Taller 4

Nombre del estudiante:	Matias Lugmaña, Camilo Orrico, Denise Rea, Julio Viche
Docente:	Mgt. Jenny Alexandra Ruiz Robalino
Fecha:	27/11/2025
NRC:	27835

1. Objetivo General

El objetivo de este informe es demostrar la implementación práctica de los patrones de diseño creacionales **Factory Method** y **Builder** dentro de una arquitectura **MVC (Modelo-Vista-Controlador)**. Se busca desacoplar la lógica de creación de objetos de la lógica de negocio, asignando responsabilidades específicas a cada patrón para resolver problemas de validación estricta y construcción compleja, manteniendo al mismo tiempo el patrón **Singleton** para la persistencia de datos.

MVC + FACTORY

2. ¿Por qué escogimos Factory?

Para el primer enfoque, seleccionamos el patrón **Factory Method (Simple Factory)** para resolver el problema de la **Integridad de Datos** al momento de la creación.

- **Problema Detectado:** Permitir que el `Service` o el `Controller` instancien objetos directamente (`new Estudiante(...)`) es riesgoso, ya que se pueden crear objetos con datos inválidos (cédulas falsas, edades negativas) dispersando la lógica de validación por todo el código.
- **Solución Factory:** Centralizamos la creación en una sola clase "Fábrica". Su responsabilidad es actuar como un **filtro estricto**: si los datos no cumplen con las reglas de negocio (como el algoritmo de cédula ecuatoriana), el objeto **jamás se crea**.
- **Valor Agregado:** Implementamos un mecanismo `ThreadLocal` para permitir pruebas unitarias aisladas ("Modo Test"), algo que un constructor simple no permite.

3. Estructura del Proyecto (Factory)

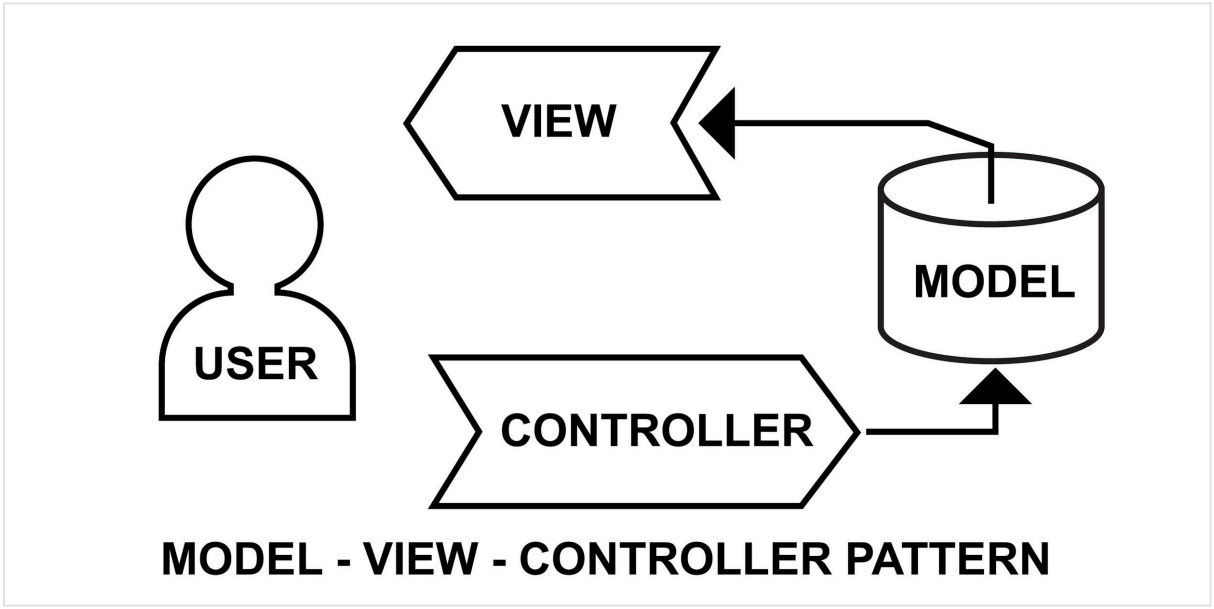
La arquitectura mantiene las capas MVC, pero introduce un paquete dedicado a la fabricación de objetos en la capa de Datos.

```
MVC
└── src
    └── ec
        └── edu
            └── espe
                ├── controller
                │   └── EstudianteController.java  (Coordina Vista <-> Negocio)
                │
                ├── datos
                │   ├── factory
                │   │   └── EstudianteFactory.java (Método estático 'crearEstudiante')
                │   │
                │   ├── model
                │   │   └── Estudiante.java
                │   │
                │   └── repository
                │       └── EstudianteRepository.java
                │
                ├── negocio
                │   └── EstudianteService.java     (Llama a EstudianteFactory.crear...)
                │
                ├── vista
                │   └── EstudianteUI.java          (JFrame / Ventana)
                │
                └── Main.java                      (Punto de entrada)
```

4. Partes Clave del Código

A continuación, se muestra cómo la Fábrica intercepta la creación.

A. La Fábrica (EstudianteFactory.java)




```
public class EstudianteFactory {
    // ThreadLocal garantiza aislamiento en pruebas concurrentes
    private static final ThreadLocal<Boolean> modoTest =
ThreadLocal.withInitial(() -> false);

    public static Estudiante crearEstudiante(String id, String nombres,
String edadStr) throws Exception {
        // 1. Validaciones "Fail-Fast" (Fallo Rápido)
        if (id == null || id.trim().isEmpty()) throw new Exception("La
cédula es obligatoria.");

        // 2. Validación de Cédula (Algoritmo Módulo 10)
        // Se ejecuta solo si NO estamos en modo test
        if (!modoTest.get() && !validarCedulaEcuatoriana(id.trim())) {
            throw new Exception("Error: La cédula ingresada no es válida.");
        }

        // 3. Conversión segura de tipos
        int edad = Integer.parseInt(edadStr.trim());

        // 4. Retorno del objeto solo si todo es válido
        return new Estudiante(id.trim(), nombres.trim(), edad);
    }
}
```

B. El Consumo en Negocio (EstudianteService.java)

Java

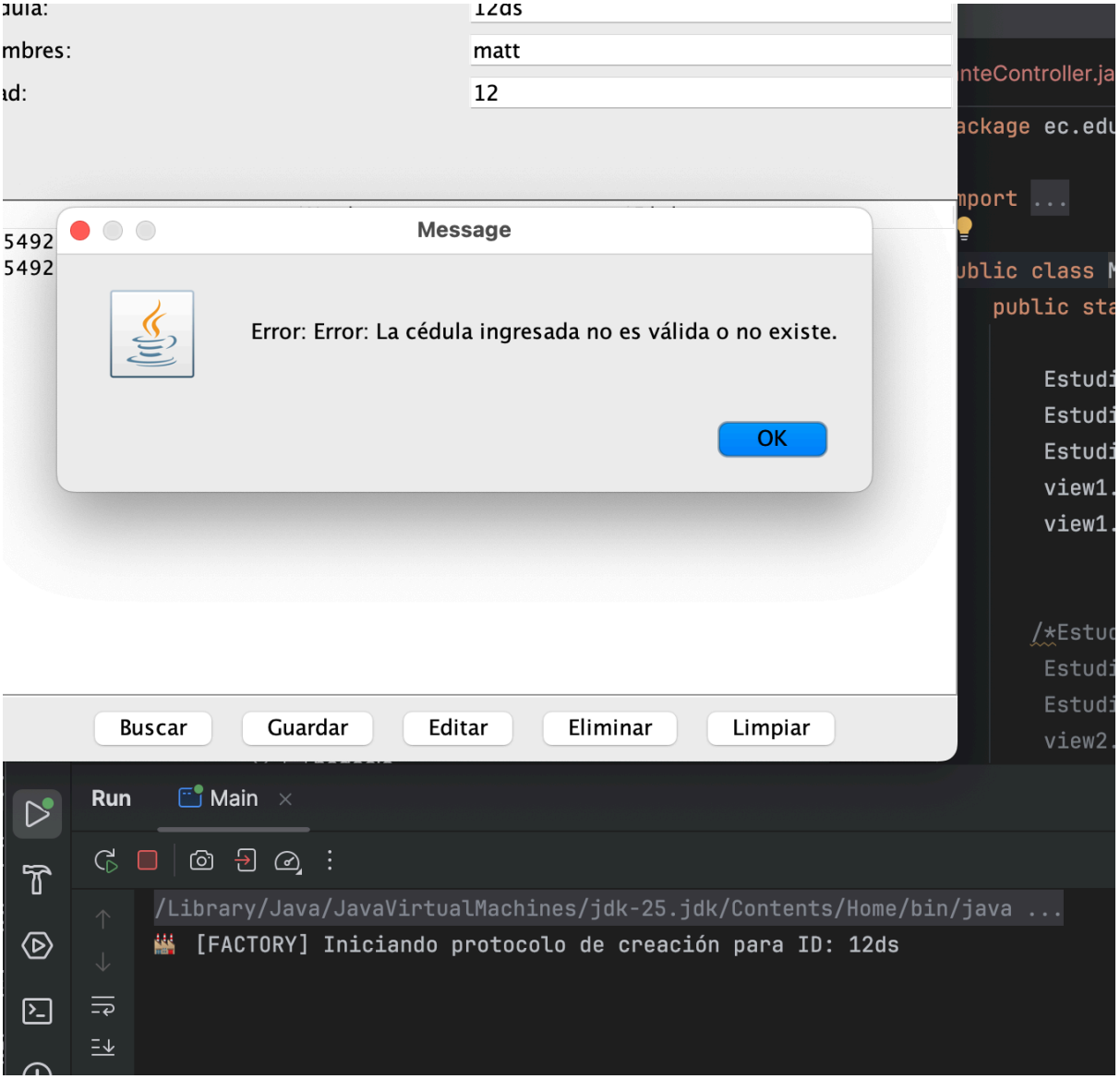
```
public void guardar(String id, String nombre, String edadStr) throws
Exception {
    // Delegamos la creación a la Factory. El servicio NO hace 'new'.
    Estudiante nuevoEstudiante = EstudianteFactory.crearEstudiante(id,
nombre, edadStr);

    // El Singleton sigue gestionando el almacenamiento
    repository.crear(nuevoEstudiante);
}
```

5. Ejecución y Evidencia

Al ejecutar el programa, el funcionamiento del patrón es evidente cuando se intentan violar las reglas:

1. **Acción:** El usuario ingresa la cédula "123" y da clic en Guardar.
2. **Resultado:** Se muestra un `JOptionPane` con el error: *"Error: La cédula ingresada no es válida"*.
3. **Análisis:** El error fue lanzado por la **Factory** antes de instanciar el objeto. El Repositorio nunca recibió datos basura.



MVC + BUILDER

6. ¿Por qué escogimos Builder?

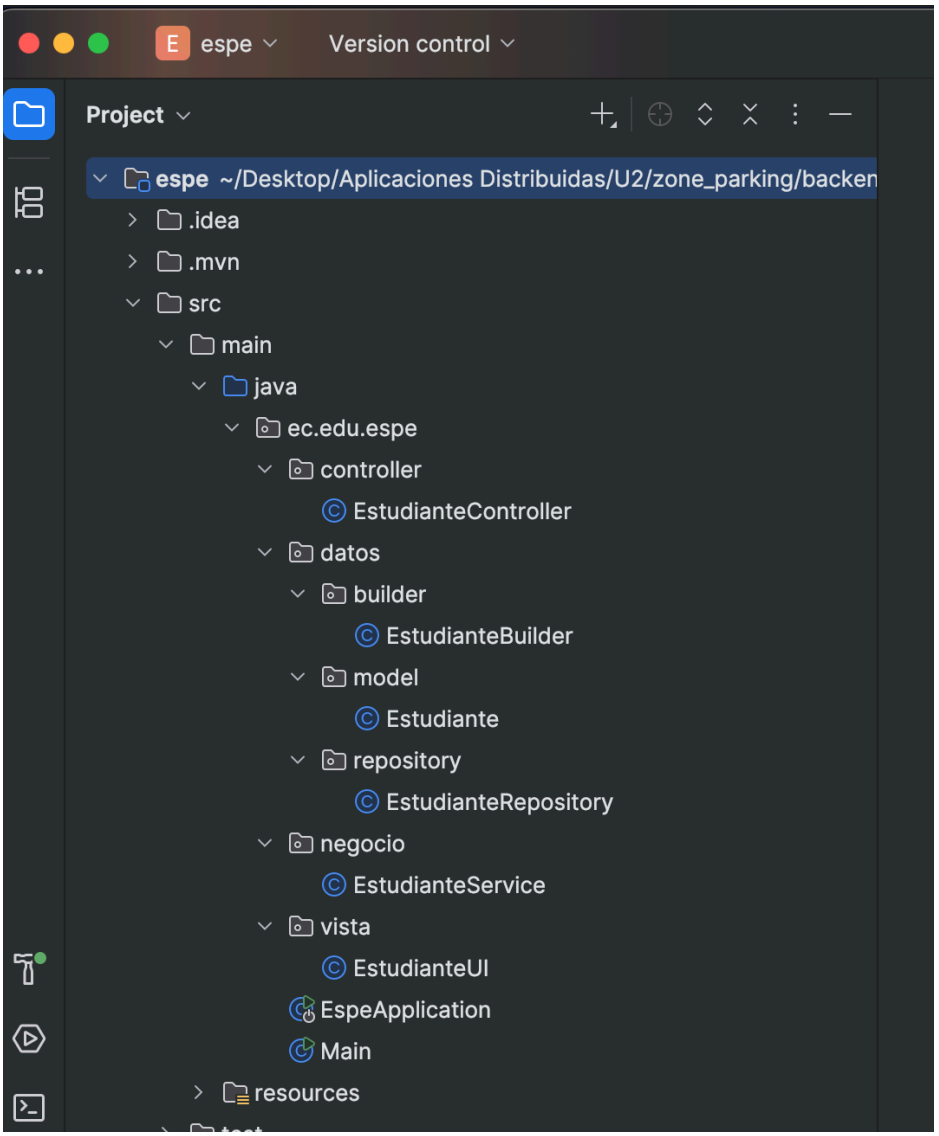
Para el segundo enfoque, utilizamos el patrón **Builder** para resolver el problema de la **Complejidad en la Construcción** y la **Legibilidad**.

- **Problema Detectado:** Cuando necesitamos reconstruir objetos desde un archivo de texto (CSV) o crear objetos con configuraciones opcionales, los constructores tradicionales se vuelven confusos y rígidos.
- **Solución Builder:** Implementamos un constructor paso a paso con una **Interfaz Fluida** (Fluent Interface). Esto permite configurar el objeto propiedad por propiedad, transformar datos (ej. convertir nombres a mayúsculas) y validar la consistencia solo al final del proceso (`.build()`).

- **Valor Agregado:** Facilita la lectura del código ("se lee como una frase") y permite la inmutabilidad parcial durante la construcción.

7. Estructura del Proyecto (Builder)

Se reemplaza el paquete factory por builder en la capa de Datos.



MVC_Builder

└── src

└── ec

└── edu

```
└── espe
    ├── controller
    │   └── EstudianteController.java  (Igual que en el otro proyecto)
    │
    ├── datos
    │   ├── builder  <-- [DIFERENCIA CLAVE]
    │   │   └── EstudianteBuilder.java (Clase con .withId().withNombre().build())
    │   │
    │   ├── model
    │   │   └── Estudiante.java
    │   │
    │   └── repository
    │       └── EstudianteRepository.java )
    │
    ├── negocio
    │   └── EstudianteService.java     (Llama a new EstudianteBuilder()...)
    │
    ├── vista
    │   └── EstudianteUI.java          (Igual que en el otro proyecto)
    │
    └── Main.java                      (Punto de entrada)
```

8. Partes Clave del Código

A. El Constructor (`EstudianteBuilder.java`)

```java
public class EstudianteBuilder {
    private String id;
    private String nombres;
    private int edad;

    // Métodos encadenados (Fluent Interface)
    public EstudianteBuilder withId(String id) {
        this.id = id;
        System.out.println("[Builder] ID configurado: " + id);
        return this;
    }

    public EstudianteBuilder withNombres(String nombres) {
        // Lógica de transformación incluida: Convertir a Mayúsculas
        this.nombres = nombres != null ? nombres.toUpperCase() : null;
        return this;
    }

    // Método final de construcción
    public Estudiante build() throws Exception {
        if (id == null) throw new Exception("Falta ID");
        return new Estudiante(id, nombres, edad);
    }
}
```

B. El Consumo en Negocio (EstudianteService.java)

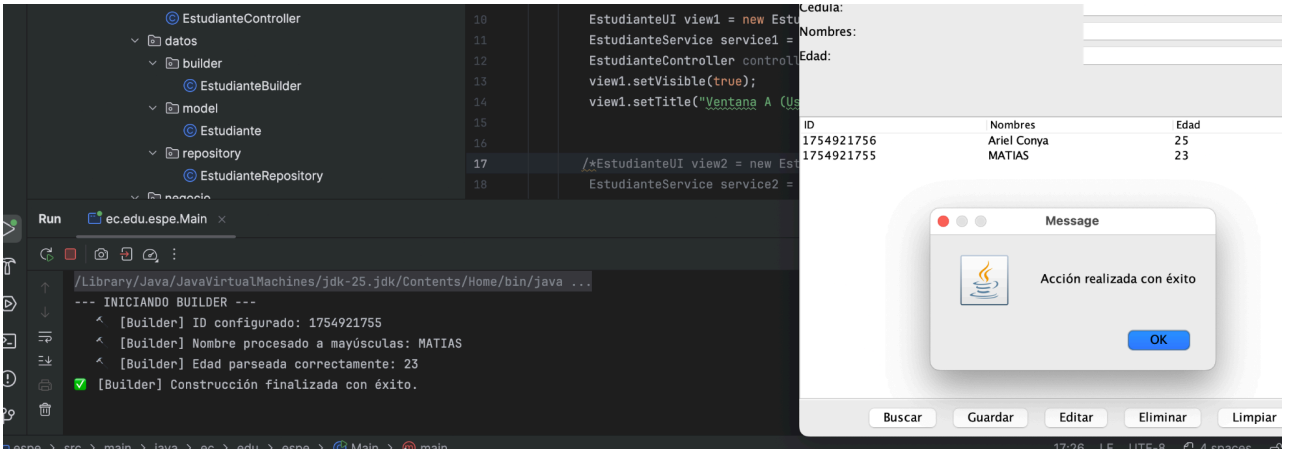
```java
public void guardar(String id, String nombre, String edadStr) throws
Exception {
    // Construcción paso a paso y legible
    Estudiante nuevoEstudiante = new EstudianteBuilder()
                                    .withId(id)
                                    .withNombres(nombre)
                                    .withEdadStr(edadStr)
                                    .build(); // Aquí nace el objeto

    repository.crear(nuevoEstudiante);
}
```

9. Ejecución y Evidencia

La evidencia del Builder se observa mejor en la consola de depuración (Logs):

1. **Acción:** El usuario ingresa el nombre "matias" (minúsculas) y guarda.
2. **Resultado en Consola:**



[Builder] ID configurado: 1712345678
[Builder] Nombre procesado a mayúsculas: MATIAS
[Builder] Edad parseada correctamente: 23
[Builder] Construcción finalizada.

3. **Análisis:** Se verifica visualmente cómo el objeto fue mutando y transformándose paso a paso antes de existir formalmente.

CONCLUSIONES

1. Mientras el patrón **Singleton** (usado en el Repositorio) es insustituible para garantizar una única fuente de verdad y persistencia compartida, carece de flexibilidad para la creación de objetos. **Factory** y **Builder** llenan este vacío, encargándose exclusivamente de la lógica de instanciación.
2. El **Factory** demostró ser superior para garantizar la integridad de datos mediante validaciones estrictas y centralizadas ("Gatekeeper").El **Builder** demostró ser superior para la legibilidad del código y la transformación de datos durante la construcción paso a paso.
3. Ambos patrones mejoran significativamente la capacidad de realizar pruebas unitarias en comparación con el código acoplado, ya que permiten aislar la lógica de creación.

RECOMENDACIONES

1. **Uso Híbrido:** Se recomienda utilizar una arquitectura que combine estos patrones: Singleton para los Servicios/Repositorios y Factory/Builder para las Entidades de Dominio.
2. **Criterio de Selección:** Utilizar **Factory** cuando se requiera validación estándar y estricta (ej. crear un usuario nuevo). Utilizar **Builder** cuando el objeto sea complejo, tenga muchos parámetros opcionales o requiera reconstrucción desde fuentes externas (ej. leer un archivo CSV).

Aplicación en el Proyecto "Karioz Mix"

Para el proyecto de emprendimiento **Karioz Mix** (Venta de Frutos Secos), la aplicación de estos patrones sería la siguiente:

1. **Patrón Builder (Para "Mix Personalizados"):**
    - **Caso de Uso:** Un cliente desea armar su propio mix.
    - **Implementación:** `MixBuilder`. El cliente agrega ingredientes paso a paso: `.agregarNueces()`, `.sinSal()`, `.empaqueVidrio()`, `.build()`. Esto es ideal porque cada producto final es único y complejo.
2. **Patrón Factory (Para "Productos de Estantería"):**
    - **Caso de Uso:** Venta de productos pre-empacados estándar (ej. "Bolsa de Maní 50g").
    - **Implementación:** `ProductoFactory.crearManiSalado()`. No requiere configuración paso a paso, solo entregar el producto estándar validado y listo para facturar.
3. **Patrón Singleton (Para "Inventario"):**
    - **Caso de Uso:** Control de Stock de ingredientes.
    - **Implementación:** `InventarioRepository`. Garantiza que si se vende 1kg de nueces en un Mix Personalizado (Builder) o en una Bolsa Estándar (Factory), se descuente del **mismo** almacén central.

Referencias

- Bloch, J. (2018). *Effective Java* (3.a ed.). Addison-Wesley Professional. https://www.oreilly.com/library/view/effective-java-3rd/9780134686097/
- Freeman, E., & Freeman, E. (2020). *Head First Design Patterns: Building Extensible and Maintainable Object-Oriented Software* (2.a ed.). O'Reilly Media.https://www.oreilly.com/library/view/head-first-design/9781492077992/
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional. https://books.google.com/books/about/Design_Patterns.html?id=6oHuKQe3TjQC
- Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., & Lea, D. (2006). *Java Concurrency in Practice*. Addison-Wesley Professional. https://www.oreilly.com/library/view/java-concurrency-in/0321349601/
- Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*.

Prentice Hall.

https://books.google.com/books/about/Clean_Code.html?id=dwSfGQAACAAJ

- Oracle. (2023). *Java Documentation: When is a Singleton not a Singleton?*. Oracle Technical

Resources.https://www.oracle.com/technical-resources/articles/java/singleton.html