Arquitectura NVC y Persistencia Singleton

Universidad de las Fuerzas Armadas ESPE

Carlos Ñato, Juan Granda, David Cepeda

Análisis y Diseño de Software

Sangolquí, 25 de Noviembre del 2025

1. Introducción y Contextualización del Proyecto

La ingeniería de software contemporánea se encuentra en una encrucijada constante entre la adopción de patrones de diseño clásicos, establecidos durante las décadas formativas de la programación orientada a objetos, y la integración de nuevas metodologías de documentación asistidas por inteligencia artificial. Este informe técnico presenta un análisis forense y prospectivo de un proyecto de desarrollo de software basado en Java, cuyo objetivo central es la gestión de información estudiantil mediante una arquitectura de escritorio. El documento base analizado, titulado *Implementación del Patrón Singleton y Arquitectura NVC en Java*, no solo propone una solución técnica a un problema recurrente —la persistencia de datos en memoria— sino que también sirve como artefacto cultural que evidencia la transición hacia herramientas de presentación generativa como Gamma App.[1]

El mandato del análisis es desglosar la viabilidad técnica, la robustez arquitectónica y las implicaciones de mantenimiento del sistema propuesto. El sistema en cuestión es una aplicación CRUD (Crear, Leer, Actualizar, Eliminar) diseñada sobre la biblioteca gráfica Java Swing. El desafío fundamental que busca resolver es la volatilidad del estado en una aplicación de múltiples vistas: cómo mantener una lista de entidades (estudiantes) consistente y accesible mientras el usuario navega entre diferentes paneles de interfaz, sin recurrir a un motor de base de datos persistente en disco como MySQL o PostgreSQL.[1]

Para abordar este reto, el proyecto introduce dos decisiones de diseño críticas: la adopción de una arquitectura denominada **NVC (Negocio - Vista - Control)** y la implementación del **Patrón Singleton** para el repositorio de datos. Este informe no se limita a describir estas decisiones, sino que las somete a un escrutinio riguroso, contrastándolas con estándares industriales, literatura académica sobre patrones de diseño y discusiones contemporáneas en foros de ingeniería sobre la validez de estas arquitecturas en el desarrollo moderno.[2] Además, se examina el medio de entrega del proyecto —la plataforma Gamma— como un indicador de cómo la inteligencia artificial está redefiniendo la fase de especificación y presentación en el ciclo de vida del desarrollo de software.[5]

1.1 El Problema de la Persistencia Volátil en Aplicaciones Swing

En el desarrollo de aplicaciones de escritorio con Java Swing, la gestión del ciclo de vida de los objetos es una preocupación primordial. Por defecto, Swing opera bajo un modelo de componentes donde las ventanas (JFrames) o paneles (JPanels) pueden ser creados y destruidos dinámicamente según la interacción del usuario. El documento del proyecto identifica correctamente el problema: si la estructura de datos que almacena a los estudiantes se instancia dentro de una vista o un controlador local, dicha estructura está vinculada al ciclo de vida de ese componente específico. Al cerrar la vista o cambiar de contexto, el recolector de basura (*Garbage Collector*) de la Máquina Virtual Java (JVM) reclama la memoria, resultando en la pérdida irreversible de los datos ingresados.[1]

La "Necesidad Fundamental" descrita es evitar que la lista de estudiantes se reinicie. En arquitecturas empresariales, esto se resuelve delegando el estado a una base de datos externa. Sin embargo, para aplicaciones ligeras, prototipos o sistemas donde la latencia de E/S de disco es inaceptable, la persistencia en memoria (*in-memory persistence*) es una estrategia válida, siempre que se garantice la referencia global y la unicidad del almacenamiento. Aquí es donde el proyecto propone el Singleton como mecanismo de "bloqueo de creación de múltiples instancias", una solución que, si bien efectiva, introduce complejidades significativas en términos de concurrencia y acoplamiento que serán analizadas en profundidad en las secciones subsiguientes.[1]

2. Paradigmas Arquitectónicos: Deconstrucción del Modelo NVC

La arquitectura de software elegida para organizar el código es referida en el documento como **NVC (Negocio - Vista - Control)**. Esta nomenclatura merece un análisis detallado, ya que representa una desviación —o una adaptación específica— del omnipresente patrón MVC (Modelo - Vista - Controlador).

2.1 Definición y Roles en la Arquitectura NVC

Según la documentación del proyecto, la arquitectura NVC propone una separación de responsabilidades en tres capas bien definidas, diseñadas para facilitar el desarrollo modular y el mantenimiento.[1]

Capa Arquitectónica	Componente Java Representativo	Responsabilidad Principal	Interacción y Flujo de Datos
V - Vista (View)	Clases Swing (JFrame, JPanel)	Interfaz de Usuario (UI). Presentación visual de datos y captura de eventos. Se define explícitamente como "tonta" (*dumb view*).	Comunica eventos al Controlador. No posee lógica de negocio ni manipula datos directamente.
C - Control (Control)	EstudianteController	Intermediario inteligente. Orquestación de flujo, transformación de datos (parsing) y manejo de excepciones.	Recibe inputs de la Vista, invoca servicios de la capa de Negocio y actualiza la Vista según el resultado.
N - Negocio (Business)	Service y Repository	Lógica pura del dominio. Aplicación de reglas de negocio y gestión de la persistencia en memoria (Singleton).	Recibe comandos del Controlador, procesa o almacena datos, y retorna resultados operacionales.

2.1.1 La Capa de Vista: El Patrón "Dumb View"

El proyecto enfatiza que la vista debe mantenerse "tonta". En la teoría de diseño de interfaces, esto se alinea con el principio de separación de preocupaciones (*Separation of Concerns*). La vista, construida con componentes Swing como botones y tablas, no debe "saber" qué sucede con los datos. Su única función es pintar la pantalla y delegar cualquier interacción al controlador. El documento especifica que la vista maneja eventos de clic y entrada de teclado, pasándolos al controlador.[1] Esta restricción es vital para la testabilidad; una vista sin lógica puede ser reemplazada o modificada (por ejemplo, cambiando el *Look and Feel* de Swing) sin riesgo de romper las reglas de negocio.

2.1.2 La Capa de Control: Transformación y Protección

El Controlador en NVC se describe como un componente que realiza "Parsing" y "Manejo de Excepciones". Esto es una distinción importante respecto a implementaciones MVC más laxas donde el controlador es un simple "pasamanos". Aquí, el controlador actúa como un *firewall* para la capa de negocio. Si un usuario introduce texto en un campo numérico, es responsabilidad del Controlador interceptar este error de formato antes de que llegue al modelo de dominio. Esta validación temprana mejora la robustez del sistema y previene que datos corruptos contaminen el repositorio en memoria.[1]

2.1.3 La Capa de Negocio: El Núcleo Lógico

Lo que en MVC tradicional se denomina "Modelo", aquí se explicita como "Negocio". Esta capa encapsula tanto los servicios (la lógica de qué hacer con los datos) como el repositorio (dónde están los datos). Es en esta capa, específicamente en el Repositorio, donde se implementa el Patrón Singleton para garantizar la integridad referencial de los datos a través de toda la aplicación. El documento describe esta capa como el "corazón" de la aplicación, donde se aplican todas las reglas de gestión.[1]

2.2 Análisis Comparativo: NVC frente a los Estándares de la Industria

El término "NVC" no es estándar en la literatura canónica de ingeniería de software, lo que genera cierta ambigüedad y debate en la comunidad técnica.

2.2.1 Divergencia Semántica con MVC

El patrón MVC (Modelo-Vista-Controlador) tiene sus raíces en Smalltalk y ha sido la base de frameworks modernos como Spring MVC, ASP.NET MVC y Cocoa.2 En MVC puro, el "Modelo" notifica a la "Vista" de los cambios. La variante NVC descrita en el proyecto parece acercarse más al patrón MVP (Model-View-Presenter) o a una arquitectura de capas estricta, donde la comunicación es lineal: Vista -> Controlador -> Negocio.
La crítica en foros especializados sugiere que inventar acrónimos como NVC puede ser contraproducente. Usuarios en comunidades de programación han señalado irónicamente que "Reddit no está diseñado para NVC", evidenciando que el término carece de reconocimiento universal y podría confundirse con otros conceptos.2

2.2.2 Colisión Terminológica: Compiladores NVIDIA y AngularJS

Una investigación profunda revela que el acrónimo NVC tiene significados radicalmente diferentes en otros contextos de computación de alto rendimiento. Específicamente, nvc++ es un compilador de C++ para GPUs de NVIDIA, utilizado en paralelismo estándar y computación heterogénea.[8] En análisis de rendimiento de la JVM (Java Virtual Machine), NVC puede referirse a "Non-Virtual Calls" (Llamadas no virtuales), una métrica de optimización de código.[10] Incluso en discusiones sobre frameworks web antiguos como AngularJS, a veces se ha usado erróneamente NVC para describir la triada modelo-vista-controlador.[11]

Esta sobrecarga semántica implica que el uso de "NVC" en el proyecto analizado es probablemente una decisión pedagógica local o una interpretación idiosincrásica del autor para resaltar la importancia de la "Lógica de Negocio", distanciándose de la abstracción a veces vaga de "Modelo". Sin embargo, para un

entorno profesional, sería recomendable adherirse a la terminología estándar (MVC o Arquitectura de Capas) para evitar confusiones con herramientas de compilación de hardware o métricas de la JVM.

2.2.3 La Evolución hacia Componentes

Es relevante notar que la industria web moderna, liderada por frameworks como React y Next.js, se está alejando de MVC/NVC hacia arquitecturas orientadas a componentes y flujos de datos unidireccionales (como FLUX), donde la mutabilidad del estado se gestiona de manera diferente.[12] Sin embargo, para una aplicación de escritorio Java Swing, que es inherentemente imperativa y orientada a eventos con estado, la estructura de Controlador y Servicio propuesta sigue siendo una práctica válida y robusta.

3. El Patrón Singleton: Mecánica, Implementación y Controversia

El documento identifica al Patrón Singleton como el mecanismo clave para la "Eficiencia de Memoria" y la "Integridad de Datos Garantizada".[1] Este patrón es uno de los más conocidos, pero también uno de los más debatidos en la ingeniería de software debido a sus implicaciones en la testabilidad y la concurrencia.

3.1 Fundamentos Teóricos y Mecánica en Java

El Singleton es un patrón de diseño creacional cuya intención formal es asegurar que una clase tenga una única instancia y proporcionar un punto de acceso global a ella. En el contexto del proyecto, el EstudianteRepository actúa como este punto único de verdad.

La implementación canónica en Java requiere tres elementos estructurales:

1. **Constructor Privado:** private EstudianteRepository() {}. Esto es fundamental para bloquear la instanciación externa mediante el operador new. Si el constructor fuera público, cualquier controlador podría crear su propia lista de estudiantes, rompiendo la persistencia compartida.[1]
2. **Variable Estática Privada:** private static EstudianteRepository instancia;. Esta variable reside en una zona específica de la memoria de la JVM (anteriormente PermGen, ahora Metaspace para la clase y Heap para el objeto estático) y sobrevive durante todo el ciclo de vida de la clase cargada.
3. **Método de Acceso Público Estático:** public static EstudianteRepository getInstance(). Este método encapsula la lógica de creación, generalmente usando "inicialización perezosa" (*lazy initialization*), donde la instancia solo se crea la primera vez que se solicita.[3]

3.2 Validación de la Unicidad

El proyecto propone una prueba empírica para validar el patrón: colocar un mensaje de impresión (System.out.println) en el constructor. Si el mensaje aparece una sola vez en la consola, se confirma que el objeto se creó una única vez, independientemente de cuántas veces se acceda a él desde distintas vistas.[1] Esta validación confirma que la memoria es compartida, resolviendo el problema de la pérdida de datos al

navegar entre pantallas.

3.3 Análisis Crítico: El Desafío de la Concurrencia (Thread Safety)

Uno de los puntos ciegos más peligrosos al implementar Singleton en Java es la seguridad en entornos multihilo. Aunque Swing es *single-threaded* en lo que respecta al despacho de eventos (EDT), una aplicación robusta podría tener hilos de fondo procesando datos.

Si se utiliza una implementación ingenua:

```
public static EstudianteRepository getInstance() {
    if (instancia == null) {
        instancia = new EstudianteRepository();
    }
    return instancia;
}
```

Se introduce una condición de carrera (*race condition*). Si dos hilos (por ejemplo, el hilo de la UI y un hilo de carga de datos) invocan getInstance() simultáneamente cuando instancia es null, ambos podrían entrar al bloque if, creando dos instancias distintas y divergentes del repositorio. Esto violaría el principio fundamental del Singleton y causaría inconsistencia de datos.[3]

3.3.1 Estrategias de Mitigación y Mejores Prácticas

Para elevar el nivel del proyecto a un estándar profesional, se deben considerar implementaciones seguras:

- **Sincronización:** Añadir synchronized al método getInstance(). Esto garantiza la seguridad pero introduce un cuello de botella de rendimiento, ya que cada acceso al repositorio debe esperar a obtener el bloqueo (*lock*), lo cual es costoso.[3]
- **Bloqueo de Doble Verificación (Double-Checked Locking):** Esta técnica verifica si la instancia es nula, luego sincroniza, y vuelve a verificar. Requiere que la variable instancia sea declarada como volatile para asegurar que la escritura en memoria sea visible inmediatamente para otros hilos, evitando problemas de reordenamiento de instrucciones por parte del compilador JIT.[7]
- **Clase Holder (Bill Pugh Singleton):** Utiliza una clase interna estática privada que contiene la instancia. La clase interna no se carga hasta que se llama a getInstance(), y la JVM garantiza la seguridad de hilos durante la carga de clases. Es la opción recomendada para Singletons "perezosos" por su elegancia y rendimiento.[3]
- **Enum Singleton:** Joshua Bloch, autor de *Effective Java*, recomienda usar un tipo enum de un solo elemento (INSTANCE). Esto maneja automáticamente la serialización y previene la creación de múltiples instancias incluso mediante reflexión compleja, siendo la forma más robusta de Singleton

en Java moderno.[3]

3.4 El Debate del Anti-Patrón y la Testabilidad

El informe del proyecto exalta las virtudes del Singleton, pero una revisión equilibrada debe mencionar sus detractores. En círculos de arquitectura de software avanzado, el Singleton a menudo se etiqueta como un "anti-patrón" debido a que introduce un **estado global oculto**.

- **Acoplamiento:** Las clases que usan el Singleton (como EstudianteController) están fuertemente acopladas a la implementación concreta del repositorio. No se puede sustituir fácilmente por una versión de prueba (*mock*) sin herramientas complejas de reflexión.
- **Dificultad en Pruebas Unitarias:** Si el repositorio mantiene estado (la lista de estudiantes), el orden de ejecución de los tests importa. Un test que agrega un estudiante afectará al siguiente test que espera una lista vacía, violando el principio de aislamiento de las pruebas.[4]

A pesar de estas críticas, para el alcance del proyecto descrito (una aplicación Swing autocontenida sin frameworks de inyección de dependencias como Spring), el Singleton es una solución pragmática, eficiente en memoria y suficiente para cumplir los requisitos funcionales.[1]

---

4. Gestión de Memoria y Estrategias de Persistencia

El documento destaca la "Eficiencia de Memoria" como una ventaja clave, argumentando que al evitar la creación redundante de objetos se optimiza la RAM.[1] Este argumento requiere un análisis técnico sobre cómo la JVM gestiona estas estructuras.

4.1 Ciclo de Vida en el Heap de Java

Cuando se instancia el Singleton EstudianteRepository, este objeto y su colección interna (probablemente un ArrayList o HashMap) se alojan en el Heap.

- **Old Generation (Tenured):** Dado que el Singleton es estático y vive durante toda la aplicación, eventualmente será promovido por el *Garbage Collector* desde la zona "Eden" y "Survivor" hacia la "Old Generation". Esto es eficiente porque reduce la frecuencia con la que el GC debe examinar este objeto, asumiendo que vivirá para siempre.
- **Riesgos de Fugas de Memoria:** Si la lista de estudiantes crece indefinidamente sin límites, se corre el riesgo de un OutOfMemoryError. En una base de datos real, los datos están en disco y se traen a memoria (paginación) solo cuando se necesitan. En este modelo puramente en memoria, la escalabilidad está limitada por la RAM física asignada a la JVM (-Xmx).

4.2 Comparativa de Estructuras de Datos

La eficiencia del repositorio depende críticamente de la estructura de datos subyacente:

- **ArrayList:** Acceso rápido por índice, pero búsqueda lenta O(n) para encontrar un estudiante por ID, y eliminación costosa O(n) debido al desplazamiento de elementos.
- **HashMap:** Acceso y búsqueda casi instantáneos O(1) usando el ID del estudiante como clave. Para un sistema CRUD en memoria optimizado, se recomendaría migrar de una lista simple a un mapa para mejorar el rendimiento de las operaciones de lectura y actualización.

4.3 La Limitación de la Volatilidad

La principal debilidad no abordada explícitamente como un problema en el documento es que la memoria RAM es volátil. El sistema cumple el requisito de persistencia durante la ejecución, pero falla en la persistencia entre ejecuciones. Si la aplicación se cierra o falla (crash), todos los datos se pierden.
Una evolución natural de esta arquitectura sería implementar el patrón DAO (Data Access Object) dentro del Singleton para serializar la lista a un archivo JSON o binario (Serializable) al cerrar la aplicación, y deserializarla al iniciar. Esto convertiría la solución en una base de datos persistente ligera.

5. Recomendaciones de Implementación y Refactorización

5.1 Escalabilidad de la Estructura de Datos

Para optimizar la búsqueda y actualización de estudiantes, se debe reemplazar la List<Estudiante> por un Map<String, Estudiante> (donde la clave es el ID o matrícula).

Operación	Complejidad con ArrayList	Complejidad con HashMap	Beneficio
Buscar (Leer)	O(n) - Lineal	O(1) - Constante	Acceso inmediato independiente del tamaño de la lista.
Insertar (Crear)	O(1) - Constante	O(1) - Constante	Similar rendimiento.
Eliminar	O(n) - Lineal (requiere búsqueda y desplazamiento)	O(1) - Constante	Mejora drástica en grandes volúmenes de datos.

7. Actividad Práctica

Criterio	Arquitectura NVC (Negocio - Vista - Control)	Patrón Singleton
¿Qué problema resuelve?	Resuelve la necesidad de una gestión de datos y lógica clara, organizando el proyecto en capas bien definidas para evitar código desordenado.	Resuelve el problema de la persistencia de datos en memoria; evita que la lista de estudiantes se reinicie o pierda al navegar entre vistas o interactuar con distintos controladores.
¿En qué capa se utiliza?	Es la estructura global del sistema que integra tres capas: **Negocio** (Service), **Vista** (UI/Swing) y **Control** (Controller).	Se implementa específicamente en la **Capa de Datos (Repository)**, dentro de la clase EstudianteRepository.
¿Cómo influye en el mantenimiento?	Facilita enormemente el mantenimiento gracias a la separación de responsabilidades. Un cambio en la interfaz (Vista) no rompe la lógica de negocio ni la gestión de datos subyacente.	Influye en la eficiencia y el uso de recursos. Al evitar la creación redundante de objetos, optimiza el uso de la memoria RAM, haciendo el sistema más eficiente.
¿Cómo evita fallas de diseño?	Evita el acoplamiento excesivo manteniendo la lógica de negocio independiente de la tecnología de presentación. Mantiene la vista "tonta" (sin lógica), asegurando que solo se encargue de presentar información.	Garantiza la integridad de los datos asegurando la unicidad de la instancia. Todos los controladores acceden a la misma y única lista de información, evitando inconsistencias en los datos.

8. Conclusión

El proyecto documentado en *Implementación del Patrón Singleton y Arquitectura NVC en Java* representa un ejercicio arquitectónico sólido que aborda con eficacia el problema de la gestión de estado en aplicaciones de escritorio Swing. A través de la deconstrucción de sus componentes, se evidencia que la arquitectura **NVC**, aunque terminológicamente no estándar y propensa a colisiones semánticas con tecnologías de compilación de NVIDIA, ofrece una estructura lógica clara (Negocio, Vista, Control) que promueve la separación de preocupaciones y la modularidad.[1]

La aplicación del **Patrón Singleton** se valida como la estrategia óptima para la persistencia en memoria bajo las restricciones del proyecto, proporcionando un acceso global unificado a los datos. Sin embargo, el análisis experto revela que la implementación ingenua sugerida carece de protecciones contra condiciones de carrera, lo cual es un riesgo latente en cualquier aplicación Java moderna. La adopción de implementaciones robustas como el *Bill Pugh Singleton* o *Enum Singleton* es imperativa para garantizar la integridad profesional del software.[3]

En síntesis, el sistema propuesto es viable y didácticamente valioso. Con las refactorizaciones sugeridas en seguridad de hilos y estructuras de datos, puede evolucionar de un prototipo académico a una aplicación robusta, capaz de soportar las exigencias de un entorno de producción real, manteniendo la coherencia de sus datos y la eficiencia de sus recursos.

Bibliografía

1. Implementación del Patrón Singleton y Arquitectura NVC en Java _ Gamma.pdf
2. A great explanation of MVC : r/programming - Reddit, fecha de acceso: noviembre 25, 2025, https://www.reddit.com/r/programming/comments/7x086/a_great_explanation_of_mvc/
3. How to Implement a Thread-Safe Singleton in Java? | Baeldung, fecha de acceso: noviembre 25, 2025, https://www.baeldung.com/java-implement-thread-safe-singleton
4. What are drawbacks or disadvantages of singleton pattern? [closed] - Stack Overflow, fecha de acceso: noviembre 25, 2025, https://stackoverflow.com/questions/137975/what-are-drawbacks-or-disadvantages-of-singleton-pattern
5. Thread Safety in Java Singleton Classes | DigitalOcean, fecha de acceso: noviembre 25, 2025, https://www.digitalocean.com/community/tutorials/thread-safety-in-java-singleton-classes
6. Standard C++ Parallelism using nvc++ is slow - Stack Overflow, fecha de acceso: noviembre 25, 2025, https://stackoverflow.com/questions/71714200/standard-c-parallelism-using-nvc-is-slow
7. (PDF) Algoritmo de mínima cobertura de Redes de Petri vectorizado y monitor concurrente de alto rendimiento - ResearchGate, fecha de acceso: noviembre 25, 2025, https://www.researchgate.net/publication/377386078_Algoritmo_de_minima_cobertura_de_Redes_de_Petri_vectorizado_y_monitor_concurrente_de_alto_rendimiento
8. Novel Approaches to Systematically Evaluating and Constructing Call Graphs for Java Software - Teamscale, fecha de acceso: noviembre 25, 2025, https://teamscale.com/hubfs/26978363/Publications/2021-novel-approaches-to-systematically-evaluating-and-constructing-call-graphs-for-java-software.pdf
9. AngularJS Interview Questions for Experienced - Cloud Foundation, fecha de acceso: noviembre 25, 2025, https://cloudfoundation.com/blog/angularjs-interview-questions/
10. Is it necessary to use some sort of MVC in NextJS or waste of time? - Reddit, fecha de acceso: noviembre 25, 2025, https://www.reddit.com/r/nextjs/comments/1mavbo3/is_it_necessary_to_use_some_sort_of_mvc_in_nextjs/
11. Thread-Safe Singleton Design Pattern using Java | by Sanjjushri Varshini R | Medium, fecha de acceso: noviembre 25, 2025, https://medium.com/@Sanjjushri/thread-safe-singleton-design-pattern-using-java-4ac6009c18bb
12. Thread safety in Singleton - java - Stack Overflow, fecha de acceso: noviembre 25, 2025, https://stackoverflow.com/questions/2912281/thread-safety-in-singleton
13. Mastering the Singleton Design Pattern in Java – A Complete Guide - DEV Community, fecha de acceso: noviembre 25, 2025, https://dev.to/zeeshanali0704/mastering-the-singleton-design-pattern-in-java-a-complete-guide-13nn
14. Singleton classes and memory usage - Stack Overflow, fecha de acceso: noviembre 25, 2025, https://stackoverflow.com/questions/8390073/singleton-classes-and-memory-usage
15. Performance and memory impact of the @Singleton annotation in Dagger : r/androiddev,

fecha de acceso: noviembre 25, 2025, https://www.reddit.com/r/androiddev/comments/12tyil8/performance_and_memory_impact_of_the_singleton/