Análisis Comparativo de Patrones de Diseño: Singleton, Abstract Factory y Observer

Caetano Flores
Jordan Guaman
Anthony Morales
Leonardo Narvaez

Noviembre 2025

Índice

1. Introducción

Este informe presenta un análisis comparativo exhaustivo de tres patrones de diseño fundamentales aplicados en el desarrollo de una aplicación CRUD para la gestión de estudiantes: Singleton, Abstract Factory y Observer. Cada uno de estos patrones fue implementado en versiones separadas del mismo sistema para evaluar sus características, ventajas, limitaciones y casos de uso óptimos.

La aplicación CRUD base implementa una arquitectura de 3 capas (Presentación, Lógica de Negocio y Datos) con las siguientes características comunes:

- Gestión de estudiantes con atributos: ID, nombres y edad
- Operaciones CRUD completas: Create, Read, Update, Delete
- Interfaz gráfica desarrollada en Java Swing
- Almacenamiento en memoria mediante `ArrayList`
- Validaciones robustas en capa de negocio

El objetivo de este análisis es proporcionar una comprensión profunda de cómo cada patrón afecta la arquitectura, mantenibilidad, extensibilidad y calidad del código, ayudando a los desarrolladores a tomar decisiones informadas sobre qué patrón utilizar según el contexto y requisitos específicos de sus proyectos.

2. Descripción de los Patrones Implementados

2.1. Patrón Singleton

El patrón Singleton es un patrón de diseño creacional que garantiza que una clase tenga una única instancia en toda la aplicación y proporciona un punto de acceso global a dicha instancia.

2.1.1. Implementación en el Proyecto

En la versión inicial del CRUD, el patrón Singleton se implementó en la clase `EstudianteRepository`

- **Instancia estática final:** `private static final EstudianteRepository INSTANCE`
- **Constructor privado:** Previene la creación directa de instancias desde fuera de la clase
- **Método de acceso:** `public static EstudianteRepository getInstance()` retorna siempre la misma instancia
- **Uso en servicio:** `EstudianteService` accede al repositorio mediante `EstudianteRepository.ge`

2.1.2. Características Principales

- Garantiza una única instancia del repositorio durante toda la ejecución
- Acceso global simplificado desde cualquier punto del código
- Persistencia de datos garantizada en memoria compartida
- Implementación técnicamente simple y directa
- Control estricto sobre la instanciación de la clase

2.2. Patrón Abstract Factory

El patrón Abstract Factory es un patrón de diseño creacional que proporciona una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas.

2.2.1. Implementación en el Proyecto (CRUD2)

En la versión CRUD2, el patrón Abstract Factory reemplazó al Singleton:

- **Clase abstracta:** `AbstractRepositoryFactory` define el método abstracto `createEstudianteRe`
- **Factory concreto:** `RepositoryFactory` extiende la clase abstracta e implementa la creación de repositorios
- **Repositorio sin Singleton:** `EstudianteRepository` con constructor público, creado por el factory
- **Uso en servicio:** `EstudianteService` utiliza el factory para obtener instancias del repositorio en el constructor

2.2.2. Características Principales

- Desacopla el código cliente de las clases concretas
- Permite cambiar fácilmente entre diferentes implementaciones
- Centraliza la lógica de creación de objetos
- Cumple con el principio de inversión de dependencias (DIP)
- Facilita la extensibilidad sin modificar código existente (Open/Closed)

2.3. Patrón Observer

El patrón Observer es un patrón de diseño de comportamiento que define una dependencia de uno-a-muchos entre objetos, de manera que cuando un objeto cambia su estado, todos sus dependientes son notificados y actualizados automáticamente.

2.3.1. Implementación en el Proyecto (CRUD3)

En la versión CRUD3, el patrón Observer se añadió al sistema:

- **Interfaz Observer:** `EstudianteObserver` define métodos `onEstudianteAgregado()`, `onEstudianteEditado()`, `onEstudianteEliminado()`
- **Interfaz Subject:** `EstudianteSubject` define `agregarObservador()`, `removerObservador()`
- **Subject concreto:** `EstudianteRepository` implementa `EstudianteSubject` y notifica a observadores
- **Observer concreto:** `EstudianteLogger` implementa `EstudianteObserver` para logging en consola
- **Configuración:** Los observadores se registran en `Main.java` al iniciar la aplicación

2.3.2. Características Principales

- Desacopla el emisor de eventos de los receptores
- Permite agregar/remover observadores dinámicamente
- Soporta comunicación broadcast de uno-a-muchos
- Promueve bajo acoplamiento y alta cohesión
- Facilita la extensibilidad mediante nuevos observadores

3. Comparación Detallada de los Patrones

3.1. Clasificación de Patrones

Patrón	**Categoría**	**Propósito Principal**
Singleton	Creacional	Control de instanciación
Abstract Factory	Creacional	Creación flexible de objetos
Observer	Comportamiento	Notificación de eventos

Cuadro 1: Clasificación de los patrones según Gang of Four

3.2. Problema que Resuelve Cada Patrón

3.2.1. Singleton

Resuelve el problema de **control de instanciación** cuando se necesita exactamente una instancia de una clase. En el proyecto CRUD, evita la creación de múltiples repositorios con listas diferentes de estudiantes, lo que causaría pérdida de datos y comportamiento inconsistente. Garantiza un punto de acceso global compartido a los datos.

3.2.2. Abstract Factory

Resuelve el problema de **acoplamiento directo** entre clases cliente y clases concretas. En CRUD2, permite que `EstudianteService` no dependa directamente de la implementación específica de `EstudianteRepository`, facilitando cambiar entre diferentes implementaciones (memoria, archivo, base de datos) sin modificar el código del servicio.

3.2.3. Observer

Resuelve el problema de **comunicación desacoplada** entre objetos. En CRUD3, permite que múltiples componentes (loggers, auditores, notificadores) reaccionen automáticamente a cambios en el repositorio sin que éste necesite conocer quiénes son o qué hacen, promoviendo bajo acoplamiento.

3.3. Capas de Utilización

Patrón	**Capa(s) de Utilización**
Singleton	Capa de Datos (`EstudianteRepository`). Accedido desde la capa de Lógica de Negocio mediante `getInstance()`.
Abstract Factory	Entre capa de Lógica de Negocio y Datos. El paquete `factory` actúa como puente desacoplando `EstudianteService` del repositorio.
Observer	Capa de Datos como Subject (`EstudianteRepository`). Observadores en paquete dedicado. Configuración en capa de Presentación (`Main`).

Cuadro 2: Ubicación de cada patrón en la arquitectura de 3 capas

3.4. Influencia en el Mantenimiento

3.4.1. Singleton

Ventajas:

- Simplifica el acceso a recursos compartidos con un único punto de acceso
- Reduce bugs relacionados con múltiples instancias inconsistentes
- Centraliza la gestión de datos facilitando debugging

Desventajas:

- Dificulta el testing unitario por el estado global compartido
- Complica el uso de inyección de dependencias modernas
- Puede ocultar dependencias entre clases

3.4.2. Abstract Factory

Ventajas:

- Permite agregar nuevas implementaciones sin modificar código existente (Open/-Closed)
- Facilita testing mediante inyección de implementaciones mock
- Reduce acoplamiento haciendo el sistema más modular
- Centraliza lógica de creación evitando código duplicado

Desventajas:

- Introduce complejidad adicional con clases abstractas y concretas
- Puede resultar excesivo para sistemas muy simples
- Requiere crear nuevas clases factory para cada familia de productos

3.4.3. Observer

Ventajas:

- Permite agregar nuevos observadores sin modificar el Subject (Open/Closed)
- Facilita debugging activando/desactivando observadores según necesidad
- Reduce acoplamiento entre componentes reactivos
- Permite extensión mediante nuevos comportamientos

Desventajas:

- Las notificaciones ocurren en orden arbitrario
- Puede causar actualizaciones en cascada inesperadas
- Riesgo de memory leaks si observadores no se remueven correctamente
- Overhead de rendimiento con muchos observadores

3.5. Prevención de Fallas de Diseño

Patrón	Fallas de Diseño que Previene
Singleton	▪ Inconsistencias por múltiples instancias con datos diferentes ▪ Problemas de sincronización de datos ▪ Pérdida de estado compartido
Abstract Factory	▪ Acoplamiento fuerte entre servicio y repositorio ▪ Violaciones del principio Open/Closed ▪ Dificultad para testing con dependencias concretas ▪ Código de creación disperso y duplicado
Observer	▪ Acoplamiento fuerte entre emisor y receptores de eventos ▪ Referencias cíclicas y dependencias complejas ▪ Dificultad para testing de notificaciones ▪ Código duplicado de notificación

Cuadro 3: Fallas de diseño prevenidas por cada patrón

3.6. Complejidad de Implementación

Aspecto	Singleton	Factory	Observer
Clases adicionales	0	2	3+
Interfaces necesarias	0	0	2
Complejidad código	Baja	Media	Media
Curva aprendizaje	Baja	Media	Media
Líneas de código	Mínimas	Moderadas	Moderadas

Cuadro 4: Comparación de complejidad de implementación

3.7. Extensibilidad y Escalabilidad

3.7.1. Singleton

Extensibilidad: Limitada. Agregar variantes del repositorio requeriría modificar el patrón Singleton o crear múltiples Singletons independientes, lo que complica el diseño.

Escalabilidad: Problemática en sistemas distribuidos. El Singleton es específico a una JVM, dificultando la escalabilidad horizontal y sistemas multi-instancia.

3.7.2. Abstract Factory

Extensibilidad: Excelente. Se pueden agregar fácilmente nuevos factories concretos:

- `DatabaseRepositoryFactory` para persistencia en BD
- `FileRepositoryFactory` para archivos JSON/XML
- `CachedRepositoryFactory` con sistema de caché

Escalabilidad: Buena. Facilita la migración a diferentes tecnologías de persistencia según las necesidades de escala del sistema.

3.7.3. Observer

Extensibilidad: Excelente. Se pueden agregar nuevos observadores sin límite:

- `EstudianteAuditor` para auditoría en BD
- `EstudianteEmailNotifier` para notificaciones
- `EstudianteStatistics` para métricas en tiempo real
- `EstudianteCacheInvalidator` para gestión de caché

Escalabilidad: Media. Puede presentar problemas de rendimiento con muchos observadores, pero se puede optimizar con observadores asíncronos o selectivos.

3.8. Casos de Uso Óptimos

3.8.1. Cuándo Usar Singleton

- Necesitas exactamente una instancia de una clase (configuración, pool de conexiones)
- Requieres punto de acceso global a un recurso compartido
- El objeto es costoso de crear y no cambiará durante la ejecución
- No necesitas inyección de dependencias ni testing avanzado
- La aplicación es monolítica y no se distribuirá

Ejemplo en el proyecto: Repositorio en memoria cuando solo se necesita una lista compartida de estudiantes y la aplicación es simple.

3.8.2. Cuándo Usar Abstract Factory

- El sistema debe ser independiente de cómo se crean sus objetos
- Necesitas trabajar con familias de productos relacionados
- Prevés cambios en las implementaciones concretas
- Requieres facilitar el testing con mocks
- Quieres centralizar la lógica de creación

Ejemplo en el proyecto: Cuando se planea migrar de almacenamiento en memoria a base de datos, o soportar múltiples backends simultáneamente.

3.8.3. Cuándo Usar Observer

- Un cambio en un objeto requiere cambiar otros objetos desconocidos
- Necesitas notificaciones broadcast de eventos
- Quieres desacoplar emisores de receptores de eventos
- Requieres comportamientos reactivos extensibles
- Necesitas logging, auditoría o monitoreo de cambios

Ejemplo en el proyecto: Cuando necesitas auditoría, logging, notificaciones por email, o actualización de estadísticas cada vez que se modifican los datos.

4. Análisis de Principios SOLID

4.1. Single Responsibility Principle (SRP)

Singleton: Cumplimiento parcial. El repositorio maneja tanto el almacenamiento como el control de instanciación.

Abstract Factory: Excelente cumplimiento. Separa la responsabilidad de creación (factory) del almacenamiento (repositorio).

Observer: Excelente cumplimiento. Separa la lógica de notificación de la lógica de negocio del repositorio.

4.2. Open/Closed Principle (OCP)

Singleton: Cumplimiento bajo. Difícil extender sin modificar la clase Singleton.

Abstract Factory: Excelente cumplimiento. Abierto para extensión (nuevos factories) y cerrado para modificación.

Observer: Excelente cumplimiento. Se pueden agregar observadores sin modificar el Subject.

4.3. Liskov Substitution Principle (LSP)

Singleton: No aplicable directamente (no hay jerarquía de herencia).

Abstract Factory: Cumplimiento excelente. Cualquier factory concreto puede sustituir al abstracto.

Observer: Cumplimiento excelente. Cualquier observador concreto puede sustituir a la interfaz.

4.4. Interface Segregation Principle (ISP)

Singleton: No aplicable (no usa interfaces).

Abstract Factory: Cumplimiento bueno. Interfaces específicas para creación de repositorios.

Observer: Cumplimiento excelente. Interfaces segregadas para Observer y Subject.

4.5. Dependency Inversion Principle (DIP)

Singleton: Cumplimiento bajo. El servicio depende directamente de la clase concreta Singleton.

Abstract Factory: Excelente cumplimiento. El servicio depende de la abstracción (factory abstracto) no de la implementación concreta.

Observer: Excelente cumplimiento. El Subject depende de la interfaz Observer, no de implementaciones concretas.

5. Testing y Mantenibilidad

5.1. Facilidad de Testing Unitario

Patrón	Puntuación	Observaciones
Singleton	3/10	Muy difícil. Estado global compartido dificulta aislar tests. Requiere técnicas especiales como reflection.
Abstract Factory	9/10	Excelente. Fácil inyectar mocks mediante factory de prueba. Testing aislado y limpio.
Observer	8/10	Muy bueno. Fácil crear observadores mock para verificar notificaciones. Posible aislar Subject.

Cuadro 5: Evaluación de facilidad para testing unitario

5.2. Facilidad de Debugging

Singleton: Media. El estado global puede ser conveniente para inspeccionar, pero dificulta rastrear quién modifica los datos.

Abstract Factory: Alta. El flujo de creación es claro y rastreable. Fácil identificar qué factory se está usando.

Observer: Media-Alta. Los logs de observadores ayudan al debugging, pero las cascadas de notificaciones pueden complicar el rastreo.

5.3. Refactoring y Evolución del Código

Singleton: Difícil. Migrar de Singleton a otro patrón requiere cambios en todo el código que lo usa.

Abstract Factory: Fácil. Agregar nuevas implementaciones o modificar existentes es straightforward.

Observer: Fácil. Agregar, modificar o remover observadores no afecta al código existente.

6. Rendimiento y Recursos

6.1. Overhead de Memoria

- **Singleton:** Mínimo. Solo una instancia en memoria durante toda la ejecución.
- **Abstract Factory:** Bajo. Una instancia del factory más las instancias creadas (similar a uso normal).
- **Observer:** Medio. Lista de observadores más instancias de cada observador concreto.

6.2. Overhead de Procesamiento

- **Singleton:** Mínimo. Acceso directo mediante método estático.
- **Abstract Factory:** Bajo. Llamada adicional al factory durante la creación.
- **Observer:** Medio-Alto. Iteración sobre observadores y llamadas a métodos de notificación en cada evento.

6.3. Optimizaciones Posibles

Singleton: Lazy initialization, double-checked locking para thread-safety.
Abstract Factory: Caché de instancias creadas, factories con pools de objetos.
Observer: Notificaciones asíncronas, observadores selectivos por tipo de evento, weak references para prevenir memory leaks.

7. Combinación de Patrones

Los tres patrones no son mutuamente excluyentes y pueden combinarse efectivamente:

7.1. Singleton + Observer

El repositorio puede ser Singleton para garantizar una única instancia Y Observer para notificar cambios. Esto combina consistencia de datos con capacidad de reacción.

7.2. Abstract Factory + Observer

El factory puede crear repositorios que implementan el patrón Observer. Diferentes factories podrían crear repositorios con diferentes estrategias de notificación.

7.3. Implementación Ideal para el Proyecto

Para un sistema CRUD completo y profesional, la combinación óptima sería:

- **Abstract Factory:** Para crear repositorios (flexibilidad de implementación)
- **Observer:** En el repositorio para notificar eventos (logging, auditoría)
- **Sin Singleton:** Usar inyección de dependencias en su lugar

8. Recomendaciones por Contexto

8.1. Para Proyectos Pequeños (¡10 clases)

Recomendación: Singleton o implementación directa sin patrón.
Justificación: La simplicidad es más valiosa que la flexibilidad. Abstract Factory y Observer añaden complejidad innecesaria.

8.2. Para Proyectos Medianos (10-50 clases)

Recomendación: Abstract Factory + Observer selectivo.
Justificación: La extensibilidad comienza a ser importante. El Factory facilita testing y cambios futuros. Observer para funcionalidades críticas como auditoría.

8.3. Para Proyectos Grandes (¿50 clases)

Recomendación: Abstract Factory + Observer + Inyección de Dependencias.
Justificación: La arquitectura sólida es crucial. Evitar Singleton en favor de DI frameworks. Usar extensivamente Factory y Observer para modularidad.

8.4. Para Sistemas Distribuidos

Recomendación: Evitar Singleton. Usar Abstract Factory + Observer con implementaciones distribuidas.
Justificación: Singleton no funciona en entornos distribuidos. Factory permite crear repositorios distribuidos. Observer puede usar sistemas de mensajería (Kafka, RabbitMQ).

9. Lecciones Aprendidas

9.1. Del Patrón Singleton

- La simplicidad es valiosa, pero puede convertirse en deuda técnica
- El estado global facilita el desarrollo inicial pero complica el testing
- Útil para prototipos y MVPs, pero considerar alternativas para producción
- Excelente para configuraciones y recursos verdaderamente únicos

9.2. Del Patrón Abstract Factory

- La inversión inicial en abstracciones paga dividendos a largo plazo
- Facilita enormemente cambios de tecnología (de memoria a BD)
- El testing se vuelve trivial con la capacidad de inyectar mocks
- La complejidad adicional está justificada en sistemas que evolucionarán

9.3. Del Patrón Observer

- La comunicación desacoplada es poderosa para extensibilidad
- Permite agregar funcionalidades (logging, auditoría) sin tocar código existente
- Requiere disciplina para evitar cascadas de notificaciones complejas
- Ideal para sistemas reactivos y event-driven architectures

10. Conclusiones

1. **Singleton vs Factory - Problema que resuelven:** Singleton resuelve control de instanciación garantizando una única instancia, mientras Factory resuelve acoplamiento directo permitiendo flexibilidad en la creación. Para un CRUD evolutivo, Factory es superior al facilitar migración entre implementaciones (memoria → BD) sin modificar el servicio.

2. **Factory vs Observer - Capa de utilización:** Factory opera entre capas de Negocio y Datos como puente de creación, mientras Observer opera dentro de la capa de Datos con configuración en Presentación. Son complementarios: Factory decide QUÉ crear, Observer determina QUIÉN es notificado de cambios. Su combinación proporciona máxima flexibilidad arquitectónica.

3. **Los tres patrones - Influencia en mantenimiento:** Singleton dificulta testing y extensión pero simplifica acceso. Factory facilita testing y cambios de implementación cumpliendo Open/Closed. Observer permite agregar comportamientos reactivos sin modificar código existente. Para mantenibilidad a largo plazo, Factory + Observer superan a Singleton, justificando su mayor complejidad inicial.

4. **Prevención de fallas y selección de patrón:** Singleton previene inconsistencias de múltiples instancias pero crea acoplamiento global. Factory previene acoplamiento fuerte y violaciones de DIP facilitando arquitecturas limpias. Observer previene acoplamiento entre emisores y receptores de eventos promoviendo extensibilidad. La selección debe basarse en: proyecto pequeño → Singleton, proyecto evolutivo → Factory, sistema con auditoría/logging → Observer, sistema empresarial → Factory + Observer.

11. Recomendaciones Finales

1. **Evaluar el contexto antes de elegir:** No existe un patrón universalmente superior. Singleton es apropiado para prototipos y configuraciones simples. Factory es ideal cuando se prevén cambios de implementación. Observer es necesario para sistemas reactivos con auditoría y logging.

2. **Combinar patrones estratégicamente:** Los patrones no son mutuamente excluyentes. Una arquitectura robusta puede beneficiarse de Factory para creación flexible + Observer para notificaciones + evitar Singleton en favor de inyección de dependencias, obteniendo lo mejor de cada enfoque.

3. **Priorizar testing y mantenibilidad:** En proyectos empresariales, la facilidad de testing debe ser un criterio primordial. Factory y Observer facilitan testing unitario mediante inyección de mocks, mientras que Singleton lo dificulta significativamente. El costo inicial de abstracción se recupera rápidamente en mantenimiento.

4. **Prepararse para evolución:** Los sistemas de software evolucionan inevitablemente. Usar Factory desde el inicio permite migrar de almacenamiento en memoria a archivos o base de datos sin refactoring masivo. Implementar Observer tempranamente facilita agregar auditoría, métricas y notificaciones cuando los requisitos cambien, evitando reescrituras costosas.

12. Referencias

- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
- Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall.
- Freeman, E., & Robson, E. (2020). *Head First Design Patterns*. O'Reilly Media.
- Fowler, M. (2018). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional.
- Oracle Corporation. (2024). *Java Platform, Standard Edition Documentation*.

12.1. Información del Proyecto

- **Institución:** Escuela Politécnica del Ejército (ESPE)
- **Asignatura:** Arquitectura de Software
- **Taller:** U2T4 - Análisis Comparativo de Patrones de Diseño
- **Grupo:** G6
- **Integrantes:** Caetano Flores, Jordan Guaman, Anthony Morales, Leonardo Narvaez
- **Fecha:** Noviembre 2025