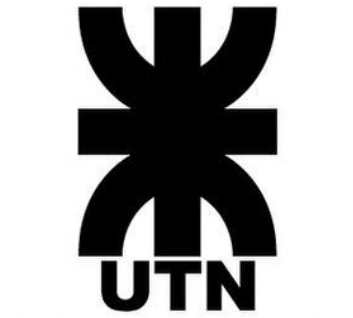
Universidad Tecnológica Nacional

Facultad Regional Córdoba

Ingeniería en Sistemas de Información

Ingeniería y Calidad de Software



Práctico 6: TDD - Test Driven Development

Año: 2025

Curso: 4K2

Grupo Nro: 10

Integrantes:

- 82603 - Maldonado, Francisco Daniel
- 86199 - Ladux, Carlos Agustin
- 86495 - Lypnik, Zoi
- 89398 - Sisca, Tomas Nazareno
- 90023 - Carreras, Nicolás
- 90230 - Ocampo Vysin, Arian Javier
- 90317 - Blencio, Juan Mateo
- 91140 - Salum, Mauricio

TABLA DE CONTENIDOS

Introducción

Funcionalidad analizada: Inscripción a actividades del bioparque EcoHarmony Park

User Story:

Inscribirme a actividad **COMO visitante QUIERO inscribirme a una actividad PARA reservar mi lugar en la misma.**	**3**
Criterios de Aceptación: • Debe requerir seleccionar una actividad del conjunto de actividades de la lista de "Tirolesa", "Safari", "Palestra" y "Jardinería", siempre y cuando tengan cupos disponibles para el horario seleccionado • Debe requerir seleccionar el horario dentro de los disponibles • Debe indicar la cantidad de personas que participaran de la actividad • Para cada persona que participa, debe ingresar los datos del visitante: nombre, DNI, edad y talla de vestimenta si la actividad lo demanda • Debe requerir aceptar los términos y condiciones específicos de la actividad en la que participarán.	
Pruebas de usuario: - Probar inscribirse a una actividad del listado que poseen cupos disponibles, seleccionando un horario, ingresando los datos del visitante (nombre, DNI, edad, talla de la vestimenta si la actividad lo requiere) y aceptando los términos y condiciones (pasa) - Probar inscribirse a una actividad que no tiene cupo para el horario seleccionado (falla) - Probar inscribirse a una actividad sin ingresar talle de vestimenta porque la actividad no lo requiere (pasa) - Probar inscribirse a una actividad seleccionando un horario en el cual el parque está cerrado o la actividad no está disponible (falla) - Probar inscribirse a una actividad sin aceptar los términos y condiciones de la actividad (falla) - Probar inscribirse a una actividad sin ingresar el talle de la vestimenta requerido por la actividad (falla)	

Objetivo principal: Permitir a los visitantes reservar lugares en actividades mediante una aplicación móvil

Contexto del desarrollo: TDD con Python y Pytest, implementación con Flask para el frontend.

Reglas de Estilo de Código y Estructura, indica el formato y las convenciones de calidad del código, se encuentra en la última sección del documento.

Arquitectura y Patrones de Diseño

Patrones Arquitectónicos

Patrón MVC (Model-View-Controller)
Separación clara entre la lógica de negocio (**Model**), la interfaz de usuario (**View**) y el control de flujo (**Controller**)

- **Models**: Actividad, Visitante, Inscripción
- **Controller**: app.py (maneja flujo de datos y comunicación entre vistas y modelos)
- **View**: plantillas HTML renderizadas desde Flask.

Las ventajas de la aplicación de este patrón serían:
- Facilita mantenimiento y escalabilidad.
- Permite probar la lógica de negocio sin depender del frontend.
- Mejora la legibilidad y separación de responsabilidades.

Se utiliza un diseño modular que extiende el patrón MVC con una Capa de Boundary, siguiendo principios de Arquitectura Limpia para desacoplar la Lógica de Negocio de la interfaz y los formatos de datos externos.

Diseño por Capas

Capa	Archivo	Responsabilidad
Presentación	`app.py`	Interactúa con el usuario (formularios Flask), define las rutas API (/api/inscribir) y rutas de la interfaz.
Boundary (Adaptación)	`boundary.py`	Actúa como un adaptador. Recibe datos serializables (JSON/Dicts) de la capa de Presentación y los transforma a estructuras de datos primitivas (Tuplas) que la Lógica de Aplicación requiere. Expone métodos de consulta de datos (obtener_horarios, obtener_dias_unicos).
Lógica de Aplicación	`gestor_actividades.py`	Contiene el flujo de negocio central (`inscribir`), coordina entidades y es responsable de **crear objetos de dominio** (`Visitante`) a partir de los datos crudos recibidos.

Dominio	`actividad.py`, `visitante.py`, `inscripcion.py`	Representan entidades del negocio con su comportamiento propio (validaciones, actualización de cupos, relaciones).

Cada capa tiene responsabilidades bien delimitadas, lo que permite aislar las pruebas y seguir principios de Clean Architecture.

Las ventajas de la aplicación de esta arquitectura serían:

- Separación clara de responsabilidades, facilitando el mantenimiento y la escalabilidad.
- El dominio y la lógica de aplicación son independientes del framework web (Flask).
- La lógica de negocio puede ser probada unitariamente sin dependencias de la UI (TDD).

Decisiones de la Capa de Presentación

Diseño de Formularios HTML

Simplificación: Un visitante principal + campo de cantidad en lugar de un formulario dinámico para múltiples visitantes.

Justificación:

- Más simple para la implementación que queremos realizar.
- Cumple con los requisitos mínimos
- Puede extenderse posteriormente

Modelo de Dominio

Entidad Actividad

```
class Actividad:
    def __init__(self, nombre, horarios, cupos_por_horario,
requiere_talle=False):
```

Atributos clave:

- nombre: Identificador único de la actividad
- horarios: Lista de horarios disponibles
- cupos_por_horario: Diccionario que mapea horarios a cupos disponibles
- requiere_talle: Flag para validaciones específicas

Comportamiento:
tiene_cupo(): Verifica disponibilidad
descontar_cupo(): Actualiza inventario

Entidad Visitante

```
class Visitante:
    def __init__(self, nombre, dni, edad, talle=None):
```

Decisiones de diseño:

- talle es opcional (None por defecto)
- No hay validaciones complejas en el constructor (simplicidad)
- La complejidad de creación del objeto se maneja dentro de la función crear_visitante() del GestorActividades

Agregado Inscripción

```
class Inscripcion:
    def __init__(self, visitante, actividad, horario,
cantidad_personas, acepta_terminos):
```

Responsabilidades:

- Coordinar la relación entre Visitante y Actividad.
- **No** implementa validaciones ni confirmación

Validaciones

Validaciones implementadas:

- Existencia de Actividad: Se verifica que la actividad solicitada exista.
- Horario y Día: El día y el horario seleccionados deben ser válidos y pertenecer a la estructura de cupos de la actividad.
- Cupo Suficiente: Debe haber cupos disponibles iguales o superiores a la cantidad de visitantes solicitada.
- Términos y Condiciones: El visitante debe aceptar explícitamente los términos y condiciones.
- Talle (Condicional): Si la actividad lo requiere, se verifica que cada visitante haya provisto su talla.

Flujo de validación: Es secuencial. La capa GestorActividades realiza las validaciones principales antes de iterar por los visitantes y lanza una excepción al encontrar el primer error.

Mensajes de error:

- Específicos y orientados al usuario
- En español, como solicita el negocio

Diseño de Servicios y Componentes

Servicio GestorActividades

```
def inscribirse_a_actividad(visitante, nombre_actividad, horario,
cantidad_personas, acepta_terminos, actividades):
```

Decisiones:
- Función pura que recibe todos sus dependencias
- Retorna mensajes en lugar de excepciones
- Busca la actividad por nombre en la lista proporcionada

Justificación:
- Evita manejo complejo de errores.
- Facilita el testeo unitario.

Manejo de Datos y Persistencia

Inicialmente, el diseño se basó en el mantenimiento de datos **en memoria** (MVF). Sin embargo, se ha introducido una capa de persistencia mediante la clase `Persistencia` para asegurar la integridad de los datos, la gestión de cupos y la trazabilidad de las inscripciones, utilizando **SQLite** como motor de base de datos.

Diseño del Esquema de Base de Datos

Tabla	**Propósito**
activities	Almacena las actividades disponibles (nombre, requiere_talle).
horarios	Almacena los cupos disponibles por actividad, día y hora.
visitantes	Almacena los datos únicos de cada persona (nombre, dni, edad, talle).
inscripciones	Registro maestro de la transacción de inscripción (día, horario, actividad, aceptación de términos).

inscripcion_visitantes	**Tabla Intermedia.** Asocia múltiples visitantes a una única inscripción.

Vista Lógica: Se creó la vista `visitante_horarios` para facilitar la validación de la **regla de negocio de no-conflicto de horarios** (un visitante no puede estar inscrito dos veces en el mismo día y hora).

Estrategia de Testing

Tecnología usada: Pytest.
Justificación: permite pruebas unitarias simples y repetibles, compatibles con TDD.

Estructura de pruebas:

- Cada test cubre un caso del criterio de aceptación.
- Se va a usar un fixture actividades_disponibles para simular datos reales.
- Los nombres de los tests describen el escenario que prueban.

Cobertura:

- Inscripción exitosa.
- Falta de cupos.
- Inscripción sin talle cuando no se requiere.
- Horario inválido.
- Términos no aceptados.
- Talle faltante cuando es requerido.

TDD aplicado:
Red: se redactan los tests antes de la implementación. Se escriben tests que llaman a métodos (ej. GestorActividades.inscribir) que inicialmente fallan porque las validaciones de negocio aún no existen.
Green: se desarrolla el código mínimo para aprobarlos. Se añade el código mínimo para satisfacer el test, como la implementación de `raise ValueError` para errores (ej. `if not actividad: raise ValueError(...)`).
Refactor: se refactoriza el código manteniendo la funcionalidad. Se introduce la capa **Boundary** y se reestructura la responsabilidad de `crear_visitante()` para mejorar el diseño sin romper los tests existentes que interactúan con `GestorActividades`.

Manejo de Errores

Decisión: Se utiliza el mecanismo de Excepciones (raise ValueError) para comunicar errores de negocio.

Implementación: Las capas de Dominio (Actividad) y Lógica de Aplicación (GestorActividades) lanzan ValueError al fallar validaciones críticas (ej. falta de cupo, horario inválido, talle requerido faltante).

La capa de Presentación (app.py) o Frontera (rutas API en app.py) es responsable de capturar estas excepciones y transformarlas en mensajes de error visibles para el usuario.

Justificación:
- Más simple para el frontend Flask
- No requiere manejo de excepciones complejo
- Adecuado para errores de negocio esperados

Cumplimiento de Requisitos

Criterios de aceptación (User Story):
- CA1: Seleccionar actividad con cupos disponibles.
- CA2: Seleccionar horario válido.
- CA3: Indicar cantidad de personas.
- CA4: Ingresar datos del visitante.
- CA5: Aceptar términos y condiciones.

Pruebas de usuario cubiertas:
- PU1: Inscripción exitosa.
- PU2: Falla por falta de cupos.
- PU3: Éxito sin talle cuando no se requiere.
- PU4: Falla con horario no disponible.
- PU5: Falla sin aceptar términos.
- PU6: Falla sin talle cuando la actividad lo requiere.

Nota: Se incluye la validación técnica adicional de “actividad inexistente”, aunque no forma parte de los criterios de aceptación funcionales.

Escenario (Prueba de Usuario)	**Código de Validación Asociado**	**Criterio (CA) Cubierto**
PU1: Inscripción exitosa.	Pasa todas las validaciones y llama a `Actividad.descontar_cupo()`.	CA1, CA2, CA3, CA4, CA5

PU2: Falla por falta de cupos.	Capturado por `Actividad.tiene_cupo()` dentro de `GestorActividades.inscribir`.	Falla CA1
PU4: Falla por horario no disponible.	Validado en `GestorActividades.inscribir` con la búsqueda `if dia not in actividad.cupos_por_horario...`	Falla CA2
PU5: Falla sin aceptar términos.	Validado al comienzo de `GestorActividades.inscribir`: `if not acepta_terminos_condiciones:`	Falla CA5
PU6: Falla sin talle cuando es requerido.	Validado dentro del bucle de visitantes en `GestorActividades.inscribir`: `if actividad.requiere_talle and (len(v) < 4 or v[3] is None):`	Falla CA4

Reglas de Estilo de Código y Estructura

Esta sección define las convenciones de estilo de código adoptadas por el Grupo Nro. 10 para el desarrollo del Trabajo Práctico, basadas principalmente en el estándar PEP 8 de Python, ajustadas para mantener la claridad en el contexto de la Arquitectura por Capas implementada.

Estructura y Organización de Archivos

Regla	Convención Adoptada	Justificación
Arquitectura	Estructura por Capas (Dominio, Lógica de Aplicación, Boundary, Presentación).	Facilita el **TDD** y el cumplimiento de principios de **Clean Architecture** (desacoplamiento).
Nombres de Archivos	Usar **snake_case** (minúsculas_con_guiones_bajos).	Estándar PEP 8 para módulos y paquetes. (Ej: gestor_actividades.py, test_inscripcion_actividad.py).
Importaciones	Agrupar importaciones en bloques separados y ordenados: Librerías estándar > Librerías de terceros > Módulos locales.	Mejora la legibilidad y la identificación de dependencias. (Ej: import pytest, luego from gestor_actividades import GestorActividades).

Nombres de Identificadores (Clases, Funciones y Variables)

Identificador	Convención Adoptada	Ejemplos en el Código	Justificación
Clases	**PascalCase** (Nombres Comenzando con Mayúscula).	GestorActividades, Actividad, Visitante, Inscripcion, Boundary.	Estándar PEP 8 para Clases. Identifica claramente los **objetos de Dominio y Servicio**.
Funciones/ Métodos	**snake_case** (minúsculas_con_guiones_bajos).	inscribir, agregar_actividad, obtener_cupos_disponibles, test_inscripcion_exitosa_con_talle_con_cupo_ok.	Estándar PEP 8. Mantiene la uniformidad y la legibilidad.
Argumentos /Variables	**snake_case** (minúsculas_con_guiones_bajos).	nombre_actividad, cupos_por_horario, acepta_terminos_condiciones, excinfo.	Estándar PEP 8.

Convenciones Específicas para Testing y TDD

Regla	Convención Adoptada	Ejemplos en el Código	Justificación
Nombres de Tests	Usar **test_** como prefijo. Nombre debe ser descriptivo (Acción + Resultado).	test_falla_por_sin_cupo, test_inscripcion_exitosa_sin_talle_con_cupo_ok.	Cumple con el estándar Pytest para descubrimiento de pruebas y documenta el **escenario funcional** que se está probando.
Manejo de Fallas	Usar pytest.raises(ValueError) para validar escenarios de fallo de negocio.	with pytest.raises(ValueError) as excinfo:	El código de servicio lanza ValueError. Esta convención asegura que el test verifique la excepción correcta y el mensaje de error específico.
Fixtures	Usar @pytest.fixture para inicializar el estado del sistema (GestorActividades con datos iniciales).	def gestor(): ...	Evita la repetición de código de inicialización y asegura que cada prueba tenga un estado limpio y predecible.
Estructura de Test	Uso de secciones (# Precondiciones, # Llamadas a funciones, # Resultados) dentro de cada test.		Mejora la legibilidad de las pruebas y facilita la revisión de pares.