Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación
IIC2233 — Programación Avanzada 2026-1

Interrogación 1

15 de Abril 2026

Selección múltiple [20 % nota final]

Cada alternativa vale lo mismo.

1. D
2. E
3. B
4. E
5. E

Desarrollo

Pregunta 1. Estructuras de Datos [20 % nota final]

Distribución de puntaje (máx. 5 puntos)

Por cada lista remplazada es 1 punto. Si el alumno no justifica correctamente, o si no especifica información relevante como qué se usa de llave para un diccionario, tiene mitad de puntaje.

Parte A.

- **(1 punto)** **(0.5)** La lista "`self.esperando`" puede ser remplazada por la EDD una cola de Python (`deque`).

  (0.5) Justificación: Esto permitiría que el método `siguiente_persona` elimine el primer de la cola de manera eficiente. Se da puntaje parcial **(0.25)** si la justificación está incompleta.

Parte B.

- **(1 punto)** **(0.5)** La lista "`self.puntos_de_interes`" puede ser remplazado por un `set` (Un `dict` también puede servir).

  (0.5) Justificación: Tanto el `set` como el `diccionario` permiten buscar por *key* y eliminar los elemento repetidos de manera eficiente. Se da puntaje parcial **(0.25)** si la justificación está incompleta.

- **(1 punto)** **(0.5)** La lista "`nuevo_punto`" debe ser remplazada por una `tuple`.

  (0.5) Justificación: el `nuevo_punto` se va a guardar en `self.puntos_de_interes`, pero para hacer esto, es necesario que la estructura de datos sea *hasheable*. Una tupla con números es hasheable. Se da puntaje parcial **(0.25)** si la justificación está incompleta.

Parte C.

- **(1 punto)** **(0.5)** La lista "`self.usuarios`" debe ser remplazada por un `dict`. Es posible hacerlo de varias formas, pero es necesario que el diccionario use como `key` el rut y almacene la información en el `value`. Es posible también que se creen varios diccionarios, uno para cada valor.

  (0.5) Justificación: El diccionario permite búsqueda eficiente de información usando `keys` sin tener que iterar por sobre todos los datos. Se da puntaje parcial **(0.25)** si la justificación está incompleta.

- **(1 punto)** **(1.0)** La lista en el método de `agregar_datos` puede ser remplazada por un diccionario, una `namedtuple` o incluso una clase con atributos.

  También es posible que si el alumno crea un diccionario por cada atributo, no se use una nueva estructura de datos. En ese caso también sería correcto.

  Se da puntaje parcial **(0.5)** por una remplazo parcialmente correcto.

Pregunta 2. Modelación [20 % nota final]

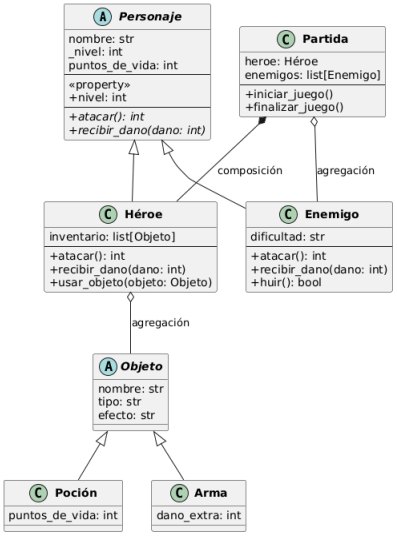
Parte A. Modelación UML

Distribución de puntaje (máx. 20 puntos)

- **(8 puntos)** Genera un modelo consistente que se adecúa a la solución del problema.
- **(3 puntos)** Hay al menos un tipo de herencia que se adecúa a la solución del problema.
- **(2 puntos)** Utiliza agregación correctamente en base al contexto plateado.
- **(2 puntos)** Utiliza composición correctamente en base al contexto plateado.
- **(3 puntos)** Identifica bien todos los tipos de datos en el modelo.
- **(2 puntos)** Valida de forma adecuada el nivel de un personaje, ya se mediante properties o funciones auxiliares.

Medio puntaje si presenta errores menores que no afectan la idea general de la solución

Posible solución



NOTA: las clases con A (azul) son clases abstractas y las clases con C (verde) son clases concretas.

Parte B. Modelación de código

Distribución de puntaje (máx. 20 puntos)

- **(2 puntos)** ___init___ de Personaje bien definido y consistente con el UML.
- **(3 puntos)** ___init___ de Heroe bien definido y consistente con el UML.
- **(3 puntos)** ___init___ de Enemigo bien definido y consistente con el UML.
- **(2 puntos)** ___init___ de Partida bien definido y consistente con el UML.
- **(4 puntos)** Valida de forma correcta el nivel, ya sea con getter y setter de una property o con funciones auxiliares.
- **(3 puntos)** Implementa adecuadamente el método atacar en Heroe.
- **(3 puntos)** Implementa adecuadamente el método atacar en Enemigo.

Medio puntaje si presenta errores menores que no afectan la idea general de la solución

Posible solución

```python
class Personaje(ABC):

    def __init__(self, nombre: str, nivel: int, puntos_de_vida: int):
        self.nombre: str = nombre
        self._nivel: int = nivel if nivel >= 1 else 1
        self.puntos_de_vida: int = puntos_de_vida

    @property
    def nivel(self) -> int:
        return self._nivel

    @nivel.setter
    def nivel(self, valor: int) -> None:
        if valor < 1:
            self._nivel = 1
        else:
            self._nivel = valor


class Heroe(Personaje):

    def __init__(self, nombre: str, nivel: int, puntos_de_vida: int):
        super().__init__(nombre, nivel, puntos_de_vida)
        self.inventario: List[Objeto] = []  # list[Objeto]

    def atacar(self) -> int:
        return self.nivel  # daño base = nivel


class Enemigo(Personaje):

    def __init__(
        self, nombre: str, nivel: int, puntos_de_vida: int, dificultad: str
    ):
        super().__init__(nombre, nivel, puntos_de_vida)
        self.dificultad: str = dificultad

    def atacar(self) -> int:
        if self.dificultad == "fácil":
            return 5
        else:  # "difícil"
            return 15

class Partida:

    def __init__(self, heroe: Heroe):
        self.heroe = heroe  # (composición)
        self.enemigos: List[Enemigo] = []  # (agregación)
```

Pregunta 3. Implementación y ejecución de código [20 % nota final]

Parte A. Implementación de menú [3 puntos]

Distribución puntaje (máx. 3 puntos)

- Flujo Inicial:
  - **(0.2 puntos)** Revisión de número de alumno inicial,
  - **(0.2 puntos)** volviendo a pedirlo en caso de que se ingrese un número de usuario inexistente.
- Escoger opciones:
  - **(0.2 puntos)** Muestra y permite escoger las 4 opciones inicialmente, y
  - **(0.4 puntos)** luego de cada vez de realizar algo distinto de salir
- Mostrar grupo:
  - **(0.2 puntos)** Maneja correctamente la opción de mostrar el grupo del usuario, y
  - **(0.2 puntos)** muestra en consola para el caso sin grupo
  - **(0.2 puntos)** muestra en consola para el caso con grupo
- Crear grupo:
  - **(0.2 puntos)** Maneja correctamente la opción de crear grupo.
- Cambiarse grupo:
  - **(0.2 puntos)** Maneja correctamente la opción de cambiarse de grupo, y
  - **(0.2 puntos)** muestra en consola lo solicitado (pide input al usuario)
- Salir:
  - **(0.2 puntos)** Se reconoce escoger salir como salir
  - **(0.4 puntos)** Se reconoce cualquier valor distinto a las opciones provistas como salir
  - **(0.2 puntos)** Salir finaliza el programa

Posible solución

```
while True:
    num_alumno = input("ingresa tu número de alumnno")
    if Plataforma.existe_alumno(num_alumno):
        while True:
            print("[1] Ver mi grupo")
            print("[2] Crear grupo")
            print("[3] Unirse a un grupo")
            print("[0] Salir")
            opcion = input("ingresa el número de la opción")

            if opcion not in "123" or len(opcion) != 1:
                break
            elif opcion == "1":
                usuarios_grupo = Plataforma.grupo_usuario(num_alumno)
                if usuarios_grupo is None:
                    print("no tienes grupo")
                else:
                    print(usuarios_grupo)
            elif opcion == "2":
                Plataforma.crear_grupo(num_alumno)
            elif opcion == "3":
                print("ingresa número de alumno de algún miembro de ese grupo")
                num_alumno_grupo = input()
                Plataforma.moverse_a_grupo(num_alumno, num_alumno_grupo)
        break
    else:
        print("número de alumnno incorrecto")
```

Parte B. Ejecución de código [3 puntos]

Distribución de puntaje (máx. 3 puntos)

- **(0.25 puntos)** Identifica que son 6 llamadas de la función.
- **(0.75 puntos)** Se identifican valores vacíos corectamente, es decir, cuando `tercero` es `None`, cuando `es_fibonacci` y `viene_fibonacci` no se llegan a asignar, y cuando `otros` es una estructura de datos `tuple` o `list` vacía. Mitad del puntaje si identifica 3 o 4 de los 5 casos.
- **(0.75 puntos)** `otros` se va consumiendo de izquiera a derecha hasta que quede un valor.
- **(0.75 puntos)** `primero`, `segundo` y `tercero` poseen los valores correctos (sin considerar caso de `None`). Mitad del puntaje si hay a lo más un error de arrastre.
- **(0.5 puntos)** `es_fibonacci`, `viene_fibonacci` y `valor_retornado` poseen los valores correctos. Mitad del puntaje si hay a lo más un error de arrastre.

Solución

#	primero	segundo	tercero	otros	es_fibonacci	viene_fibonacci	valor retornado
1	0	1	1	(2, 2, 4, 6)	True	False	False
2	1	1	2	(2, 4, 6)	True	False	False
3	1	2	2	(4, 6)	False	True	False
4	2	2	4	(6,)	True	True	True
5	2	4	6	tuple()	True	True	True
6	4	6	None	tuple()			True

Nota: Para los valores no vacíos de la columna `otros`, se permite:

- Como `tuple` (como está en la solución)
- Como `list`
- Mientras tenga la coma, que no hayan paréntesis.

Pregunta 4. Backtracking [20 % nota final]

Distribución de puntaje (máx. 7 puntos)

- **(0.5 puntos)** Se distingue correctamente si se debe ejecutar el caso base o el caso recursivo.

- **(2 puntos)** Se prueban los valores entre 1 y 9 para una misma celda. Se realiza un llamado recursivo para encontrar una solución al tablero.

- **(1 punto)** Si se encuentra solución, se retorna correctamente el tablero obtenido hasta el llamado inicial de la función recursiva.

- **(1 punto)** Si no se encuentra solución, se realiza *backtracking*.

- **(1 punto)** Se comprueba correctamente que los valores utilizados en el tablero cumplan las 3 reglas fundamentales del Sudoku.

  Esta revisión se puede realizar tanto en el caso base como en el caso recursivo.

- **(1.5 puntos)** Se asegura de que los cambios en el tablero no impida que el algoritmo encuentre un resultado.

  Algunas formas de lograr lo anterior es hacer copias del tablero entre los llamados recursivos (p.ej. uso de `deepcopy`) o deshacer cambios una vez que se comprueba que un llamado recursivo no encontró solución.

Para cada ítem: se entrega puntaje total si se cumple el ítem al 100 %, mitad de puntaje si presenta errores menores, puntaje nulo en otro caso.

Posible solución 1

Algoritmo que aplica "fuerza bruta" y prueba todos los casos posibles, independientemente de si son válidos o no.

```python
def encontrar_solucion(tablero: list[list[int]]) -> list[list[int]] | None:
    celda_vacia = encontrar_celda_vacia(tablero)

    ## CASO BASE
    # Si no quedan celdas vacías por completar,
    # se comprueba que el tablero es solución.
    if not celda_vacia:
        for indice in range(9):
            # Se revisa cada una de las filas, columnas y cuadrantes del tablero.
            fila = comprobar_fila(tablero, indice)
            columna = comprobar_columna(tablero, indice)
            cuadrante = comprobar_cuadrante(tablero, indice)

            # Si alguna comprobación falla, entonces está no es la solución
            # y se retorna None.
            if not (fila and columna and cuadrante):
                return

        # Si pasan todas las comprobaciones, entonces el tablero es solución
        # y se retorna.
        return tablero


    ## CASO RECURSIVO
    # Se hace recursión por cada celda vacía a completar. Se prueban los posibles
    # valores y se hace el llamado recursivo por cada uno de ellos.
    x, y = celda_vacia

    for valor in range(1, 10):
        tablero[x][y] = valor
        solucion = encontrar_solucion(deepcopy(tablero))

        if solucion is not None:
            return solucion

    # Si ninguno de los valores encontró una solución, se retorna None.
    # (Retorno por defecto de la última línea de la definición)
```

Posible solución 2

Algoritmo con "poda" para evitar llamados recursivos con casos inválidos.

```python
def encontrar_solucion(tablero: list[list[int]]) -> list[list[int]] | None:
    celda_vacia = encontrar_celda_vacia(tablero)

    ## CASO BASE
    # Si no quedan celdas vacías por completar, se retorna el tablero.
    # En los llamados anteriores se comprobó que ningún valor utilizado fuese inválido.
    if not celda_vacia:
        return tablero

    ## CASO RECURSIVO
    # Se hace recursión por cada celda vacía a completar. Se prueban los posibles
    # valores y solo se hace el llamado recursivo si el valor utilizado es válido.
    x, y = celda_vacia

    for valor in range(1, 10):
        tablero[x][y] = valor

        fila = comprobar_fila(tablero, x)
        columna = comprobar_columna(tablero, y)
        cuadrante = comprobar_cuadrante(tablero, encontrar_cuadrante(x, y))

        # Si todas las comprobaciones son válidas, hacemos el llamado recursivo.
        if fila and columna and cuadrante:
            solucion = encontrar_solucion(deepcopy(tablero))

            if solucion is not None:
                return solucion

    # Si no fue posible rellenar la celda con un valor o ninguno
    # de los valores encontró una solución, se retorna None.
    # (Retorno por defecto de la última línea de la definición)
```