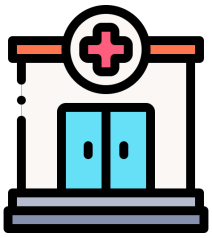
Universidad de Mendoza

Computación I

Trabajo Práctico

Sistema de Gestión para una Clínica



Ingeniería en Informática

Año 2025

Consigna

Desarrollar un sistema de gestión para una **clínica médica** utilizando programación orientada a objetos en Python. El sistema debe permitir:

- Registrar y administrar **pacientes** y **médicos**.
- Gestionar las **especialidades médicas** de cada profesional y sus **días de atención**.
- Agendar **turnos** entre pacientes y médicos respetando disponibilidad, especialidad y horarios.
- Emitir **recetas médicas**.
- Mantener una **historia clínica** para cada paciente, con registro de turnos y recetas.

Requisitos técnicos

El sistema debe implementar clases para:

1. `Paciente`,
2. `Medico`,
3. `Turno`,
4. `Receta`,
5. `Especialidad`,
6. `HistoriaClinica`,
7. y una clase principal `Clinica`.

También se deben implementar las validaciones estrictas desde el **modelo** (no desde la interfaz de consola) y **Excepciones personalizadas** para manejar. Es decir, las validaciones deben ser implementadas en la clase del modelo en el que se está trabajando, como por ejemplo en la clase `Paciente` se deben implementar las validaciones respectivas al paciente no en otro lado.

Debe incluir una **interfaz de consola (CLI)** para interactuar con el sistema y las respectivas **pruebas unitarias** usando `unittest`.

Entregables

Esto es lo que se debe entregar en el repositorio:

1. Código fuente del sistema con separación clara entre modelo y CLI.
2. Pruebas unitarias que cubran los casos principales y errores.
3. Documentación breve de:
   - Cómo ejecutar el sistema.
   - Cómo ejecutar las pruebas.
   - Explicación de diseño general.

Clases y Responsabilidades

En este apartado se presentará una sugerencia de las clases del sistema y sus responsabilidades. Cada clase se presentará especificando el nombre y el tipo de dato (ej: **str** → string) de los atributos privados y los métodos que se implementan.

Se pueden agregar más atributos y métodos según sea necesario.

Clase Paciente

Representa a un paciente de la clínica.

Atributos Privados

- `__nombre__`: **str** → Nombre completo del paciente.
- `__dni__`: **str** → DNI del paciente (identificador único).
- `__fecha_nacimiento__`: **str** → Fecha de nacimiento del paciente en formato **dd/mm/aaaa**.

Métodos

Se deben implementar métodos en las clases para:

- **Acceso a información**: `obtener_dni()` → `str` devuelve el DNI del paciente.
- **Representación**: `__str__()` → `str` devuelve una representación legible del objeto (no su dirección de memoria).

Ejemplo de uso

```
paciente = Paciente("Juan Pérez", "12345678", "12/12/2000")
print(paciente.obtener_dni()) # "12345678"
print(paciente) # "Juan Pérez, 12345678, 12/12/2000"
```

Clase Medico

Representa a un médico del sistema, con sus especialidades y matrícula profesional.

Atributos Privados

Al definir la clase medico se deben incluir los siguientes atributos privados:

- `__nombre__`: **str** → Nombre completo del médico.
- `__matricula__`: **str** → Matrícula profesional del médico (clave única).
- `__especialidades__`: **list[Especialidad]** → Lista de especialidades con sus días de atención.

Métodos

Se deben implementar los siguientes métodos:

- **Registro de datos**: `agregar_especialidad(especialidad: Especialidad)` agrega una especialidad a la lista del médico.
- **Acceso a información**:
  - `obtener_matricula()` → `str` devuelve la matrícula del médico.
  - `obtener_especialidad_para_dia(dia: str)` → `str | None` (dia es "lunes", "martes", etc) devuelve el nombre de la especialidad disponible en el día especificado, o `None` si no atiende ese día.
- **Representación**: `__str__()` → `str` devuelve una representación legible del objeto.

Ejemplo de uso

```
medico = Medico(
  "Dr. Juan Pérez",
  "12345678",
  [
    Especialidad(
      "Pediatría",
      ["lunes", "miércoles", "viernes"]
    )
  ]
)
medico.agregar_especialidad(
  Especialidad(
    "Cardiología",
    ["martes", "jueves"]
  )
)
print(medico.obtener_matricula()) # "12345678"
print(medico.obtener_especialidad_para_dia("lunes")) # "Pediatría"
print(medico)
# "Juan Pérez,
#  12345678,
#  [
#    Pediatría (Días: lunes, miércoles, viernes),
#    Cardiología (Días: martes, jueves)
#  ]"
```

Clase Especialidad

Representa una especialidad médica junto con los días de atención asociados.

Atributos Privados

Al definir la clase especialidad se deben incluir los siguientes atributos privados:

- `__tipo__`: **str** → Nombre de la especialidad (por ejemplo, "Pediatría", Çardiología").
- `__dias__`: **list[str]** → Lista de días en los que se atiende esta especialidad, en minúsculas.

Métodos

Se deben implementar los siguientes métodos:

- **Acceso a información**: `obtener_especialidad()` → `str` devuelve el nombre de la especialidad.
- **Validaciones**: `verificar_dia(dia: str)` → `bool` devuelve `True` si la especialidad está disponible en el día proporcionado (no sensible a mayúsculas/minúsculas), `False` en caso contrario.
- **Representación**: `__str__()` → `str` devuelve una cadena legible con el nombre de la especialidad y los días de atención (por ejemplo: P̈ediatría (Días: lunes, miércoles, viernes))̈

Ejemplo de uso

```
especialidad = Especialidad(
  "Pediatría",
  ["lunes", "miércoles", "viernes"]
)
print(especialidad.obtener_especialidad()) # "Pediatría"
print(especialidad.verificar_dia("lunes")) # True
print(especialidad)
# "Pediatría:
#   Días: lunes, miércoles, viernes"
```

Clase Turno

Representa un turno médico entre un paciente y un médico para una especialidad específica en una fecha y hora determinada.

Atributos Privados

Al definir la clase turno se deben incluir los siguientes atributos privados:

- `__paciente__`: **Paciente** → Paciente que asiste al turno.
- `__medico__`: **Medico** → Médico asignado al turno.

- `__fecha_hora__`: **datetime** → Fecha y hora del turno.
- `__especialidad__`: **str** → Especialidad para la cual se agendó el turno.

Métodos

Se deben implementar los siguientes métodos:

- **Acceso a información**:
  - `obtener_medico()` → `Medico` devuelve el médico asignado al turno.
  - `obtener_fecha_hora()` → `datetime` devuelve la fecha y hora del turno.
- **Representación**: `__str__()` → `str` devuelve una representación legible del turno, incluyendo paciente, médico, especialidad y fecha/hora.

Ejemplo de uso

Se utilizarán los mismos objetos de paciente y médico que se usaron en los ejemplos anteriores.

```
fecha_hora = datetime(2025, 12, 12, 12, 0)
turno = Turno(paciente, medico, fecha_hora, "Pediatría")
print(turno.obtener_medico()) # "Dr. Juan Pérez"
print(turno.obtener\_fecha\_hora()) # "2025-12-12 12:00:00"
print(turno)
# "Turno(
#     Paciente(Juan Pérez, 12345678, 12/12/2000),
#     Medico(
#         Juan Pérez,
#         12345678,
#         [Pediatría (Días: lunes, miércoles, viernes)]
#     ),
#     2025-12-12 12:00:00,
#     Pediatría
# )"
```

Clase Receta

Representa una receta médica emitida por un médico a un paciente, incluyendo los medicamentos recetados y la fecha de emisión.

Atributos Privados

Al definir la clase receta se deben incluir los siguientes atributos privados:

- `__paciente__`: **Paciente** → Paciente al que se le emite la receta.
- `__medico__`: **Medico** → Médico que emite la receta.
- `__medicamentos__`: **list[str]** → Lista de medicamentos recetados.
- `__fecha__`: **datetime** → Fecha de emisión de la receta (automáticamente asignada con `datetime.now()`).

Métodos

Se deben implementar los siguientes métodos:

- **Representación**: `__str__()` → `str` devuelve una representación en cadena de la receta.

Ejemplo de uso

Para no repetir código se supondrá que el médico y el paciente ya han sido creados y son los mismos que se usaron en los ejemplos anteriores.

```
medicamentos = ["Paracetamol", "Ibuprofeno"]
receta = Receta(paciente, medico, medicamentos)
print(receta)
# "Receta(
#     Paciente(Juan Pérez, 12345678, 12/12/2000),
#     Medico(
#         Juan Pérez,
#         12345678,
#         [Pediatría (Días: lunes, miércoles, viernes)]
#     ),
#     [Paracetamol, Ibuprofeno],
#     2025-12-12 12:00:00
# )"
```

Clase HistoriaClinica

Representa la historia clínica de un paciente, que incluye una lista de turnos y recetas emitidas.

Atributos Privados

Al definir la clase historia clínica se deben incluir los siguientes atributos privados:

- `__paciente__`: **Paciente** → Paciente al que pertenece la historia clínica.
- `__turnos__`: **list[Turno]** → Lista de turnos agendados del paciente.
- `__recetas__`: **list[Receta]** → Lista de recetas emitidas para el paciente.

Métodos

Se deben implementar los siguientes métodos:

- **Registro de datos**: `agregar_turno(turno: Turno)` agrega un nuevo turno a la historia clínica.
- **Acceso a información**:
  - `obtener_turnos()` → `list[Turno]` devuelve una copia de la lista de turnos del paciente.

    - `obtener_recetas()` → `list[Receta]` devuelve una copia de la lista de recetas del paciente.
- **Representación**: `__str__()` → `str` devuelve una representación textual de la historia clínica, incluyendo turnos y recetas.

Ejemplo de uso

Para no repetir código se supondrá que el médico, el paciente, la receta y el turno ya han sido creados y son los mismos que se usaron en los ejemplos anteriores.

```
historia_clinica = HistoriaClinica(paciente)
historia_clinica.agregar_turno(turno)
historia_clinica.agregar_receta(receta)
historia_clinica.agregar_receta(Receta(
  paciente,
  medico,
  ["Paracetamol", "Ibuprofeno"]
))
print(historia_clinica)
# "HistoriaClinica(
#     Paciente(Juan Pérez, 12345678, 12/12/2000),
#     [
#       Turno(
#         Paciente(Juan Pérez, 12345678, 12/12/2000),
#         Medico(
#             Juan Pérez,
#             12345678,
#             [Pediatría (Días: lunes, miércoles, viernes)]
#         ),
#         2025-12-12 12:00:00,
#         Pediatría
#       ),
#       Receta(
#         Paciente(Juan Pérez, 12345678, 12/12/2000),
#         Medico(
#             Juan Pérez,
#             12345678,
#             [Pediatría (Días: lunes, miércoles, viernes)]
#         ),
#         [Paracetamol, Ibuprofeno],
#         2025-12-12 12:00:00
#       )
#     ]
# )"
```

Clase Clinica

Clase principal que representa el sistema de gestión de la clínica.

Atributos Privados

Al definir la clase clínica se deben incluir los siguientes atributos privados:

- `__pacientes__`: **dict[str, Paciente]** → Mapea DNI del paciente a su objeto correspondiente.
- `__medicos__`: **dict[str, Medico]** → Mapea matrícula de médico a su objeto correspondiente.
- `__turnos__`: **list[Turno]** → Lista de todos los turnos agendados.
- `__historias_clinicas__`: **dict[str, HistoriaClinica]** → Mapea DNI a su historia clínica.

Métodos

Se deben implementar los siguientes métodos:

Registro de datos:

- `agregar_paciente(paciente: Paciente)` → registra un paciente y crea su historia clínica.
- `agregar_medico(medico: Medico)` → registra un médico.
- `agendar_turno(dni: str, matricula: str, especialidad: str, fecha_hora: datetime)` → agenda un turno si se cumplen todas las condiciones.
- `emitir_receta(dni: str, matricula: str, medicamentos: list[str])` → emite una receta para un paciente.

Acceso a información:

- `obtener_pacientes()` → `list[Paciente]` → devuelve todos los pacientes registrados.
- `obtener_medicos()` → `list[Medico]` → devuelve todos los médicos registrados.
- `obtener_medico_por_matricula(matricula: str)` → `Medico` → devuelve un médico por su matrícula.
- `obtener_turnos()` → `list[Turno]` → devuelve todos los turnos agendados.
- `obtener_historia_clinica_por_dni(dni: str)` → `HistoriaClinica` → devuelve la historia clínica completa de un paciente por su DNI.

Validaciones y utilidades:

- `validar_existencia_paciente(dni: str)` → verifica si un paciente está registrado.
- `validar_existencia_medico(matricula: str)` → verifica si un médico está registrado.
- `validar_turno_no_duplicado(matricula: str, fecha_hora: datetime)` → verifica si un turno es válido.
- `obtener_dia_semana_en_espanol(fecha_hora: datetime)` → `str` → traduce un objeto `datetime` al día de la semana en español.

- `obtener_especialidad_disponible(medico: Medico, dia_semana: str)` → `str` → obtiene la especialidad disponible para un médico en un día.
- `validar_especialidad_en_dia(medico: Medico, especialidad_solicitada: str, dia_semana: str)` → verifica que el médico atienda esa especialidad ese día.

En este apartado no se realizará un ejemplo de uso ya que es una combinación de los ejemplos anteriores.

Definición de Excepciones

El sistema utiliza **excepciones personalizadas** para representar errores específicos del dominio de la clínica. Estas excepciones son lanzadas por la clase `Clinica` cuando ocurre una situación inválida o inesperada, como por ejemplo:

- `PacienteNoEncontradoException`
- `MedicoNoDisponibleException`
- `TurnoOcupadoException`
- `RecetaInvalidaException`

La clase `CLI` **captura estas excepciones** usando bloques `try-except` y muestra mensajes claros y amigables para el usuario final, evitando que el programa se detenga o muestre trazas técnicas.

Interfaz de Consola (CLI)

La clase `CLI` actúa como la interfaz de usuario por consola para interactuar con el sistema de gestión de la clínica representado por la clase `Clinica`.

Propósito

Los propósitos de esta clase yacen en la interacción con el usuario. Representa un frontend para el sistema realizado. Entre los objetivos que debe cumplir para este proyecto se pueden destacar las siguientes funciones principales:

- Mostrar un menú interactivo con las opciones disponibles para el usuario.
- Solicitar datos por consola para cada operación.
- Llamar a los métodos correspondientes de la clase `Clinica` para realizar las acciones solicitadas.
- No realizar validaciones de negocio ni lógica compleja; esas responsabilidades están en la clase `Clinica`.
- Gestionar errores y excepciones que ocurren en `Clinica` para mostrar mensajes claros al usuario.

Flujo principal

Al ejecutar el programa, se muestra un menú con opciones numeradas, por ejemplo:

```
--- Menú Clínica ---
1) Agregar paciente
2) Agregar médico
3) Agendar turno
4) Agregar especialidad
5) Emitir receta
6) Ver historia clínica
7) Ver todos los turnos
8) Ver todos los pacientes
9) Ver todos los médicos
0) Salir
```

El menú se muestra en un bucle continuo hasta que el usuario elige salir (0).

Operaciones principales

En base al menú anterior se definen algunas operaciones principales que se realizan en el menú:

- **Agregar paciente**: Solicita nombre, DNI y fecha de nacimiento, crea un objeto `Paciente` y lo registra en la clínica.
- **Agregar médico**: Solicita nombre y matrícula, y las especialidades con sus días de atención. Registra el médico en la clínica.
- **Agendar turno**: Solicita DNI de paciente, matrícula de médico, especialidad y fecha/hora. Intenta agendar el turno validando que no haya conflictos.
- **Agregar especialidad a médico**: Permite añadir especialidades y días de atención a un médico ya registrado.
- **Emitir receta**: Solicita DNI de paciente, matrícula de médico y medicamentos, luego registra la receta.
- **Ver historia clínica**: Muestra la historia clínica completa de un paciente (turnos y recetas).
- **Ver listados completos**: Muestra todos los turnos, pacientes o médicos registrados.

Manejo de errores

Cuando una operación falla por razones como datos inválidos o entidades inexistentes, `CLI` captura las excepciones lanzadas por `Clinica` y muestra mensajes amigables en consola.

Tests Unitarios

El sistema debe incluir pruebas unitarias utilizando el módulo `unittest`, que validan el correcto funcionamiento de las operaciones del modelo, especialmente los casos esperados y los errores posibles.

Casos de prueba cubiertos

Pacientes y Médicos

- Registro exitoso de pacientes y médicos.
- Prevención de registros duplicados (por DNI o matrícula).
- Verificación de errores por datos faltantes o inválidos.

Especialidades

- Agregar especialidades nuevas a un médico ya registrado.
- Evitar duplicados de especialidad en el mismo médico.
- Detección de especialidades con días de atención inválidos.
- Error si se intenta agregar especialidad a un médico no registrado.

Turnos

- Agendar turnos exitosos (medico disponible, especialidad correcta, fecha/hora no duplicada).
- Evitar turnos duplicados (mismo médico y fecha/hora).
- Error si el paciente o médico no existen.
- Error si el médico no atiende la especialidad solicitada.
- Error si el médico no trabaja ese día de la semana.

Recetas

- Emitir recetas exitosas (medico disponible, especialidad correcta, fecha/hora no duplicada).
- Error si el paciente o médico no existen.
- Error si no hay medicamentos listados.

Historias Clínicas

- Obtener la historia clínica completa de un paciente (turnos y recetas).
- Error si se intenta obtener la historia clínica de un paciente no registrado.