Coffee Shop System Documentation

This system models a simple coffee shop where customers can select drinks, customize their orders with syrup, and choose cup sizes. The project demonstrates several design patterns, including **Decorator**, **Factory Method**, and **Observer**.

System Overview

The coffee shop system provides a set of drink options such as Americano, Caffè Latte, Cappuccino, and Espresso. Customers can customize these drinks by adding syrup, and the amount of syrup is constrained by the cup size. Additionally, the system alerts the user when a cup is full or overflowing, leveraging the Observer pattern.

Design Patterns

1. **Decorator Pattern**

The **Decorator** pattern is used to dynamically add functionality to an object without altering its structure. This pattern is demonstrated in the `ItemDecorator` class and its subclass `Syrup`. The `ItemDecorator` extends the functionality of the base `Item` interface by adding behavior to the `getName()`, `getPrice()`, and `getAmount()` methods.

- **ItemDecorator**: This abstract class implements the `Item` interface and holds a reference to an `Item` object. It delegates method calls to the wrapped item, enabling subclasses to add additional behavior.
- **Syrup**: A concrete decorator that adds syrup to a drink. It changes the name and price of the drink, and specifies the syrup amount.

```
public abstract class ItemDecorator implements Item { ... }

public class Syrup extends ItemDecorator { ... }
```

2. **Factory Method Pattern**

The **Factory Method** pattern provides an interface for creating objects in a super class but allows subclasses to alter the type of objects that will be created. This is demonstrated by the `CupFactory` class, which creates instances of the `Cup` class based on the size requested.

- **CupFactory**: This class provides a static method `createCup()` to create different sizes of cups (normal, grande, venti). It abstracts the process of creating cups, allowing easy expansion if more sizes are added in the future.

```
public class CupFactory {
    public static Cup createCup(String size) { ... }
}
```

3. **Observer Pattern**

The **Observer** pattern allows an object to notify other objects about changes in its state. In this project, the `AlarmObserver` and `CupObserver` interfaces implement the Observer pattern to monitor cup fullness and overflow.

- **CupObserver**: An interface that defines two methods for observing when a cup is full or overflowing.
- **AlarmObserver**: A concrete observer that reacts to the events and prints a warning message when the cup is full or overflowing.

```
public interface CupObserver {
    void onCupFull(Cup cup);
    void onCupOverflow(Cup cup);
}

public class AlarmObserver implements CupObserver {
    @Override public void onCupFull(Cup cup) { ... }
    @Override public void onCupOverflow(Cup cup) { ... }
}
```

Key Components

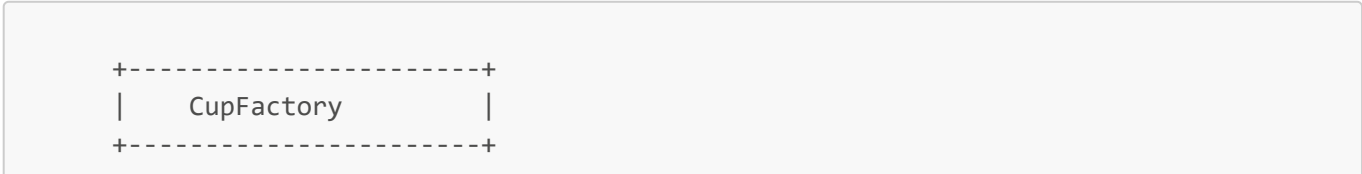
- **Items (Drinks)**: Each drink (e.g., `Americano`, `Cappuccino`) implements the `Item` interface, which defines methods like `getName()`, `getPrice()`, and `getAmount()`. Drinks have different base prices and amounts.
- **Cup**: A `Cup` can hold a specific amount of drink and syrup. The cup's size is fixed, and it has methods to add syrup and drink while checking for overflow.
- **Decorators**: The `Syrup` decorator adds syrup to a drink, modifying its price and name.

System Flow

1. The user chooses a cup size (e.g., `grande`), which is created using the `CupFactory`.
2. The user selects a drink, which is instantiated through a `Drink` implementation (e.g., `Americano`).
3. The user can then add syrup to the drink by wrapping the drink object with a `Syrup` decorator.
4. If the cup reaches its capacity, the `CupObserver` triggers the `AlarmObserver`, notifying the user of a full or overflowing cup.

UML diagram

Factory method



```
| +createCup(type): Cup |
+-----------------------+
        |
        v
+--------------------+
|      Cup           |
+--------------------+
| -type: String      |
| -size: int         |
+--------------------+
| +getType(): String |
| +getSize(): int    |
+--------------------+
```

Decorator

```
+---------------------------+
|      Item                 |
+---------------------------+
| +getDescription(): String |
| +cost(): double           |
+---------------------------+
          ^
          |
+---------------------------+
|  ItemDecorator            |
+---------------------------+
| -item: Item               |
+---------------------------+
| +getDescription(): String |
| +cost(): double           |
+---------------------------+
          ^
          |
+---------------------------+
|     Syrup                 |
+---------------------------+
| -syrupCost: double        |
+---------------------------+
| +getDescription(): String |
| +cost(): double           |
+---------------------------+
```

Observer

```
+---------------------------+
|      CupObserver          |
+---------------------------+
| +onCupFull(Cup): void     |
```

```
| +onCupOverflow(Cup): void |
+---------------------------+
          ^
          |
+---------------------------+
|  AlarmObserver            |
+---------------------------+
| +onCupFull(Cup): void     |
| +onCupOverflow(Cup): void |
+---------------------------+
```

Cup class

```
+---------------------------------------------+
|      Cup                                    |
+---------------------------------------------+
| -observers: List<CupObserver>               |
+---------------------------------------------+
| +addObserver(observer: CupObserver): void   |
| +removeObserver(observer: CupObserver): void |
| +notifyObservers(): void                    |
+---------------------------------------------+
```

Conclusion

This coffee shop system effectively demonstrates three major design patterns that provide flexibility and extensibility:

- The **Decorator** pattern allows us to add custom features (like syrup) to drinks.
- The **Factory Method** pattern simplifies the creation of cup objects based on size.
- The **Observer** pattern ensures that the system responds to changes in the cup's state, notifying users of overflows or fullness.

The system's modular design makes it easy to extend (e.g., by adding new drink types or decorators), and it ensures that functionality is neatly separated, promoting maintainability.