Taller 01: Refactorización Arquitectónica

Aplicación de Patrones en Proyectos Reales

Prof. Nicolás Ramírez Vélez - Arquitectura de Software 2026

1. Descripción del Taller

En esta sesión, los equipos de trabajo aplicarán los conceptos de **Arquitectura Limpia, SOLID y Patrones de Diseño** sobre sus proyectos de curso.

El objetivo no es refactorizar toda la aplicación, sino seleccionar **UNA (1) funcionalidad crítica de negocio** (ej: Crear Reserva, Procesar Matrícula, Agendar Cita) y transformarla de un enfoque monolítico a una arquitectura desacoplada y profesional.

2. Instrucciones de Ejecución

2.1. 1. Selección del Candidato (15 min)

Identifiquen el flujo más complejo de su proyecto actual. Ese que tiene muchos 'if/else', validaciones y llamadas a base de datos en la vista. Ese será su objetivo.

2.2. 2. Implementación de Capas (70 min)

Aplique la siguiente estructura para garantizar el cumplimiento de SOLID:

1. **Capa de Interfaz (Django View):**
   - Utilice una `View` basada en clase (CBV).
   - Su única responsabilidad es capturar los datos del `request` y llamar al **Service**.
2. **Capa de Aplicación (Service Layer):**
   - Cree un archivo `services.py`.
   - Aquí reside el *algoritmo* del negocio. El servicio orquestará al Builder y a la Factory.
3. **Capa de Dominio (Patrones Creacionales):**
   - **Builder:** Implemente un Builder para construir el modelo principal, asegurando que el objeto sea válido antes de llamar al método `.save()`.
   - **Factory:** Implemente una Factory para instanciar una dependencia externa o un servicio de apoyo (ej: un enviador de correos o un calculador de impuestos).

2.3. 3. Documentación en Wiki (30 min)

Actualicen la Wiki de su repositorio con una página titulada **Implementación del Patrón Creacional**. Deben explicar con un diagrama breve cómo interactúa la Vista con el Servicio y el Builder.

—

3. Entregables y Rúbrica

La entrega se realizará mediante la URL del repositorio de GitHub. La calificación se basará en la evidencia de código y la documentación en la Wiki.

Componente	Criterio de Evaluación	Puntos
Service Layer y SOLID	La vista tiene menos de 15 líneas. La lógica reside en el servicio. Se aplica Inyección de Dependencias.	**1.5**
Patrón Factory	Existe una Factory en `infra/`. Se demuestra el cambio de comportamiento usando variables de entorno (ej: MOCK vs REAL).	**1.0**
Patrón Builder	Se implementa un Builder en `domain/` para la creación de objetos complejos, garantizando su validez (Fluent Interface).	**1.0**
Wiki del Repo	Documentación técnica clara. Diagrama de clases simple y justificación de las decisiones de diseño tomadas.	**1.0**
Git Flow	Commits semánticos y ordenados. Trabajo colaborativo evidente en el historial.	**0.5**

¡Bonificación por Tiempo! (+0.5)

Los equipos que realicen el **Push final** y suban el enlace antes de finalizar la sesión presencial , recibirán una bonificación de +0.5 sobre la nota final del taller.

4. Guía para la Wiki (Ejemplo)

Su documentación en GitHub Wiki debe verse similar a esto:

Módulo: Procesamiento de Reservas

Problema: La creación de reservas implicaba validar fechas, calcular tarifas dinámicas y enviar correos, todo mezclado en la vista.

Solución Arquitectónica:

- **Service Layer:** Se creó `ReservaService` para orquestar el flujo.
- **Factory:** `NotificadorFactory` decide si enviar emails reales (SendGrid) o imprimir en consola (Dev), dependiendo de `ENV_TYPE`.
- **Builder:** `ReservaBuilder` permite construir la reserva paso a paso, validando disponibilidad antes de guardar.

Snippet Clave:

```
# services.py
def crear_reserva(self, datos):
    reserva = (ReservaBuilder()
               .para_usuario(datos.user)
               .en_fechas(datos.inicio, datos.fin)
               .build())
    self.notificador.enviar_confirmacion(reserva)
```