**Before Code:** [**Previous Code**](https://github.com/RatanJN/665_Assignment1.git)

**After Code:** [**New Code Link**](https://github.com/RatanJN/665_Assignment6.git)

**The main Classes changed are:**

1. CondimentStrategy Interface
2. Milk Class
3. Sugar Class
4. BeverageMachine Class

**1. Examine your code and identify opportunities for code improvement:**

**Opportunities for Improvement and Reasoning:**

1. **Singleton Pattern Usage**: Initially, the Singleton pattern was used for **Milk** and **Sugar** classes to ensure a single instance, which can help manage state consistently across the application. However, this approach is not entirely suited for cases where the state (like the count of condiments) must be reset frequently or is specific to individual drink preparations. This can cause complications if multiple drinks are prepared in sequence, as the state is not isolated to a single drink.
2. **Interface Completeness**: The **CondimentStrategy** lacked a **resetCondimentCount()** method, which is crucial for resetting state in Singleton implementations, ensuring that each drink preparation starts with a fresh state.
3. **Enhancing Encapsulation and Cohesion**: The original implementation had the **BeverageMachine** not fully encapsulating the responsibility of managing the lifecycle of condiments, particularly in resetting their state after drink preparation. This led to potential external manipulation of condiment state, which should ideally be managed internally by the machine class.

**Reasoning Behind Improvements:**

* **Singleton Pattern Reevaluation**: Reevaluating the Singleton pattern usage ensures that each drink can be prepared with a fresh state, avoiding unintended side effects of shared state across multiple operations. This is crucial in a real-world scenario where machines need to prepare multiple orders successively.
* **Improving Interface Completeness**: Ensuring the interface includes all necessary methods for managing condiments promotes better abstraction and encapsulation. It allows for consistent and predictable implementations across different types of condiments.
* **Encapsulation and Cohesion**: By enhancing the encapsulation within the **BeverageMachine**, all aspects of beverage preparation, including condiment management, are centralized, reducing the system's complexity and improving maintainability.

**2. Describe the changes made to the code**

**Detailed Changes:**

1. **Introduction of resetCondimentCount to CondimentStrategy**:
   * **Change**: Added **resetCondimentCount()** method to the **CondimentStrategy** interface and implemented it in **Milk** and **Sugar**.
   * **Purpose**: To ensure that each condiment strategy can reset its count after a drink is prepared, which is particularly important when using Singleton instances.
2. **Condiment Management in BeverageMachine**:
   * **Change**: Integrated a mechanism in **BeverageMachine** to track used condiments during a drink's preparation and to reset only those condiments that were used.
   * **Purpose**: Optimizes performance by resetting only the necessary condiments and encapsulates this functionality within the machine, adhering to the principle of single responsibility and encapsulation.
3. **Reevaluation of Singleton Pattern**:
   * **Change**: Maintained the Singleton pattern but ensured each use case (like preparing a drink) resets the state appropriately via the **resetCondimentCount()** method, making Singleton more suitable for managing shared resources like condiments in a controlled manner.
   * **Purpose**: Provides a consistent state management approach while ensuring each drink preparation starts with a clean state, essential for accurate operation in a commercial setting.

**To See the UML before Modification open Assignment1.html and the one after open Assignment1Modify.html**