# 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);
    }
}
```

### 5.2 Factory Provider con Auto-Detection

```
public class AutoDetectFactoryProvider {
    private GUIFactory detectedFactory;

    private AutoDetectFactoryProvider() {
        this.detectedFactory = detectAndCreateFactory();
    }

    private static class ProviderHolder {
        private static final AutoDetectFactoryProvider INSTANCE =
            new AutoDetectFactoryProvider();
    }

    public static AutoDetectFactoryProvider getInstance() {
        return ProviderHolder.INSTANCE;
    }

    private GUIFactory detectAndCreateFactory() {
        String os = System.getProperty("os.name").toLowerCase();
        if (os.contains("windows")) {
            return SingletonWindowsFactory.getInstance();
        } else if (os.contains("mac")) {
            return SingletonMacFactory.getInstance();
        } else {
            return SingletonLinuxFactory.getInstance();
        }
    }

    public GUIFactory getFactory() {
        return detectedFactory;
    }
}
```

## 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