Report on Java Software Design Patterns: Singleton & Strategy Patterns

---

Objective:
The primary goal of this project is to demonstrate the Singleton and Strategy patterns to achieve code reusability, maintainability, and flexibility in software development.

---

1. Singleton Design Pattern:

Definition:
Ensures that a class has only one instance and provides a global point of access to that instance.

Implementation:
The Singleton pattern was implemented using two methods:

- **Standard Singleton**:
  Found in the `Singleton` class, it ensures that only one instance of this class is created during the program's lifetime.

```
public class Singleton {
    private static Singleton uniqueInstance;
    private Singleton() {}
    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }
}
```

- **Thread-Safe Singleton**:
  This is an enhanced version ensuring thread safety using double-checked locking. It's found in the `ThreadSafeSingleton` class.

```
public class ThreadSafeSingleton {
    private static volatile ThreadSafeSingleton uniqueInstance;
    private ThreadSafeSingleton() {}
    public static ThreadSafeSingleton getInstance() {
        if (uniqueInstance == null) {
            synchronized (ThreadSafeSingleton.class) {
                if (uniqueInstance == null) {
                    uniqueInstance = new ThreadSafeSingleton();
                }
            }
```

```
        }
        return uniqueInstance;
    }
}
```

---

2. Strategy Design Pattern:

Definition:
Allows selecting an algorithm's implementation at runtime.

Implementation:
The pattern is utilized in modeling an online shopping cart with varying payment methods.

- **PaymentStrategy Interface**:
  All payment methods implement this interface, ensuring consistency and the ability to interchange them easily.

```
public interface PaymentStrategy {
    void processPayment(double amount);
}
```

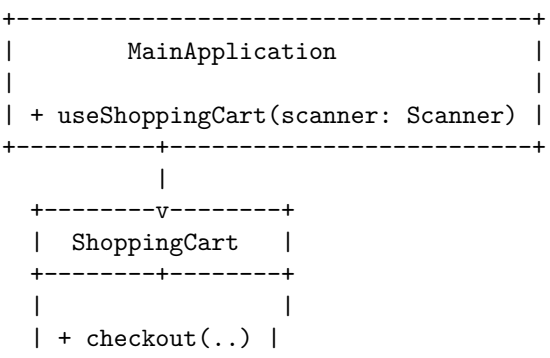
- **Concrete Payment Strategies**:
  Two strategies were implemented:
- `CreditCardPayment`
- `PayPalPayment`

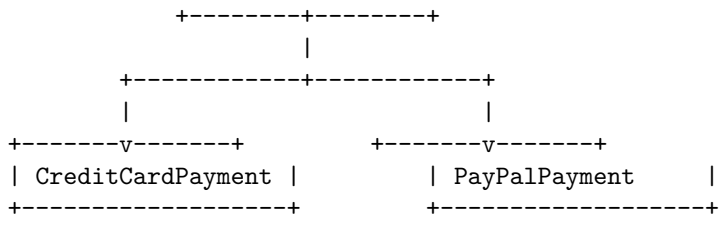
These classes take necessary parameters (like card number or PayPal credentials) and implement the `processPayment` method.

- **Shopping Cart**:
  The `ShoppingCart` class allows products to be added and provides a `checkout` method that accepts a payment strategy and currency, facilitating flexible payment processing.

---

3. Visual Representation:





---

4. Additional Components:

- **Currency and CurrencyConverter**:
  These classes allow multiple currencies, providing flexibility in payment processing. The conversion rates are currently hard-coded but can be extended to fetch real-time conversion rates in the future.

---

5. Entry Point & Client Interaction:

MainApplication Class:
This serves as the primary entry point for the program. Users are presented with a menu allowing them to:

- Demonstrate the Singleton Pattern.
- Use the Shopping Cart (adding products, choosing payment methods, and checking out).
- Exit the application.

Here's a brief section of the interaction code:

```
while (true) {
    System.out.println("\nChoose an option:");
    System.out.println("1. Demonstrate Singleton Pattern");
    System.out.println("2. Use the Shopping Cart");
    System.out.println("3. Exit");
    ...
}
```

---

6. Important Considerations:

- **Thread Safety**: To ensure that the Singleton pattern is thread-safe, we used the double-checked locking principle, guaranteeing a unique instance even in multi-threaded environments.
- **Extensibility**: The Strategy Pattern ensures that adding new payment methods in the future won't require changes to the existing codebase. Simply create a new class implementing the `PaymentStrategy` interface.

- **Currency Flexibility**: Users have the option to select their desired currency during checkout. The `CurrencyConverter` class then handles the conversion based on predefined rates.

---

7. Conclusion:
This project effectively demonstrates the Singleton and Strategy design patterns in a practical scenario. While it serves as a foundational representation, real-world applications would require more comprehensive implementations, especially around secure payment processing and dynamic currency conversion rates.
