Metodología de Sistemas II

Clase 1

Presentación

Jeremías Fassi

2025

Presentación

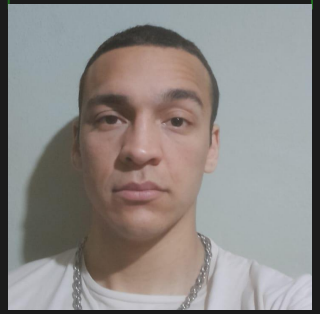
Jeremías Fassi

Ingeniero en Sistemas de Computación

Actualmente trabajo en una empresa bahiense. Estoy dentro del área de infraestructura (IT) en AWS, donde me encargo de gestionar servidores, repositorios, permisos, despliegues, monitoreo, etc. Además cumplo el rol de desarrollador backend para algunos proyectos.
Y soy profesor de 2 materias en la TUP. Arquitectura y Sistemas Operativos y Metodología de Sistemas II

Javier Kinter

Técnico Universitario en Programación



Profesor de programación en nivel primario y secundario, donde doy clases desde lógica de programación básica hasta desarrollo web.
Profesor Adjunto de 1 materia en la TUP: Gestión de Desarrollo de Software; además de
asistente de cátedra de otras 2 materias: Arquitectura y Sistemas Operativos y Metodología de Sistemas II.

Programa de la materia

Objetivos

1. Reconocer patrones de diseño de desarrollo de software.
2. Implementar buenas prácticas en el desarrollo de software.
3. Aplicar mejora continua durante todo el ciclo de desarrollo de software.
4. Utilizar herramientas de verificación y validación en el desarrollo de software.

Contenidos mínimos

- Introducción a los patrones de diseño y desarrollo de software.
- Buenas prácticas en el proceso de implementación de software.
- Técnicas de optimización del ciclo de desarrollo de software.
- Herramientas de verificación y validación en el desarrollo de software.
- Herramientas de repositorios de software.

Importancia en la carrera

Esta materia no se enfoca en aprender un lenguaje nuevo, sino en **mejorar la manera en la que desarrollan software**.

El objetivo es que al finalizar el curso puedan:

- Reconocer problemas comunes y aplicar un patrón adecuado.
- Escribir código limpio, legible y mantenible.
- Usar herramientas modernas de repositorios y testing como lo harían en un entorno profesional.

Importancia en la carrera

Implicancias:

- Eleva el nivel de calidad y mantenibilidad de los proyectos.
- Los prepara para trabajar con estándares de la industria actual.
- Fomenta el trabajo en equipo y el uso de herramientas colaborativas.

Metodología de trabajo

- 2 horas de teoría: conceptos clave + ejemplos.
- 2 horas de práctica: avance del proyecto integrador.
- Evaluaciones: proyecto integrador con entregas parciales (1 obligatoria, clase 7) + defensa final.

El proyecto integrador será en grupos (mínimo 3, máximo 4 integrantes) y estará alineado con alguna temática que ustedes elijan (ejemplos: sistema de reservas, gestor de tareas, ecommerce, API de datos, etc).

14 Clases de 4hs

Cronograma de Clases

Clase 1

Introducción a la materia

Introducción a patrones de diseño (¿Qué son? ¿Para qué sirven?).

Concepto de buenas prácticas (Clean Code, deuda técnica).

Presentación y elección temática del proyecto integrador.

Clase 2

Herramientas de repositorios (Git y GitHub)

Introducción práctica a Git (branches, commits, push, pull).

Creación de repositorio inicial del proyecto.

Organización básica del repositorio.

Clase 3

Manejo de dependencias

Herramientas de gestión de paquetes y dependencias (npm/yarn).

Organización inicial del proyecto integrador: instalación y configuración del entorno inicial.

Clase 4

Patrones Creacionales

Singleton, Factory, Builder.

Aplicación práctica en el proyecto.

14 Clases de 4hs

Cronograma de Clases

Clase 5

Patrones Estructurales

Adapter, Decorator, Facade.

Aplicación práctica en el proyecto.

Clase 6

Patrones de Comportamiento

Observer, Strategy, Command.

Aplicación práctica en el proyecto.

Clase 7

Refactoring y Clean Code (Buenas prácticas)

Principios SOLID.

Ejemplos prácticos de refactoring.

Entrega parcial

Deberán entregar un documento: Nombres de los integrantes, temática elegida, los patrones que se podrían aplicar y por qué. Primeros avances, decisiones y argumentación de las mismas.

Clase 8

Documentación del software

Tipos de documentación técnica (README, Swagger).

Desarrollo de documentación técnica del proyecto.

14 Clases de 4hs

Cronograma de Clases

Clase 9

Técnicas de optimización del ciclo de desarrollo

Automatización de tareas (npm scripts).

Flujo de desarrollo eficiente
Aplicación en el proyecto integrador.

Clase 10

Introducción a Testing (Verificación y validación)

Conceptos clave (Pruebas unitarias e integración)

Jest básico

Primera implementación de tests unitarios en el proyecto integrador

Clase 11

Testing avanzado

Cobertura de tests, Mocking.

Implementación práctica sobre proyecto integrador.

Clase 12

Herramientas adicionales de calidad

Linting y formato automático del código (ESLint, Prettier)

Análisis estático básico (introducción SonarQube)

Integración de herramientas en el proyecto integrador.

14 Clases de 4hs

Cronograma de Clases

Clase 13

Integración Continua (CI/CD)

Conceptos básicos CI/CD.

Implementación práctica con GitHub Actions sobre el proyecto integrador.

Automatización de pruebas y verificación.

Clase 14

Cierre y Presentación proyectos integradores

Exposición y evaluación final de proyectos integradores.

Cierre y feedback general.

Introducción
Objetivos de la clase

- Comprender qué son los Patrones de Diseño
- Entender la importancia y beneficios de las buenas prácticas en el desarrollo de software.
- Familiarizarse con el concepto de deuda técnica.
- Presentar el proyecto integrador y definir los grupos de trabajo.

Patrón de Diseño

Definición: Solución general y reutilizable a un problema recurrente en el diseño de software.

No es código listo para copiar → es *una plantilla adaptable*.

Jeremías Fassi

2025

Definición y enfoque

- Un patrón de diseño es una **solución** probada a un **problema recurrente** en el diseño de software.
- No es una porción de código para copiar y pegar, sino una guía o esquema que se adapta al contexto y necesidades del proyecto.
- Su **objetivo** es estandarizar soluciones a problemas que muchos desarrolladores enfrentan una y otra vez.

Origen

- El concepto nació fuera del software, en arquitectura, gracias a Christopher Alexander, quien documentó soluciones constructivas repetitivas.
- En 1994, Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides (*Gang of Four*) adaptaron estos conceptos al desarrollo de software en el libro *Design Patterns: Elements of Reusable Object-Oriented Software*.

¿Por qué son importantes?

- **Lenguaje común**: si un desarrollador dice “esto es un Singleton”, todos entienden la estructura y el objetivo sin explicaciones largas.
- **Mantenimiento y escalabilidad**: facilitan cambiar y ampliar funcionalidades sin romper el resto del sistema.
- **Eficiencia**: en lugar de inventar una solución desde cero, reutilizamos una que ya está probada y optimizada.

Clasificación de patrones

Patrones creacionales

Se enfocan en cómo se crean los objetos, buscando flexibilidad y control. Ejemplos:

- Singleton: asegura una única instancia global.
- Factory: crea objetos sin exponer la lógica de creación.
- Builder: construye objetos complejos paso a paso.

Clasificación de patrones

Patrones estructurales

Definen cómo organizar clases y objetos para formar estructuras más grandes y flexibles.
Ejemplos:

- Adapter: adapta una interfaz a otra diferente.
- Decorator: añade responsabilidades a un objeto sin modificar su código.
- Facade: provee una interfaz simplificada a un sistema complejo.

Clasificación de patrones

Patrones de comportamiento

Se centran en la interacción y comunicación entre objetos. Ejemplos:

- Observer: notifica automáticamente a múltiples objetos cuando hay cambios.
- Strategy: permite intercambiar algoritmos en tiempo de ejecución.
- Command: encapsula una petición como un objeto para parametrizar acciones.

Buenas prácticas en desarrollo de software

¿Qué son? Conjunto de métodos y principios que mejoran la calidad, mantenibilidad y comprensión del código.

Jeremías Fassi

2025

¿Qué son las buenas prácticas?

- Son un conjunto de recomendaciones y principios que buscan que el código sea más legible, mantenible y confiable.
- No son reglas estrictas, sino **guías** probadas que ayudan a que el software sea de calidad y más fácil de evolucionar con el tiempo.
- Beneficios clave:
  - Mejor legibilidad del código.
  - Facilidad de mantenimiento.
  - Menos bugs.
  - Mejores tiempos de desarrollo.
  - Trabajo en equipo más eficiente.

Clean Code

- Robert C. Martin, es un ingeniero de software, reconocido por desarrollar varios principios de diseño de software. Es autor de varios artículos y libros. Fue el editor de la revista C++ Report y primer director de la Agile Alliance.

- **Concepto central**: *"El código se lee mucho más de lo que se escribe"*.

Clean Code

- Algunas prácticas fundamentales:

    1. **Nombres claros y descriptivos**: variables, funciones y clases deben comunicar su propósito sin necesidad de comentarios extra.
    2. **Funciones pequeñas y con un único propósito**: cada función debe hacer solo una cosa, pero hacerla bien.
    3. **Evitar duplicación de código**: el código duplicado multiplica los errores y dificulta el mantenimiento; se recomienda abstraer o reutilizar.

- **Resultado**: código más fácil de leer, probar y modificar.

Deuda técnica

Definición: La deuda técnica es el costo que implica implementar soluciones rápidas o mal diseñadas, generando un esfuerzo adicional futuro en correcciones o mejoras.

Analogía con la *deuda financiera*:

- Se toma un camino más corto (ej. atajo: implementar rápido sin buenas prácticas). A corto plazo parece útil, pero a largo plazo se "paga con intereses" porque el software se vuelve más costoso de mantener.

Ejemplo práctico:

- Imaginen reparar una tubería rota con cinta adhesiva: resuelve el problema momentáneamente, pero genera más trabajo y costos futuros.

Deuda técnica

Ejemplos comunes de deuda técnica:

- Código mal documentado.
- Ausencia de pruebas (tests).
- Soluciones rápidas y poco escalables.

Consecuencias:

- Dificultad en agregar nuevas funcionalidades.
- Tiempo adicional dedicado al mantenimiento.
- Mayor probabilidad de errores.

Enseñanza clave: no siempre se puede evitar la deuda técnica, pero hay que gestionar cuándo y cómo se paga para no comprometer el futuro del proyecto.

Mal código (difícil de leer, mantener y probar)

```python
def f(d, t):
    r = 0
    for x in d:
        if t == "a":
            r += x * 0.21
        elif t == "b":
            r += x * 0.105
        else:
            r += x * 0.15
    return r
```

Problemas

- Nombres poco descriptivos (f, d, t, r).
- Función con múltiples responsabilidades.
- Falta de explicación de qué significa cada tipo "a", "b", etc.
- Duplicación de lógica (cálculo con diferentes valores).

Código limpio y refactorizado

```python
TAX_RATES = {
    "standard": 0.21,
    "reduced": 0.105,
    "intermediate": 0.15
}

def calculate_total_tax(prices, tax_type="standard"):
    """Calcula el impuesto total para una lista de precios según el tipo de tasa."""
    if tax_type not in TAX_RATES:
        raise ValueError(f"Tipo de tasa inválido: {tax_type}")

    rate = TAX_RATES[tax_type]

    total_tax = 0
    for price in prices:
        total_tax += price * rate

    return total_tax
```

Mejoras

- Nombres claros: calculate_total_tax, prices, tax_type.
- Funciones pequeñas y con un único propósito: calcula solo impuestos.
- Sin duplicación: las tasas están en un diccionario centralizado (TAX_RATES).
- Escalabilidad: agregar un nuevo tipo de impuesto es tan simple como añadir un nuevo valor al diccionario.
- Legibilidad: con solo leer la función, se entiende su propósito.

Proyecto integrador

Presentación del trabajo que se desarrollará durante el curso.

Jeremías Fassi

2025

Presentación del trabajo

Durante toda la materia se desarrollará un proyecto integrador, que servirá para aplicar lo aprendido en teoría a un caso práctico.

El proyecto no tiene que ser demasiado grande, pero sí debe demostrar correcta aplicación de patrones de diseño, buenas prácticas y herramientas modernas.

Requisitos generales

Mínimo 2 patrones de diseño aplicados (a elección).

Control de versiones con Git/GitHub.

Buenas prácticas: código limpio, organizado y documentado.

Testing básico implementado: al menos pruebas unitarias para funciones clave.

Documentación mínima:

- README con descripción del proyecto.
- Breve guía de instalación y uso.

Elección de temática

El grupo elige libremente, pero se recomienda un proyecto sencillo y alcanzable en el tiempo disponible.

Ejemplos:

- Sistema de reservas: turnos para médicos, canchas deportivas, etc.
- Gestor de tareas: tipo Trello básico.
- Ecommerce simple: catálogo de productos, carrito, compras.
- API de noticias: consulta y filtrado de artículos.

Formación de grupos

Mínimo 3 integrantes, máximo 4.

Se espera trabajo colaborativo: reparto de roles y responsabilidades.

La organización y división de tareas será parte de la evaluación (se verán los commits en el historial de Git).

Evaluación del proyecto

No se evalúa solo el resultado final, sino también:

- La evolución durante el curso.
- La calidad del código y uso de patrones.
- La gestión del repositorio (commits, branches, issues).
- La presentación final (defensa del proyecto).

Muchas Gracias

Jeremías Fassi

Javier Kinter