Abstract Factory + Singleton: Análisis Completo

1. Revisión de la problemática

Abstract Factory de manera normal

```
// PROBLEMA: Nueva instancia cada vez
GUIFactory factory1 = new WindowsFactory(); // Instancia A
GUIFactory factory2 = new WindowsFactory(); // Instancia B
// Resultado: Múltiples instancias innecesarias en memoria
```

Abstract Factory con el patrón Singleton

```
// SOLUCIÓN: Siempre la misma instancia
GUIFactory factory1 = SingletonWindowsFactory.getInstance(); // Instancia única
GUIFactory factory2 = SingletonWindowsFactory.getInstance(); // Misma instancia
// Resultado: Una sola instancia, acceso controlado
```

2. Ventajas de la Combinación

2.1 Gestión Eficiente de Memoria

- **Una sola instancia por factory**: Reduce significativamente el consumo de memoria
- **Inicialización lazy**: Se crean solo cuando se necesitan, lo que implica un correcto flujo del proceso
- **Recursos compartidos**: Configuraciones, conexiones y cachés se mantienen centralizados únicamente en la Instancia global

2.2 Consistencia Global

- **Estado compartido**: Todas las partes de la aplicación usan la misma configuración
- **Cambios globales**: Modificar la factory afecta toda la aplicación instantáneamente
- **Configuración centralizada**: Un solo punto de control para toda la familia de productos

2.3 Rendimiento Mejorado

- **Sin una sobrecarga en la creación**: No se crean instancias diferentes con la misma información
- **Cache de productos**: Se pueden implementar cachés a nivel de factory lo que permite el encapsulamiento

3. Casos de Uso Ideales

3.1 Sistemas de UI Multiplataforma

```
// Perfecto para aplicaciones que necesitan adaptar UI según el SO
GUIFactory factory = GUIFactoryManager.getInstance().getFactory();
// Una sola factory por SO, accesible globalmente
```

3.2 Conectores de Base de Datos

```
// Cada tipo de BD tiene una factory singleton
DatabaseFactory mysqlFactory = MySQLFactory.getInstance();
DatabaseFactory postgresFactory = PostgreSQLFactory.getInstance();
// Evita múltiples pools de conexiones innecesarios
```

- **Todos los módulos de la aplicación comparten el mismo *pool de conexiones***

3.3 Procesadores de Documentos

```
// Factory singleton para cada tipo de documento
DocumentFactory pdfFactory = PDFFactory.getInstance();
DocumentFactory wordFactory = WordFactory.getInstance();
// Mantiene configuraciones y recursos de parsing centralizados
```

4. Implementaciones en enfoques más Avanzados

4.1 Factory con Configuración Persistente

```
public class ConfigurableWindowsFactory extends GUIFactory {
    private Properties configuration;

    private ConfigurableWindowsFactory() {
        loadConfiguration();
    }

    private void loadConfiguration() {
        // Cargar configuración una sola vez
        configuration = new Properties();
        // ... lógica de carga
    }

    @Override
    public Button createButton() {
        // Usar configuración singleton para crear productos
        String buttonStyle = configuration.getProperty("button.style", "default");
        return new WindowsButton(buttonStyle);
    }
}
```

4.2 Factory con Cache de Productos

```
public class CachedLinuxFactory extends GUIFactory {
    private final Map<String, Object> productCache = new ConcurrentHashMap<>();

    @Override
    public Button createButton() {
        return (Button) productCache.computeIfAbsent("button",
            k -> new LinuxButton());
    }

    // Los productos se crean una vez y se reutilizan
}
```

5. Patrones de Registro y Discovery

5.1 Factory Registry Singleton

```
public class FactoryRegistry {
    private final Map<String, GUIFactory> factories = new HashMap<>();

    private FactoryRegistry() {
        registerDefaultFactories();
    }

    private static class RegistryHolder {
        private static final FactoryRegistry INSTANCE = new FactoryRegistry();
    }

    public static FactoryRegistry getInstance() {
        return RegistryHolder.INSTANCE;
    }

    private void registerDefaultFactories() {
        factories.put("windows", SingletonWindowsFactory.getInstance());
        factories.put("mac", SingletonMacFactory.getInstance());
        factories.put("linux", SingletonLinuxFactory.getInstance());
    }

    public GUIFactory getFactory(String name) {
        return factories.get(name.toLowerCase());
    }

    public void registerFactory(String name, GUIFactory factory) {
        factories.put(name.toLowerCase(), factory);
    }
}
```

6. Consideraciones de Testing

6.1 Problema: Estado Compartido

```
// PROBLEMA: Los tests pueden afectarse mutuamente
@Test
public void testWindowsFactory() {
    GUIFactory factory = SingletonWindowsFactory.getInstance();
    // Modificar estado de la factory
    // Puede afectar otros tests
}
```

6.2 Solución: Factory Reset para Testing

```
public class TestableWindowsFactory extends GUIFactory {
    // Solo en ambiente de testing
    public static void resetInstance() {
        FactoryHolder.INSTANCE = null; // Cuidado: Solo para tests
    }

    // Método para crear instancia de test
    public static TestableWindowsFactory createTestInstance() {
        return new TestableWindowsFactory();
    }
}
```

6.3 Mejor Solución: Dependency Injection en Tests

```
public class Application {
    private final GUIFactory factory;

    // Constructor para inyección de dependencias
    public Application(GUIFactory factory) {
        this.factory = factory;
    }

    // Constructor por defecto usa singleton
    public Application() {
        this(SingletonWindowsFactory.getInstance());
    }
}

// En tests
@Test
public void testApplication() {
    GUIFactory mockFactory = mock(GUIFactory.class);
    Application app = new Application(mockFactory);
    // Test aislado sin afectar el singleton
}
```

7. Variaciones del Patrón

7.1 Enum Singleton Factory (Más Segura)

```
public enum WindowsFactoryEnum implements GUIFactory {
    INSTANCE;

    @Override
    public Button createButton() {
        return new WindowsButton();
    }

    @Override
    public TextField createTextField() {
        return new WindowsTextField();
    }

    @Override
    public Checkbox createCheckbox() {
        return new WindowsCheckbox();
    }
}

// Uso
GUIFactory factory = WindowsFactoryEnum.INSTANCE;
```

7.2 Factory con Lazy Loading de Productos

```
public class LazyLoadingFactory extends GUIFactory {
    private volatile Button buttonPrototype;
    private volatile TextField textFieldPrototype;

    @Override
    public Button createButton() {
        if (buttonPrototype == null) {
            synchronized (this) {
                if (buttonPrototype == null) {
                    buttonPrototype = new WindowsButton();
                }
            }
        }
        return buttonPrototype.clone(); // Assuming cloneable
    }
}
```

- **En vez de retornar la instancia correspondiente, la clona y mantiene instancias repetidas**

8. Anti-Patrones y Problemas Comunes

8.1 ❌ Factory Singleton con Estado Mutable

```
// MALO: Estado mutable en singleton factory
public class BadFactory extends GUIFactory {
    private String currentTheme; // Problema: estado mutable compartido

    public void setTheme(String theme) {
        this.currentTheme = theme; // Afecta globalmente
    }
}
```

8.2 ✅ Factory Singleton Inmutable

```
// BUENO: Factory inmutable con configuración externa
public class GoodFactory extends GUIFactory {
    private final String theme;

    private GoodFactory() {
        this.theme = ConfigurationManager.getTheme(); // Inmutable
    }

    @Override
    public Button createButton() {
        return new WindowsButton(this.theme);
    }
}
```

8.3 ❌ Violación del Principio de Responsabilidad Única

```
// MALO: Factory que hace demasiado
public class OverloadedFactory extends GUIFactory {
    // ❌ No debería manejar logging
    public void logActivity(String message) { }

    // ❌ No debería manejar configuración
    public void saveConfiguration() { }

    // ❌ No debería manejar validación
    public boolean validateInput(String input) { return true; }
}
```

9. Mejores Prácticas

9.1 ✅ Separación de Responsabilidades

- Factory se encarga solo de crear objetos
- Se establecen responsabilidades para cada una de los anteriores usos
    - Configuración manejada por un Configuration Manager
    - Logging manejado por un Logger separado
    - Validación manejada por Validators específicos

10. Herencia al combinar Singleton y Abstract Factory

La combinación del patrón Singleton con Abstract Factory presenta desafíos únicos en términos de herencia, ya que ambos patrones manejan la creación de instancias de manera muy específica. El Singleton garantiza una única instancia por clase, mientras que Abstract Factory se enfoca en crear familias de objetos relacionados.

El problema fundamental de la herencia con Singleton

Instancia única vs. Jerarquía de clases

Cuando una clase padre implementa Singleton y tiene clases hijas, surge la pregunta: **¿debe haber una sola instancia para toda la jerarquía o una instancia por cada clase derivada?**

```
// Caso problemático
class SingletonFactory {  // Clase padre
    private static SingletonFactory instance;

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

class ConcreteFactoryA extends SingletonFactory {
    // ¿Cómo maneja su propia instancia única?
}
```

Dos enfoques principales

1. Singleton por jerarquía completa

- Una sola instancia para la clase padre y todas sus hijas
- Las clases derivadas no pueden tener sus propias instancias
- Limita severamente la flexibilidad del Abstract Factory

2. Singleton por clase derivada

- Cada clase en la jerarquía mantiene su propia instancia única
- Más flexible pero requiere implementación cuidadosa

Implementación: Singleton por clase derivada

Estructura recomendada

```
// Clase base abstracta
abstract class AbstractSingletonFactory {
    private static final Map<Class<?>, AbstractSingletonFactory> instances =
        new ConcurrentHashMap<>();

    @SuppressWarnings("unchecked")
    protected static <T extends AbstractSingletonFactory> T getInstance(Class<T> clazz) {
        return (T) instances.computeIfAbsent(clazz, k -> {
            try {
                Constructor<T> constructor = clazz.getDeclaredConstructor();
                constructor.setAccessible(true);
                return constructor.newInstance();
            } catch (Exception e) {
                throw new RuntimeException("Error creating singleton instance", e);
            }
        });
    }

    // Métodos del Abstract Factory
    public abstract ProductA createProductA();
    public abstract ProductB createProductB();
}

// Implementaciones concretas
class ConcreteFactory1 extends AbstractSingletonFactory {
    public static ConcreteFactory1 getInstance() {
        return getInstance(ConcreteFactory1.class);
    }

    @Override
    public ProductA createProductA() {
        return new ConcreteProductA1();
    }

    @Override
    public ProductB createProductB() {
        return new ConcreteProductB1();
    }
}

class ConcreteFactory2 extends AbstractSingletonFactory {
    public static ConcreteFactory2 getInstance() {
        return getInstance(ConcreteFactory2.class);
    }

    @Override
    public ProductA createProductA() {
        return new ConcreteProductA2();
    }

    @Override
    public ProductB createProductB() {
        return new ConcreteProductB2();
    }
}
```

Efectos en las relaciones padre-hijo

En las clases Factory

Ventajas:

- **Independencia de instancias:** Cada factory concreta mantiene su propia instancia única
- **Polimorfismo preservado:** Se puede trabajar con referencias al tipo base

- **Extensibilidad:** Nuevas factories pueden agregarse fácilmente

Desventajas:

- **Complejidad adicional:** Requiere manejo de reflexión o registro manual
- **Overhead de memoria:** Una instancia por cada clase derivada
- **Posible confusión conceptual:** Múltiples "singletons" en una jerarquía

En los productos creados

Los productos creados por las factories **no se ven afectados** directamente por el patrón Singleton de la factory. Cada llamada a `createProduct()` puede crear nuevas instancias de productos según sea necesario.

Consideraciones de diseño

Thread Safety

```
// Implementación thread-safe usando enum (recomendada)
public enum ConcreteFactory1 implements AbstractFactory {
    INSTANCE;

    @Override
    public ProductA createProductA() {
        return new ConcreteProductA1();
    }

    @Override
    public ProductB createProductB() {
        return new ConcreteProductB1();
    }
}
```

Serialización y deserialización

```
// Método para preservar singleton después de deserialización
private Object readResolve() {
    return getInstance(this.getClass());
}
```

Alternativas y patrones relacionados

1. Registry Pattern + Abstract Factory

En lugar de Singleton, usar un registro de factories:

```
class FactoryRegistry {
    private static final Map<String, AbstractFactory> factories = new HashMap<>();

    public static void registerFactory(String type, AbstractFactory factory) {
        factories.put(type, factory);
    }

    public static AbstractFactory getFactory(String type) {
        return factories.get(type);
    }
}
```

2. Factory Method con Singleton

Usar Factory Method en lugar de Abstract Factory cuando el Singleton es más crítico.

Mejores prácticas

✅ Recomendaciones

1. **Evaluar casos de uso:** Preguntarse si realmente se necesitan ambos patrones
2. **Documentación clara:** El comportamiento de herencia debe estar bien documentado
3. **Usar enums cuando sea posible:** Para implementations simples de Singleton
4. **Considerar inyección de dependencias:** Como alternativa más flexible

❌ Errores comunes

1. **No considerar thread safety** en la implementación de herencia
2. **Asumir una sola instancia** para toda la jerarquía sin evaluarlo
3. **Sobrecomplicar** cuando patterns más simples serían suficientes

Conclusión

La combinación de Singleton y Abstract Factory con herencia requiere decisiones cuidadosas sobre el manejo de instancias. La implementación más flexible es mantener una instancia única por cada clase derivada, pero esto introduce complejidad adicional.

En muchos casos, considerar alternativas como Registry Pattern o inyección de dependencias puede resultar en un diseño más limpio y mantenible. La clave está en evaluar si los beneficios de combinar ambos patrones superan la complejidad añadida en el contexto específico del proyecto.