Model**: Handles data storage (e.g., orders and pieces) and retrieval using the Singleton database connection.
- **View**: Displays inventory, allows users to make selections, and shows updated totals.
- **Controller**: Coordinates user actions (e.g., adding a piece) with database operations and price updates.

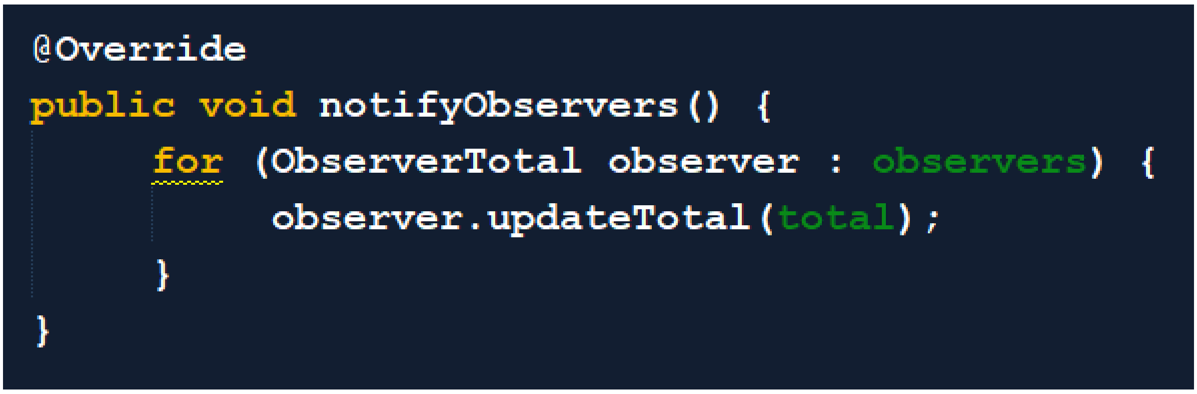
Example of Dynamic Pricing:



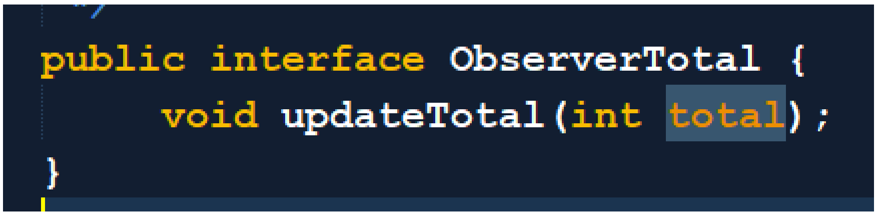
```java
public interface SubjectTotal {
	void addObserver(ObserverTotal observer);
	void removeObserver(ObserverTotal observer);
	void notifyObservers();
}
```



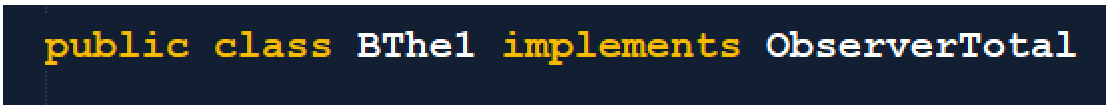
```java
public class Order implements SubjectTotal
```



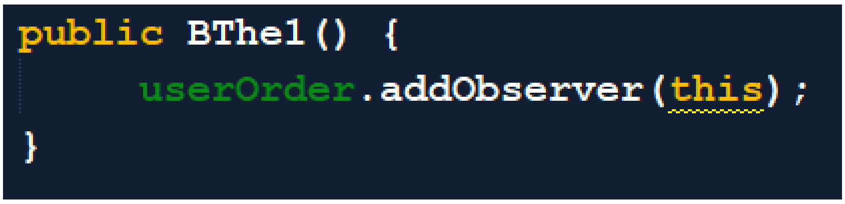
```java
@Override
public void notifyObservers() {
	for (ObserverTotal observer : observers) {
		observer.updateTotal(total);
	}
}
```



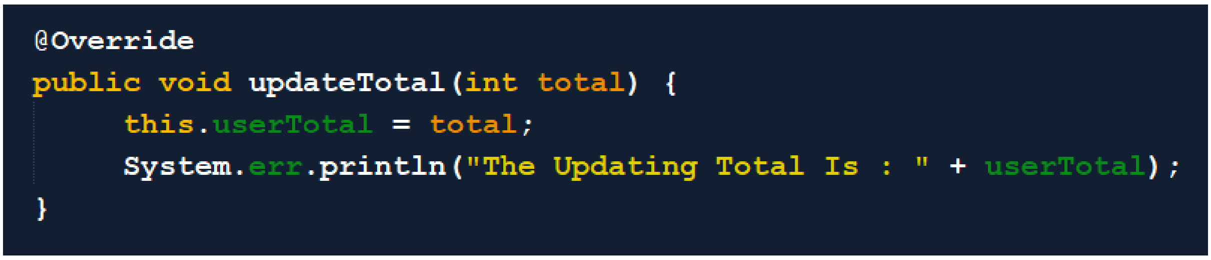
```java
public interface ObserverTotal {
	void updateTotal(int total);
}
```



```java
public class BThe1 implements ObserverTotal
```



```java
public BThe1() {
	userOrder.addObserver(this);
}
```



```java
@Override
public void updateTotal(int total) {
	this.userTotal = total;
	System.err.println("The Updating Total Is : " + userTotal);
}
```

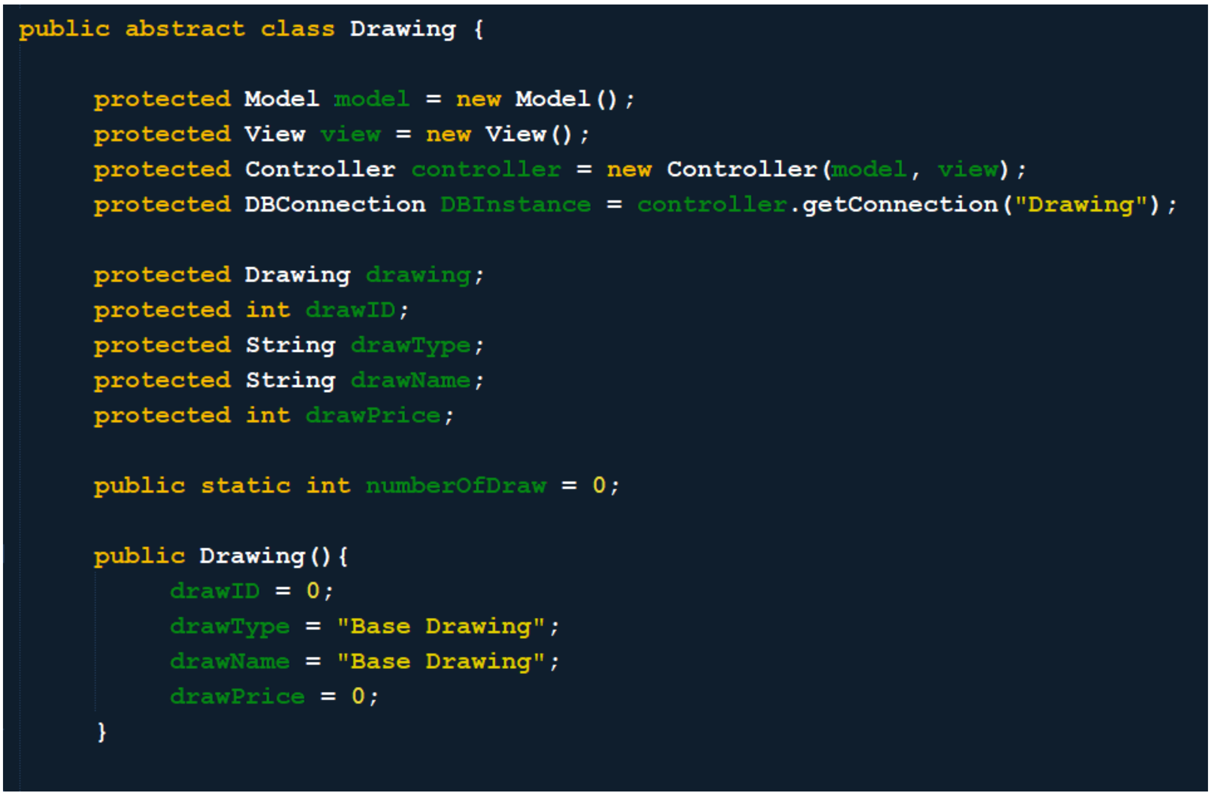
4.3 Drawing Customization

Users can customize their products by adding drawings like:

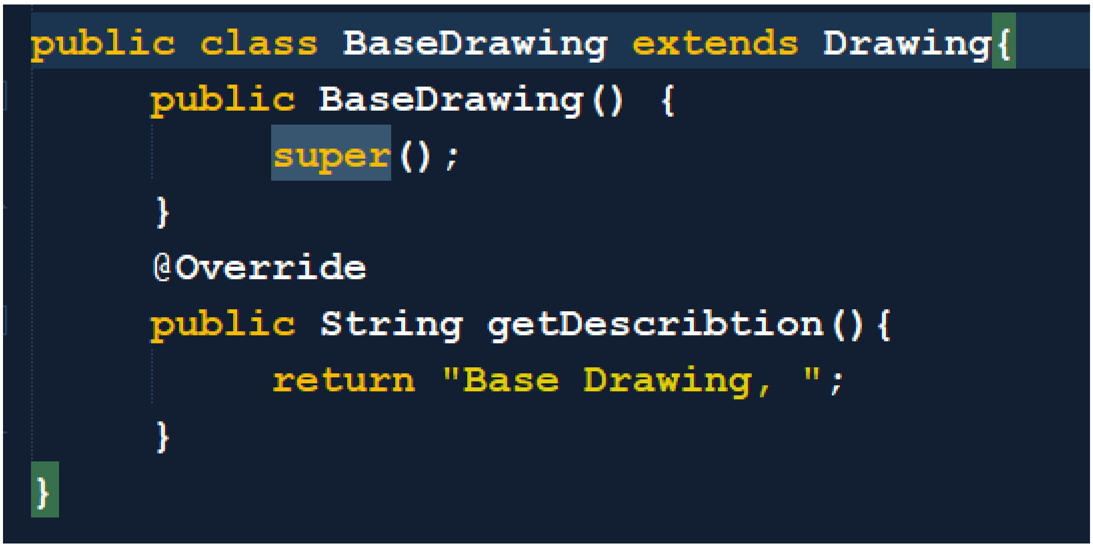
1. **BaseDrawing: The default drawing layer.**
2. **NameDrawing: Adds a name or text to the item.**
3. **LogoDrawing: Includes a logo design.**
4. **CharacterDrawing: Adds a character illustration.**

These customizations use the Decorator Pattern, enabling layers of features to be dynamically added.

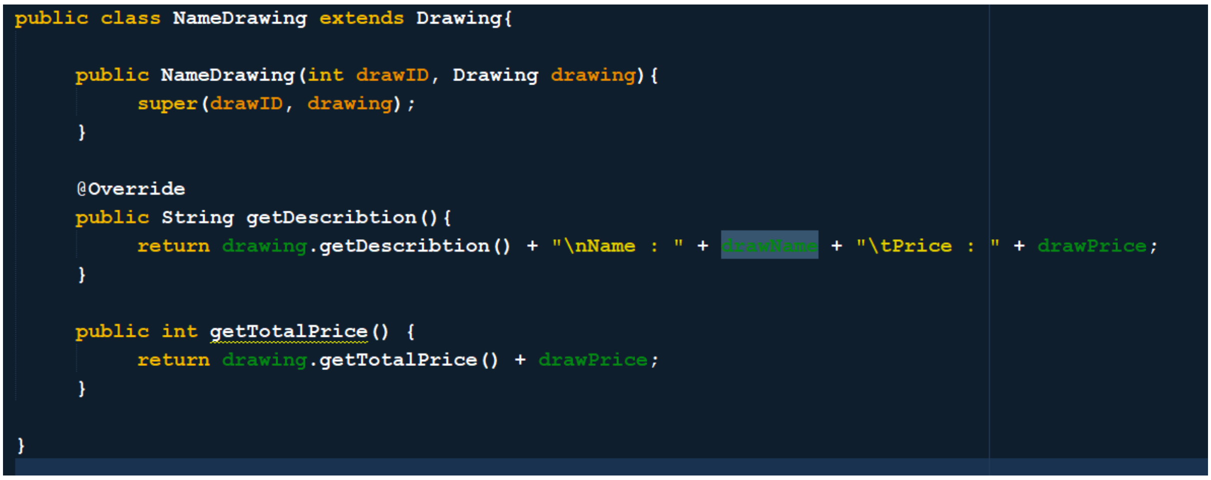
Example: Combining multiple drawings:



```
public abstract class Drawing {

    protected Model model = new Model();
    protected View view = new View();
    protected Controller controller = new Controller(model, view);
    protected DBConnection DBInstance = controller.getConnection("Drawing");

    protected Drawing drawing;
    protected int drawID;
    protected String drawType;
    protected String drawName;
    protected int drawPrice;

    public static int numberOfDraw = 0;

    public Drawing(){
        drawID = 0;
        drawType = "Base Drawing";
        drawName = "Base Drawing";
        drawPrice = 0;
    }
```



```
public class BaseDrawing extends Drawing{
        public BaseDrawing()  {
                super();
        }
        @Override
        public String getDescribtion(){
                return "Base Drawing, ";
        }
}
```



```
public class NameDrawing extends Drawing{

    public NameDrawing(int drawID, Drawing drawing){
        super(drawID, drawing);
    }

    @Override
    public String getDescribtion(){
        return drawing.getDescribtion() + "\nName : " + drawName + "\tPrice : " + drawPrice;
    }

    public int getTotalPrice()  {
        return drawing.getTotalPrice() + drawPrice;
    }

}
```

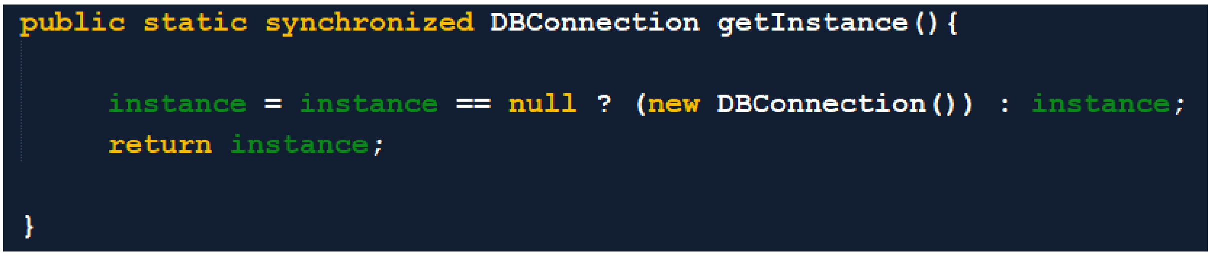
Logo and Character same as Naming

5. Database Integration

Singleton Database Connection

The `DBConnection` class ensures:

1. **Single Connection Instance**:
   - Prevents resource contention and ensures consistent data access.
2. **Thread-Safe Access**:
   - The `getInstance` method ensures only one connection exists at any time:



```
public static synchronized DBConnection getInstance(){

    instance = instance == null ? (new DBConnection()) : instance;
    return instance;

}
```

Synchronized Data Flow

The database handles:

1. **Inventory Tables**:
   - Clothes, colors, and drawings are stored in their respective tables.
2. **Orders and Pieces**:
   - Orders link to pieces, which track selected clothes, colors, and drawings.

How Data Flow Works:

1. The **View** prompts the user (e.g., "Add new clothes" the connection will be established by the class itself).
2. The **Controller** calls the appropriate method in the **Model**.
3. The **Model** return Singleton database connection.

6. Why Each Pattern Matters

Singleton Pattern:

- Centralizes database management.
- Ensures efficient and consistent access across multiple operations.

Observer Pattern:

- Provides real-time updates for order totals.
- Ensures dynamic responsiveness without tightly coupling components.

Decorator Pattern:

- Enables dynamic and extensible product customization.
- Allows features to be added or removed without altering the base structure.

MVC Architecture:

- Separates concerns for better maintainability.
- Simplifies testing and debugging by isolating layers.

7. Class Diagram

Purpose of the Class Diagram

1. **Understand the Static Structure**:
   - Show classes, their attributes, methods, and relationships.
2. **Highlight Design Patterns**:
   - Illustrate the use of **MVC**, **Singleton**, **Decorator**, and **Observer** patterns in the system.
3. **Document Interactions**:
   - Depict how different components like `Order`, `Piece`, `Drawing`, and `DBConnection` interact.

Design Patterns in the Class Diagram

1. MVC (Model-View-Controller)

The **MVC architecture** separates the application into:

- **Model**:
  - Contains the business logic and handles interactions with the database through the `DBConnection` Singleton.
  - Example: `Model` manages `Order`, `Piece`, `Clothes`, and `Color`.
- **View**:
  - Handles user input and displays data to the user.
  - Example: The `View` class prompts user actions and displays inventory, orders, and notifications.
- **Controller**:
  - Acts as a mediator between the Model and View, orchestrating data flow.
  - Example: `Controller` processes user actions like adding or modifying pieces and updates the View.

Relationships:

- `Controller` connects `Model` and `View`.
- `Model` uses the `DBConnection` Singleton for database operations.

2. Singleton

The **Singleton Pattern** ensures a single instance of the database connection:

- Class: `DBConnection`
- Responsibilities:
  - Provides a global point of access to the database.
  - Ensures thread-safe and resource-efficient database operations.

Implementation:

- `getInstance()` method ensures only one `DBConnection` object exists.
- Used across `Clothes`, `Color`, `Drawing`, `Piece`, `Order`, and `Controller` classes.

Relationships:

- `DBConnection` is used by all Model classes for database operations.

3. Decorator

The **Decorator Pattern** is used for dynamic drawing customizations:

- **Base Class**: `Drawing`
- **Decorators**:
  - `NameDrawing`: Adds names to drawings.
  - `LogoDrawing`: Adds logos to drawings.
  - `CharacterDrawing`: Adds character illustrations.
- **Dynamic Behavior**:
  - Each decorator wraps another `Drawing` object, adding new functionality without modifying the base class.

Relationships:

- `Drawing` is the base class.
- `NameDrawing`, `LogoDrawing`, and `CharacterDrawing` inherit from `Drawing` and wrap other `Drawing` objects.

4. Observer

The **Observer Pattern** handles real-time notifications of order total changes:

- **Subject**: `Order`
  - Maintains a list of observers (e.g., `BThe1`).
  - Notifies observers of total changes via the `notifyObservers` method.
- **Observer**: `ObserverTotal`
  - Implemented by `BThe1`, which updates the user interface when notified.

Relationships:

- `Order` (Subject) has a one-to-many relationship with `ObserverTotal` (Observer).
- Observers listen for updates from the `Order` class.

Key Components in the Class Diagram

1. **Core Entities**:
   - `Clothes`, `Color`, `Drawing`, `Piece`, and `Order`.
2. **Design Patterns**:
   - **MVC**: `Model`, `View`, and `Controller`.
   - **Singleton**: `DBConnection`.
   - **Decorator**: `BaseDrawing` and its decorators (`NameDrawing`, `LogoDrawing`, `CharacterDrawing`).
   - **Observer**: `Order` (Subject) and `ObserverTotal` (Observer).
3. **Relationships**:
   - `Controller` connects `Model` and `View`.
   - `Model` uses `DBConnection`.
   - `Drawing` decorators wrap the base `Drawing` class.
   - `Order` notifies observers of changes.

Class Diagram PDF:
https://drive.google.com/drive/folders/1GH5svVKraqoryWI-nRq4k4E8KHRZO_ch?usp=sharing

8. ER Diagram

Purpose:

The ER diagram is a database design tool that visually represents entities (tables), their attributes (columns), and the relationships between them. It helps in understanding:

1. **Database Schema**: How data is organized.
2. **Relationships**: Cardinalities and dependencies between tables.

ER Diagram for BThe1 Store Database

The ER diagram for the database illustrates:

1. **Entities**:
    - `User_Table`: Tracks users and their roles (admin or regular user).
    - `Colors_Table`, `Clothes_Table`, `Draw_Table`: Store inventory details.
    - `Piece_Table` and `Order_Table`: Manage user orders and associated pieces.
2. **Relationships**:
    - `Order_Table` references `User_Table` (foreign key) and `Piece_Table`.
    - `Piece_Table` links to `Colors_Table`, `Clothes_Table`, and `Draw_Table`.
    - `Piece_Draw_Table` establishes a many-to-many relationship between pieces and drawings.

ER Diagram PDF:
https://drive.google.com/drive/folders/1GH5svVKraqoryWI-nRq4k4E8KHRZO_ch?usp=sharing

9. Expected Results

By combining Singleton, Observer, Decorator, and MVC, the system achieves:

1. **Efficiency**: Centralized and synchronized database operations.
2. **Dynamic Features**: Real-time updates and flexible product customizations.
3. **Scalability**: A modular design that supports future extensions.
4. **User Satisfaction**: Intuitive interfaces and seamless workflows.