README PROYECTO DE SISTEMA DE GESTIÓN DE TAREAS

Integrantes:
Juan Pablo Gutierrez Lopez
Moises David Zambrano Saurith

Docente:
Helbert Valencia

Diseño de Software

Campus Ingeniería de Software

Santa Marta - Colombia
Universidad Cooperativa de Colombia

2025

DOCUMENTACIÓN TÉCNICA COMPLETA

Sistema de Gestión de Tareas con Prioridades Inteligentes

Tabla de Contenidos

1. Descripción General

Objetivo del Proyecto

Desarrollar un sistema completo de gestión de tareas que permita a usuarios crear, editar, eliminar y organizar tareas personales o académicas mediante un sistema de priorización manual con capacidad de ordenamiento dinámico.

Características Principales

- **CRUD Completo**: Crear, Leer, Actualizar, Eliminar tareas
- **Priorización Manual**: Control total sobre el nivel de prioridad (1-5)
- **Ordenamiento Dinámico**: Ordenar por nivel de prioridad
- **Persistencia**: Almacenamiento en formato JSON
- **Interfaz Gráfica**: Desarrollada con Tkinter
- **Arquitectura Modular**: Separación clara de responsabilidades
- **Actualización en Tiempo Real**: Vista actualizable sin reiniciar

Tecnologías Utilizadas

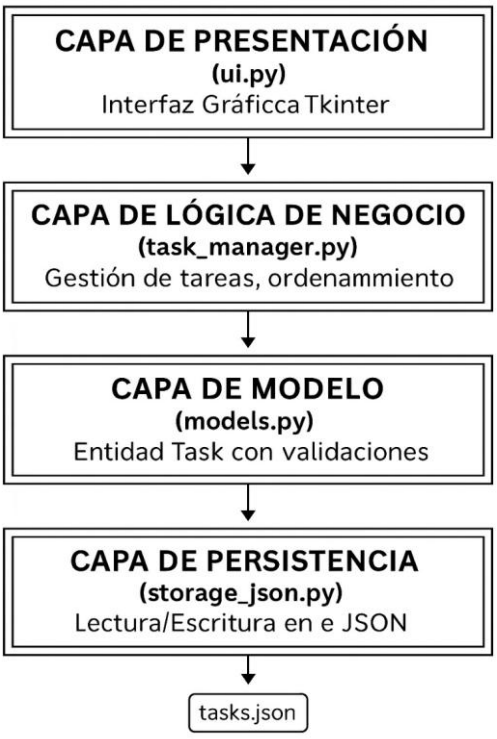
- **Lenguaje**: Python 3.8+
- **Interfaz Gráfica**: Tkinter (ttk widgets)
- **Persistencia**: JSON
- **Arquitectura**: MVC (Model-View-Controller)
- **Identificadores Únicos**: UUID4

2. Arquitectura del Sistema

Estructura del proyecto

```
gestion/
├── data/
│   └── tasks.json      # Carpeta de datos persistentes
├── source/
│   ├── __pycache__     # Caché de Python (auto-generado)
│   ├── __init__.py     # Inicializador del paquete
│   ├── app.py          # Punto de entrada principal
│   ├── models.py       # Modelo de datos (Task)
│   ├── task_manager.py # Controlador/Lógica de negocio
│   ├── storage_json.py # Capa de persistencia
│   ├── priority_strategies.py # Estrategias de priorización
│   └── ui.py           # Vista/Interfaz gráfica
└── test_sistema_tareas.py # Documentación completa del proy
└── README.md
```

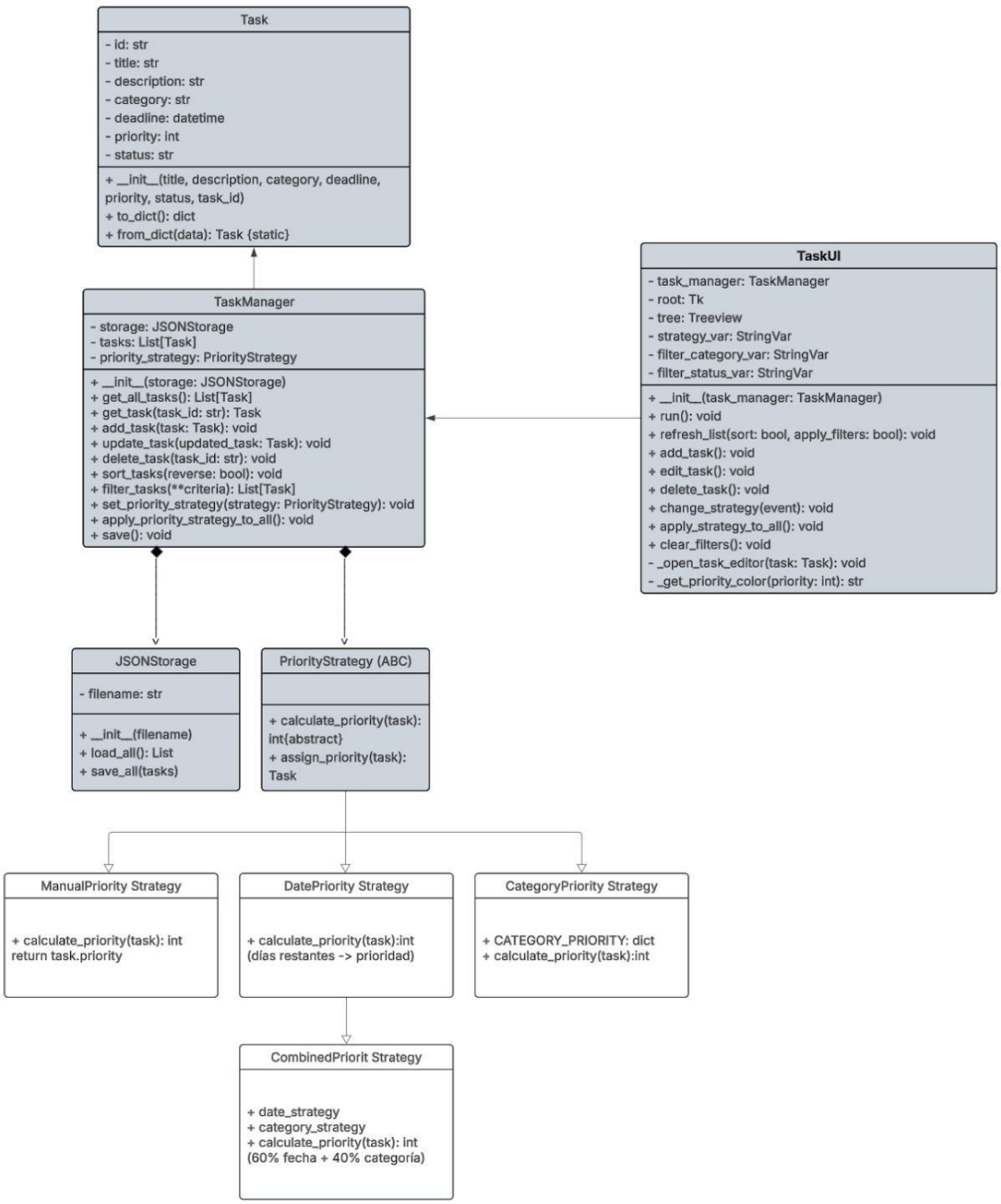
Capas de aplicación



3. DIAGRAMAS UML

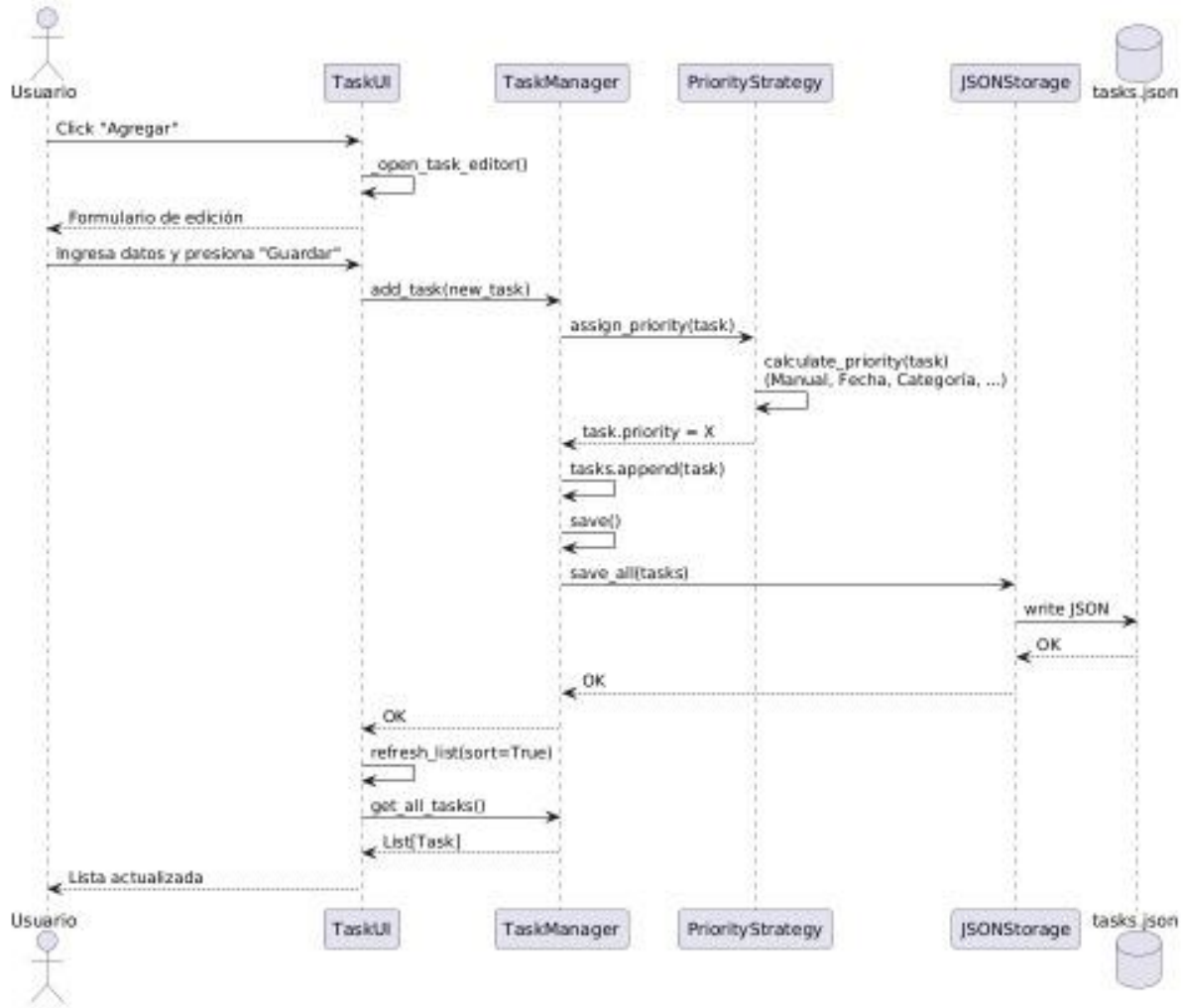
1. DIAGRAMA DE CLASES

Este diagrama muestra la estructura estática del sistema, las relaciones entre clases y sus atributos/métodos.



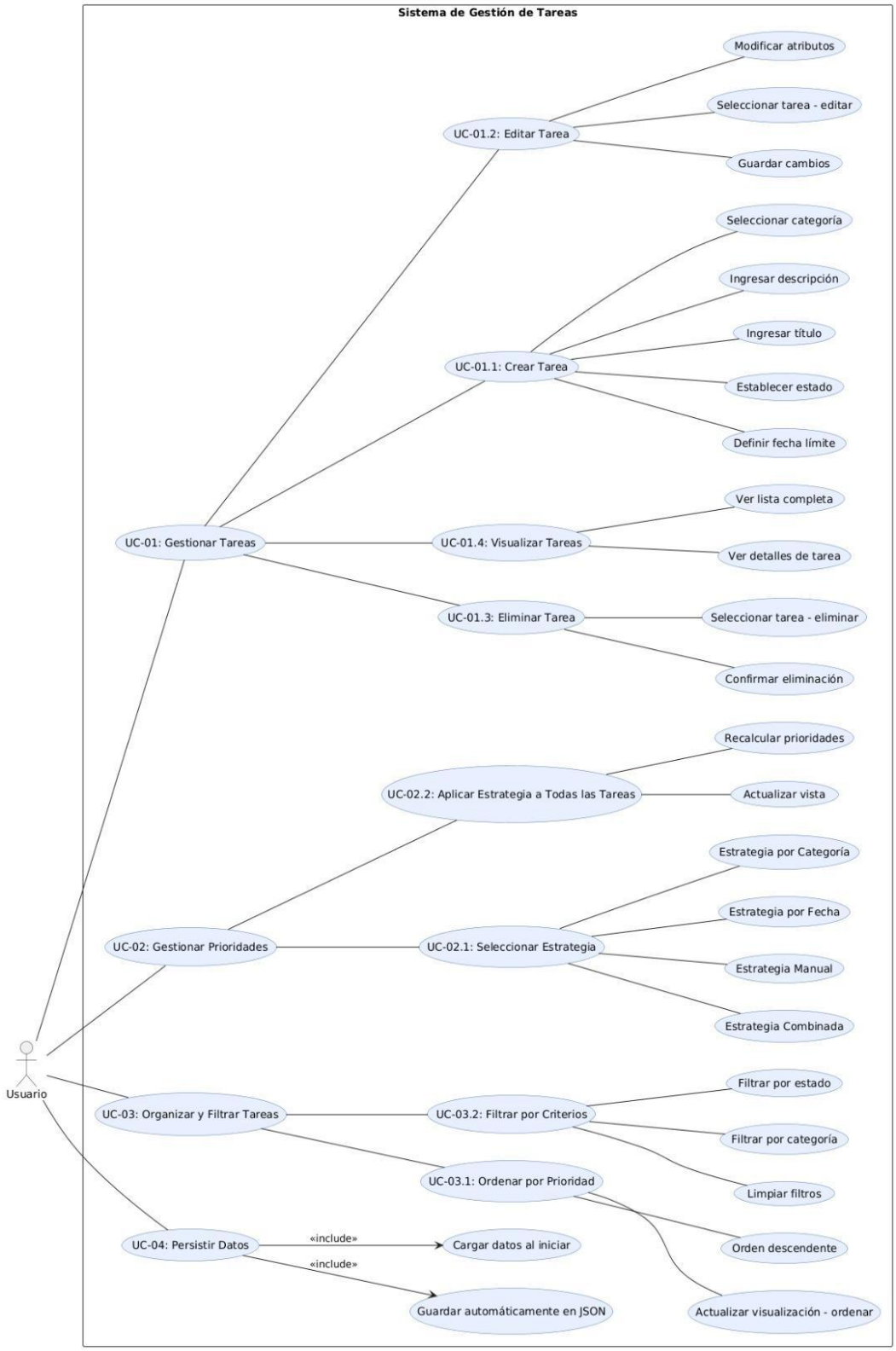
2. DIAGRAMA DE SECUENCIA

Este diagrama muestra la interacción temporal entre objetos para el caso de uso: **"Agregar tarea con estrategia de priorización"**



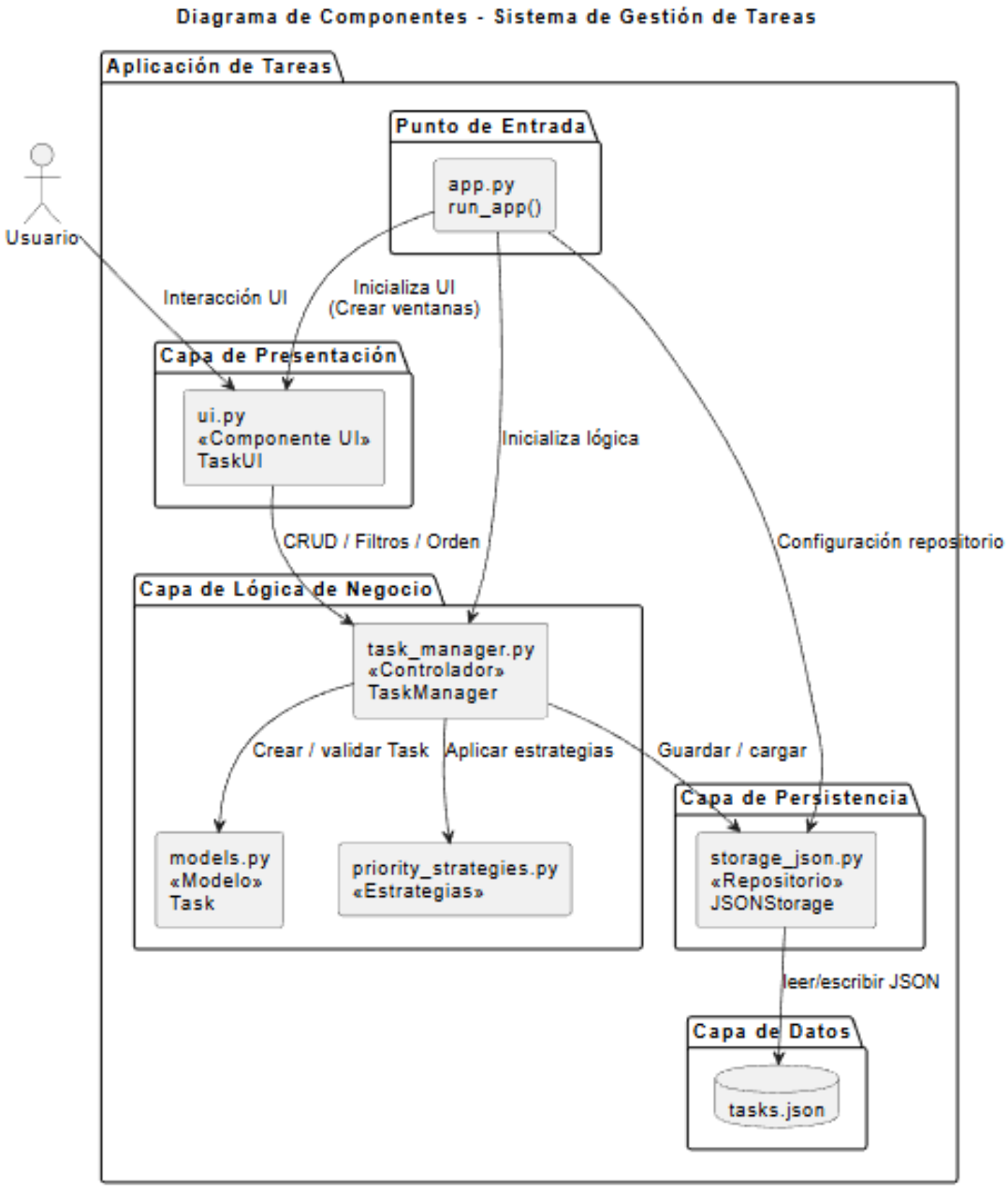
3. DIAGRAMA DE CASOS DE USO

Este diagrama muestra las funcionalidades principales del sistema desde la perspectiva del usuario.



4. DIAGRAMA DE COMPONENTES

Este diagrama muestra la arquitectura física del sistema y cómo se relacionan los diferentes módulos/archivos.



4. PATRONES DE DISEÑO

1. Strategy Pattern (Patrón Estrategia)

Propósito: El sistema está diseñado para soportar múltiples algoritmos de priorización intercambiables.

```python
class PriorityStrategy(ABC):
    @abstractmethod
    def calculate_priority(self, task):
        pass

# Estrategias concretas
class ManualPriorityStrategy(PriorityStrategy):
    def calculate_priority(self, task):
        return task.priority

class DatePriorityStrategy(PriorityStrategy):
    def calculate_priority(self, task):
        days_remaining = (task.deadline - datetime.now()).days
        if days_remaining <= 0:
            return 100  # Máxima urgencia
        return max(1, 30 - days_remaining)

class CategoryPriorityStrategy(PriorityStrategy):
    CATEGORY_PRIORITY = {
        "Urgente": 100,
        "Trabajo": 80,
        "Salud": 90,
        "General": 40
    }
    def calculate_priority(self, task):
        return self.CATEGORY_PRIORITY.get(task.category, 40)

class CombinedPriorityStrategy(PriorityStrategy):
    def calculate_priority(self, task):
        date_priority = DatePriorityStrategy().calculate_priority(task)
        category_priority = CategoryPriorityStrategy().calculate_priority(task)
        return int(date_priority * 0.6 + category_priority * 0.4)
```

Ventajas del diseño:

- Facilita agregar nuevas estrategias sin modificar código existente
- Cada algoritmo está encapsulado
- Arquitectura preparada para evolución

2. Repository Pattern (Patrón Repositorio)

Propósito: Abstrae la lógica de acceso a datos, permitiendo cambiar el mecanismo de persistencia sin afectar la lógica de negocio.

```python
class JSONStorage:
    def __init__(self, filename):
        self.filename = filename
        # Crea archivo vacío si no existe
        if not os.path.exists(self.filename):
            with open(self.filename, "w", encoding="utf-8") as f:
                f.write("[]")

    def save(self, tasks):
        """Guarda lista de objetos Task en JSON"""
        data = [task.to_dict() for task in tasks]
        with open(self.filename, "w", encoding="utf-8") as f:
            json.dump(data, f, indent=4)

    def load(self):
        """Carga tareas desde JSON y devuelve lista de Task"""
        if not os.path.exists(self.filename):
            return []

        with open(self.filename, "r", encoding="utf-8") as f:
            content = f.read().strip()
            if not content:
                return []
            data = json.loads(content)

        return [Task.from_dict(item) for item in data]
```

Ventajas:

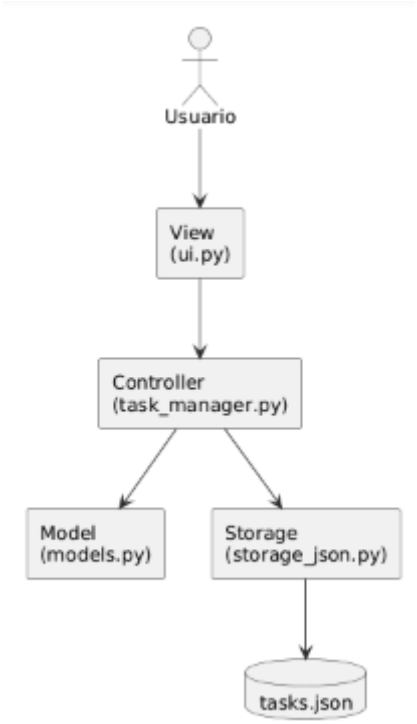
- Separación completa entre negocio y persistencia
- Facilita testing (mock del storage)
- Flexibilidad para cambiar formato de datos

3. MVC (Model-View-Controller)

Implementación real en el proyecto:

Componente	Archivo	Responsabilidad Real
Model	models.py	Entidad Task con conversión a/desde diccionario
view	ui.py	Interfaz Tkinter, eventos de usuario
Controller	task_manager.py	CRUD, ordenamiento, coordinación con storage

Flujo de datos actual:



5. MANUAL DE USUARIO

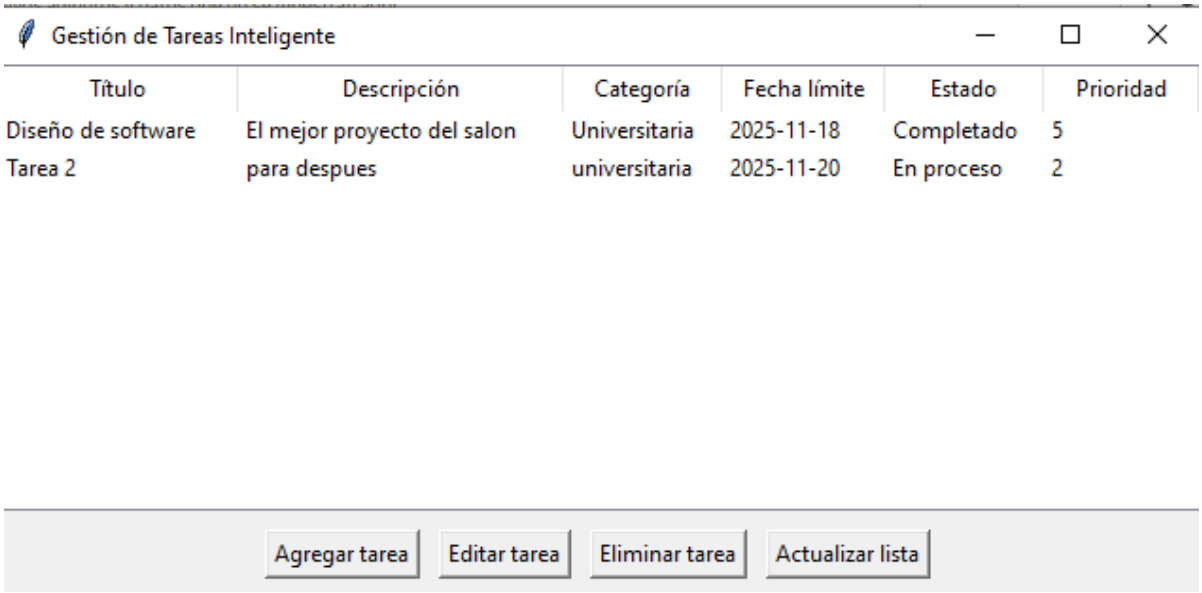
Inicio de la Aplicación

1. **Ejecutar el programa**:

cd source

python app.py

2. **Interfaz principal**:



Gestión de Tareas

Crear una Nueva Tarea

1. Click en botón **"Agregar tarea"**
2. Completar formulario:
    - **Título**: Nombre descriptivo (obligatorio)
    - **Descripción**: Detalles adicionales
    - **Categoría**: Texto libre (ej: Trabajo, Personal, Estudio)
    - **Fecha límite**: Formato YYYY-MM-DD (ej: 2025-11-20)
    - **Estado**: Texto libre (ej: Pendiente, En Proceso, Completada)
    - **Prioridad (1-5)**: Valor numérico del 1 al 5 (por defecto 5)
3. Click en **"Guardar"**
4. La tarea aparecerá en la lista

Ejemplo:

Título: Entregar informe trimestral

Descripción: Informe financiero Q4 para la junta directiva

Categoría: Trabajo

Fecha límite: 2025-11-18

Estado: Pendiente

Prioridad: 5

Editar una Tarea Existente

1. **Seleccionar** la tarea en la tabla (click sobre la fila)
2. Click en botón **"Editar tarea"**
3. Modificar los campos necesarios
4. Click en **"Guardar"**
5. Los cambios se guardan automáticamente

Nota: Si no se selecciona ninguna tarea, aparecerá un mensaje de advertencia.

Eliminar una Tarea

1. **Seleccionar** la tarea en la tabla
2. Click en botón **"Eliminar tarea"**
3. Confirmar en el diálogo de confirmación
4. La tarea se elimina permanentemente del archivo JSON

Actualizar la Lista

El botón **"Actualizar lista"** tiene dos comportamientos:

1. **Refresca la vista** mostrando todas las tareas
2. **Ordena por prioridad** de mayor a menor (reverse=True)

Esto significa que las tareas con prioridad 5 aparecerán primero, y las de prioridad 1 al final.

Funcionamiento de Prioridades

Sistema Actual: Priorización Manual

Cómo funciona:

- Al crear una tarea, el usuario ingresa un valor de prioridad (1-5)
- Este valor se guarda directamente en la tarea
- No hay recálculo automático de prioridades
- El sistema ordena las tareas según este valor cuando se presiona "Actualizar lista"

Escala de prioridades:

- **5**: Máxima prioridad (aparece primero al ordenar)
- **4**: Alta prioridad
- **3**: Prioridad media
- **2**: Baja prioridad
- **1**: Mínima prioridad (aparece último al ordenar)

Inicio del día:

- Abrir la aplicación
- Click en "Actualizar lista" para ver tareas ordenadas por prioridad

Agregar nueva tarea urgente:

- Click en "Agregar tarea"
- Completar datos
- Asignar prioridad 5 (máxima)
- Guardar

Trabajar en tareas:

- Seleccionar tarea de mayor prioridad
- Click en "Editar tarea"
- Cambiar estado a "En Proceso"
- Guardar

Completar tarea:

- Seleccionar tarea terminada
- Editar y cambiar estado a "Completada"
- O eliminarla si ya no es necesaria

Reorganizar:

- Click en "Actualizar lista" para ver el orden actualizado

6. CASOS DE USO DETALLADOS

CU-01: Organizar Tareas del Día

Actor: Usuario final

Precondición: Sistema iniciado, sin tareas

Flujo principal:

1. Usuario abre la aplicación
2. Sistema muestra interfaz vacía
3. Usuario agrega tres tareas:
    1. **Tarea A**: "Reunión con cliente" - Prioridad 5
    2. **Tarea B**: "Revisar emails" - Prioridad 2
    3. **Tarea C**: "Preparar presentación" - Prioridad 4
4. Usuario presiona "Actualizar lista"
5. Sistema ordena y muestra:
    1. Reunión con cliente (5)
    2. Preparar presentación (4)
    3. Revisar emails (2)

Resultado: Tareas organizadas visualmente por importancia

CU-02: Editar Prioridad de Tarea Existente

Actor: Usuario final

Precondición: Existen tareas en el sistema

Flujo:

1. Usuario identifica tarea que debe ser más urgente
2. Selecciona la tarea en la tabla
3. Click en "Editar tarea"
4. Cambia prioridad de 2 a 5
5. Guarda cambios
6. Sistema actualiza la vista automáticamente (sin ordenar)
7. Usuario presiona "Actualizar lista" para ver el nuevo orden
8. La tarea editada ahora aparece en la parte superior

Resultado: Prioridad actualizada y tarea reposicionada

CU-03: Eliminar Tareas Completadas

Actor: Usuario final

Precondición: Existen tareas con estado "Completada"

Flujo:

1. Usuario revisa lista de tareas
2. Identifica tareas completadas
3. Selecciona primera tarea completada
4. Click en "Eliminar tarea"
5. Confirma en diálogo
6. Sistema elimina tarea y actualiza vista
7. Repite proceso para otras tareas completadas

Resultado: Lista limpia solo con tareas pendientes/en proceso

7. PRINCIPIOS SOLID (Aplicados en el Código Actual)

S - Single Responsibility Principle

Cada clase tiene una única responsabilidad bien definida:

Clase	Responsabilidad Única	Evidencia en el código
Task	Representar datos de una tarea	`to_dict()`, `from_dict()`, atributos
TaskManager	Coordinar operaciones CRUD	`add_task()`, `update_task()`, `delete_task()`, `sort_tasks()`
JSONStorage	Persistir/recuperar datos	`load()`, `save()`
TaskUI	Renderizar interfaz y eventos	`refresh_list()`, `_open_task_editor()`

O - Open/Closed Principle

```python
# El diseño permite agregar nuevas estrategias sin modificar TaskManager
class CustomPriorityStrategy(PriorityStrategy):
    def calculate_priority(self, task):
        # Nueva lógica personalizada
        return custom_value
```

L - Liskov Substitution Principle

Implementado en el diseño de estrategias:

```python
# Cualquier estrategia puede reemplazar a otra (cuando se implemente)
# Todas respetan el contrato de PriorityStrategy
strategy1 = ManualPriorityStrategy()    # ✅ Válido
strategy2 = DatePriorityStrategy()      # ✅ Válido
# Ambas son intercambiables sin romper el sistema
```

I - Interface Segregation Principle

Interfaces mínimas y específicas:

```python
# PriorityStrategy: interfaz mínima con un solo método abstracto
class PriorityStrategy(ABC):
    @abstractmethod
    def calculate_priority(self, task): pass
    # Solo obliga a implementar lo esencial

# Task: sin métodos innecesarios
# - to_dict(): necesario para serialización
# - from_dict(): necesario para deserialización
# Nada más
```

D - Dependency Inversion Principle

Implementado correctamente:

```python
# TaskManager depende de abstracción (JSONStorage), no de implementación
class TaskManager:
    def __init__(self, storage: JSONStorage):
        self.storage = storage  # Inyección de dependencia
        self.tasks = self.storage.load()  # Usa interfaz abstracta

# Esto permite cambiar implementación sin modificar TaskManager:
# storage = JSONStorage("tasks.json")    # ✅
# storage = CSVStorage("tasks.csv")      # ✅ (si se implementa)
# storage = SQLiteStorage("tasks.db")    # ✅ (si se implementa)
```

8. PRUEBAS UNITARIAS

El proyecto incluye un conjunto completo de pruebas unitarias y de integración en el archivo

Estructura de Pruebas

```
import unittest
import sys
import os
from datetime import datetime, timedelta

# Configurar path para importar módulos del proyecto
sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__), '..',
    'source')))

from models import Task
from task_manager import TaskManager
from storage_json import JSONStorage
from priority_strategies import ManualPriorityStrategy, DatePriorityStrategy
```

Categorías de Pruebas

1. Test Task - Pruebas del Modelo

Valida la creación, serialización y deserialización de tareas.

```
class TestTask(unittest.TestCase):
    """Pruebas básicas de Task"""

    def test_crear_y_convertir_tarea(self):

        tarea = Task("Test", "Descripción", "Personal", "2025-12-31", priority=5
            )

        self.assertEqual(tarea.title, "Test")
        self.assertIsNotNone(tarea.id)

        # Convertir a dict y back
        tarea_dict = tarea.to_dict()
        tarea_nueva = Task.from_dict(tarea_dict)
        self.assertEqual(tarea_nueva.title, "Test")
        print("✓ Task - Creación y conversión: OK")
```

Verifica:

- Creación de tareas con atributos correctos
- Generación automática de IDs únicos
- Serialización a diccionario (to_dict)
- Deserialización desde diccionario (from_dict)

2. TestStorage - Pruebas de Persistencia

Valida el guardado y recuperación de datos en JSON.

```python
class TestStorage(unittest.TestCase):
    """Pruebas de almacenamiento JSON"""

    def setUp(self):
        self.test_file = "test_temp.json"

    def tearDown(self):
        if os.path.exists(self.test_file):
            os.remove(self.test_file)

    def test_guardar_y_cargar(self):
        """Guardar tareas y cargarlas después"""
        storage = JSONStorage(self.test_file)

        tareas = [
            Task("T1", "D1", "Cat1", "2025-12-31", priority=5),
            Task("T2", "D2", "Cat2", "2025-11-30", priority=3)
        ]

        storage.save_all(tareas)
        tareas_cargadas = storage.load_all()

        self.assertEqual(len(tareas_cargadas), 2)
        self.assertEqual(tareas_cargadas[0].title, "T1")
        print("✓ Storage - Guardar y cargar: OK")
```

Verifica:

- Creación de archivos JSON
- Guardado de múltiples tareas
- Carga correcta de datos persistidos
- Limpieza automática de archivos temporales

3. TestTaskManager - Pruebas del Controlador

Valida todas las operaciones CRUD y funciones de gestión.

```python
class TestTaskManager(unittest.TestCase):
    """Pruebas del gestor de tareas"""

    def test_crud_completo(self):
        """Probar crear, leer, actualizar y eliminar"""
        # Crear
        tarea = Task("Nueva", "Desc", "Personal", "2025-12-31")
        self.manager.add_task(tarea)
        self.assertEqual(len(self.manager.get_all_tasks()), 1)

        # Leer
        tarea_encontrada = self.manager.get_task(tarea.id)
        self.assertIsNotNone(tarea_encontrada)
        self.assertEqual(tarea_encontrada.title, "Nueva")

        # Actualizar
        tarea.title = "Modificada"
        tarea.priority = 10
        self.manager.update_task(tarea)
        tarea_actualizada = self.manager.get_task(tarea.id)
        self.assertEqual(tarea_actualizada.title, "Modificada")

        # Eliminar
        self.manager.delete_task(tarea.id)
        self.assertEqual(len(self.manager.get_all_tasks()), 0)
        print("✓ Manager - CRUD completo: OK")

    def test_ordenar_por_prioridad(self):
        """Ordenar tareas por prioridad"""
        self.manager.add_task(Task("Baja", "D", "C", "2025-12-31", priority=2))
        self.manager.add_task(Task("Alta", "D", "C", "2025-12-31", priority=10))
        self.manager.add_task(Task("Media", "D", "C", "2025-12-31", priority=5))

        self.manager.sort_tasks(reverse=True)
        tareas = self.manager.get_all_tasks()

        self.assertEqual(tareas[0].priority, 10)
        self.assertEqual(tareas[2].priority, 2)
        print("✓ Manager - Ordenamiento: OK")

    def test_persistencia(self):
        """Verificar que se guarda correctamente"""
        self.manager.add_task(Task("Persistente", "D", "C", "2025-12-31"))

        # Crear nueva instancia y verificar que los datos persisten
        nuevo_manager = TaskManager(JSONStorage(self.test_file))
        self.assertEqual(len(nuevo_manager.get_all_tasks()), 1)
        print("✓ Manager - Persistencia: OK")
```

Verifica:

- **Create**: Agregar nuevas tareas
- **Read**: Obtener tarea por ID y listar todas
- **Update**: Modificar atributos de tareas
- **Delete**: Eliminar tareas por ID
- **Ordenamiento**: Ordenar por prioridad (ascendente/descendente)
- **Persistencia**: Datos sobreviven entre sesiones

4. TestStrategies - Pruebas de Estrategias de Priorización

Valida el correcto funcionamiento del patrón Strategy.

python

```python
class TestStrategies(unittest.TestCase):
    """Pruebas de estrategias de prioridad"""

    def test_manual_strategy(self):
        """Estrategia manual mantiene prioridad"""
        strategy = ManualPriorityStrategy()
        tarea = Task("Test", "D", "C", "2025-12-31", priority=7)
        tarea = strategy.assign_priority(tarea)
        self.assertEqual(tarea.priority, 7)
        print("✓ Strategy - Manual: OK")

    def test_date_strategy(self):
        """Estrategia de fecha calcula prioridad"""
        strategy = DatePriorityStrategy()

        # Tarea vencida = urgente
        tarea_vencida = Task("Urgente", "D", "C", datetime.now() - timedelta(days=1))
        tarea_vencida = strategy.assign_priority(tarea_vencida)
        self.assertEqual(tarea_vencida.priority, 100)

        # Tarea en 5 días = alta prioridad
        tarea_cercana = Task("Cercana", "D", "C", datetime.now() + timedelta(days=5))
        tarea_cercana = strategy.assign_priority(tarea_cercana)
        self.assertEqual(tarea_cercana.priority, 25)

        # Tarea lejana = baja prioridad
        tarea_lejana = Task("Lejana", "D", "C", datetime.now() + timedelta(days=50))
        tarea_lejana = strategy.assign_priority(tarea_lejana)
        self.assertEqual(tarea_lejana.priority, 1)
        print("✓ Strategy - Fecha: OK")
```

Verifica:

- **ManualPriorityStrategy**: Mantiene prioridad asignada
- **DatePriorityStrategy**: Calcula según fecha límite
    - Vencida (≤0 días): Prioridad 100
    - Cercana (5 días): Prioridad 25
    - Lejana (>30 días): Prioridad mínima
- Aplicación correcta del patrón Strategy

5. TestIntegracion - Pruebas de Integración

Valida el flujo completo del sistema end-to-end.

python

```python
class TestIntegracion(unittest.TestCase):
    """Prueba de integración completa"""

    def test_flujo_completo(self):
        """Flujo completo: crear, modificar, ordenar, estrategias"""
        # Crear tareas
        t1 = Task("Tarea 1", "D1", "Universidad", "2025-12-15", priority=3)
        t2 = Task("Tarea 2", "D2", "Personal", "2025-11-20", priority=8)

        self.manager.add_task(t1)
        self.manager.add_task(t2)
        self.assertEqual(len(self.manager.get_all_tasks()), 2)

        # Modificar estado
        t1.status = "Completado"
        self.manager.update_task(t1)
        self.assertEqual(self.manager.get_task(t1.id).status, "Completado")

        # Ordenar
        self.manager.sort_tasks(reverse=True)
        self.assertEqual(self.manager.get_all_tasks()[0].priority, 8)

        # Aplicar estrategia
        date_strategy = DatePriorityStrategy()
        t1 = date_strategy.assign_priority(t1)
        self.manager.update_task(t1)

        print("✓ Integración - Flujo completo: OK")
```

Cobertura:

- Flujo completo de uso del sistema
- Interacción entre múltiples componentes
- Cambios de estado de tareas
- Aplicación de estrategias dinámicamente
- Persistencia en operaciones complejas

PRUEBA ESPERADA

```
========================================
PRUEBAS SISTEMA GESTIÓN DE TAREAS
========================================

 EJECUTANDO PRUEBAS...

✓ Task - Creación y conversión: OK
✓ Storage - Guardar y cargar: OK
✓ Manager - CRUD completo: OK
✓ Manager - Ordenamiento: OK
✓ Manager - Persistencia: OK
✓ Strategy - Manual: OK
✓ Strategy - Fecha: OK
✓ Integración - Flujo completo: OK

----------------------------------------------------------------------
Ran 8 tests in 0.123s

OK

========================================
 RESUMEN
========================================
Total: 8
 Exitosas: 8
 Fallidas: 0

====================================
 TODAS LAS PRUEBAS PASARON
====================================
```

Beneficios de las Pruebas

1. **Confiabilidad**: Detecta errores antes de producción
2. **Documentación viva**: Las pruebas documentan el comportamiento esperado
3. **Refactoring seguro**: Permite modificar código con confianza
4. **Regresión**: Evita que errores antiguos vuelvan a aparecer
5. **Calidad**: Asegura que cada componente funciona correctamente