Design Pattern Assessment

Name: Ahmed Mohamed Kamel (ahmed.gamel@vodafone.com) | Staff ID: 56485

Assessment 1: the four different types of the Singleton pattern.

1- Eager Initialization Singleton

```
public class EagerInitializedSingleton {

    private static final EagerInitializedSingleton instance = new EagerInitializ
edSingleton();

    private EagerInitializedSingleton(){}

    public static EagerInitializedSingleton getInstance() {
        return instance;
    }
}
```

2- Lazy Initialization Singleton

```
public class LazyInitializedSingleton {

    private static LazyInitializedSingleton instance;

    private LazyInitializedSingleton(){}

    public static LazyInitializedSingleton getInstance() {
        if (instance == null) {
            instance = new LazyInitializedSingleton();
        }
        return instance;
```

```
    }
}
```

3- Thread Safe Singleton

```
public class ThreadSafeSingleton {

    private static ThreadSafeSingleton instance;

    private ThreadSafeSingleton(){}

    public static synchronized ThreadSafeSingleton getInstance() {
        if (instance == null) {
            instance = new ThreadSafeSingleton();
        }
        return instance;
    }

}
```

4- Bill Pugh Singleton

```
public class BillPughSingleton {

    private BillPughSingleton(){}

    private static class SingletonHelper {
        private static final BillPughSingleton INSTANCE = new BillPughSingleto
n();
    }

    public static BillPughSingleton getInstance() {
        return SingletonHelper.INSTANCE;
    }
}
```

Which approach is the most effective and why?

Based on my understanding if we go through each approach we can have some pros and cons, in case of Eager Initialization Singleton it is Simple and thread-safe but the instance is created even if it might not be used, in case of Lazy Initialization Singleton the instance is created only when needed but it is not thread-safe, in case of the Thread-safe Singleton it is thread-safe but slower due to synchronized overhead, the last approach the Bill pugh Singleton it is thread-safe, lazy and no synchronization overhead so the **Bill pugh Singleton considered the most effective approach among the four** because of the Lazy Initialization so the instance is created only when getInstance() is called so it is efficient in terms of memory usage, and it is thread-safe without synchronization overhead, the JVM handles class loading and ensures the static inner class (`Helper`) is loaded only once and in a thread-safe manner. No need for explicit `synchronized`. and it provides High Performance so no locking is required on method calls, unlike synchronized methods which slow down access. finally the code is clean and easy to understand without complex double-checked locking or early object creation.

The difference between using the synchronized keyword on a method versus using a synchronized block in Java.

In case of Synchronized method the Lock Scope is on the whole method but in the Synchronized Block the lock scope is in specific block usually the instance creation, In case of performance the Synchronized method is slower because it is always synchronized but the Synchronized Block is faster because the lock is only when needed.

How each is applied within the Singleton pattern.

Synchronized Method Like the Thread Safe Singleton

```
public class ThreadSafeSingleton {
```

```
    private static ThreadSafeSingleton instance;

    private ThreadSafeSingleton(){}

    public static synchronized ThreadSafeSingleton getInstance() {
        if (instance == null) {
            instance = new ThreadSafeSingleton();
        }
        return instance;
    }

}
```

The getInstance() method is synchronized, so only one thread can execute it at a time, and it ensures thread safety during instance creation.

Synchronized Block

```
public class SingletonSynchronizedBlock {
    private static volatile SingletonSynchronizedBlock instance;

    private SingletonSynchronizedBlock() {}

    public static SingletonSynchronizedBlock getInstance() {
        if (instance == null) { // First check
            synchronized (SingletonSynchronizedBlock.class) {
                if (instance == null) { // Second check
                    instance = new SingletonSynchronizedBlock();
                }
            }
        }
        return instance;
    }
}
```

First check (without lock): Improves performance when instance is already initialized, Second check (with lock): Ensures thread safety when instance is not yet created.

Assessment 2: Enhancement

```
abstract class HotDrink{
    void prepare() {
        boilWater();
        typeOfDrink();
        pourIntoCup();
    }

    void boilWater() {
        System.out.println("Boiling water");
    }

    void pourIntoCup() {
        System.out.println("Pouring into cup");
    }

    abstract void typeOfDrink();
}

class Tea extends HotDrink {
    void typeOfDrink() {
        System.out.println("Adding tea bag");
    }
}

class Coffee extends HotDrink {
    void typeOfDrink() {
        System.out.println("Adding coffee");
    }
}
```

Additional practical example demonstrating the Template pattern.

```
abstract class UserRegistration {
    public final void registerUser() {
        validateData();
        saveToDatabase();
        sendWelcomeEmail();
    }

    abstract void validateData();

    void saveToDatabase() {
        System.out.println("Saving user to the database...");
    }

    void sendWelcomeEmail() {
        System.out.println("Sending welcome email...");
    }
}

class AdminRegistration extends UserRegistration {
    void validateData() {
        System.out.println("Validating admin credentials...");
    }
}

class GuestRegistration extends UserRegistration {
    void validateData() {
        System.out.println("Minimal validation for guest user...");
    }

    @Override
    void sendWelcomeEmail() {
        System.out.println("No email sent for guest.");
    }
}
```