1. **[2p]** Dado el siguiente código Java,
   a) Dibujar un diagrama de clases que muestre todas las clases/interfaces involucradas (salvo `Object`) y sus relaciones. No es necesario incluir los atributos y métodos, a menos que sean relevantes para mostrar las relaciones entre las clases.
   b) Justificar: ¿Las clases corresponden a la capa *modelo* o *presentación*?

Personaje.java

```java
public abstract class Personaje {
  protected Vector2 posicion = new Vector2(0, 0);
  protected final String nombre;

  protected Personaje(String nombre) {
    this.nombre = nombre;
  }

  public Vector2 getPosicion() {
    return posicion;
  }

  public void setPosicion(Vector2 nuevaPosicion) {
    this.posicion = nuevaPosicion;
  }

  public abstract void actualizar(Mundo mundo);
}
```

Mundo.java

```java
public interface Mundo {
  public void agregarPersonaje(Personaje personaje);
  public Personaje obtenerPersonaje(String nombre);
  public void eliminarPersonaje(String nombre);
}
```

Mario.java

```java
public class Mario extends Personaje {
  private Entrada entrada;

  public Mario(Entrada entrada) {
    super("Mario");
    this.entrada = entrada;
  }

  @Override public void actualizar(Mundo mundo) {
    if (entrada.teclaPresionada(IZQUIERDA)) {
      posicion = posicion.mover(-1, 0);
    }
    // ...
  }
}
```

Koopa.java

```java
public class Koopa extends Personaje {
  public Koopa() { super("Koopa"); }

  @Override public void actualizar(Mundo mundo) {
    if (mundo.getPersonaje("Mario").getPosicion().x <
    posicion.x) {
      posicion = posicion.mover(-1, 0);
    }
    // ...
  }
}
```

Criterios de corrección:

- En el diagrama de clases verificar principalmente que usen bien las flechas de relaciones (composición, generalización, etc). No es tan importante la diferencia entre composición y agregación.
- No es necesario que escriban los nombres de atributos y métodos.
- Las clases corresponden a la capa modelo, ya que representan entidades del dominio del problema (personajes y mundo) y no tienen lógica de presentación o interfaz de usuario.
- Verificar que usen los términos correctos de OOP en la justificación.

2. **[1p]** Considerar el siguiente código. Identificar un principio de diseño que se viola, indicar cuál es y justificar. Proponer una solución para evitar la violación del principio (no es necesario escribir el código).

```java
public class Archivo {
    void abrir() { ... }
    void cerrar() { ... }
    byte[] leer(int n) { ... }
    void escribir(byte[] datos) { ... }
}

public class ArchivoSoloLectura extends Archivo {
    @Override
    void escribir(byte[] datos) {
        throw new UnsupportedOperationException("El archivo es de solo lectura");
    }
}
```

Criterios de corrección:

- Verificar que identifiquen correctamente el principio de diseño violado (Principio de Sustitución de Liskov pero podría ser otro si está bien justificado).
- De nuevo, verificar que usen los términos correctos en la justificación y en la solución propuesta.

3. **[2p]** Escribir en Prolog la regla `ultimo/2` que relaciona una lista con su último elemento. Por ejemplo:

```
?- ultimo([a, b, c, d], X).
X = d.
```

Criterios de corrección:
Solución:

```
ultimo([X], X).
ultimo([_|Xs], Y) :- ultimo(Xs, Y).
```

4. **[2p]** Dado el siguiente código Clojure, justificar:
   a) ¿Cuál o cuáles funciones definidas en el código (con nombre o anónimas) son puras/impuras?
   b) ¿Cuál o cuáles funciones definidas en el código (con nombre o anónimas) son de orden superior?

```clojure
(defn crear-agenda [] {})

(defn agregar-contacto [agenda nombre numero]
  (assoc agenda nombre numero))

(defn obtener-numero [agenda nombre]
  (get agenda nombre))

(defn recorrer [agenda f]
  (doseq [[nombre numero] agenda]
    (f nombre numero)))

(defn imprimir-agenda [agenda]
  (recorrer agenda
            (fn [nombre numero]
              (println (str nombre ": " numero)))))

(defn main []
  (let [agenda (crear-agenda)
        agenda (agregar-contacto agenda "Alan" "1234")
        agenda (agregar-contacto agenda "Barbara" "5678")]
    (imprimir-agenda agenda)))
```

Criterios de corrección:

- Funciones puras: `crear-agenda`, `agregar-contacto`, `obtener-numero`.
- `recorrer` es pura en sí misma, pero su pureza depende de la función `f` que se le pase como argumento.
- Impuras: `imprimir-agenda`, función anónima dentro de `imprimir-agenda`, `main` (usan `println` directamente o indirectamente).
- Funciones de orden superior: `recorrer` (toma una función `f` como argumento).

5. **[2p]** Dada la siguiente expresión lambda,

   a) Identificar las variables libres y ligadas.

   b) Mostrar los pasos de reducción usando orden normal (β-redex más externa desde la izquierda) hasta llegar a la forma normal (si existe). Aplicar reducciones α según sea necesario para evitar conflictos de nombres.

   ```
   (((λx.(x x)) (λy.(y x))) w)
   ```

   Criterios de corrección:
   https://projectultimatum.org/cgi-bin/lambda?t=(((%CE%BBx.(x%20x))%20(%CE%BBy.(y%20x)))%20w)&r=&m=normal%20order

   - No es necesario que dibujen el árbol sintáctico, solo identificar las variables libres y ligadas.
   - El resultado en forma normal es `x x w`.

6. **[1p]** Dado el siguiente código JavaFX:

   a) Dibujar el árbol correspondiente al Scene Graph.

   b) Explicar brevemente el funcionamiento del programa, en términos de programación GUI y orientada a eventos. ¿Qué sucede cuando el usuario interactúa con la interfaz? ¿Cómo se logra este comportamiento?

```java
public class App extends Application {
    @Override
    public void start(Stage stage) throws Exception {
        Label lbl = new Label("Nombre:");
        TextField nombre = new TextField();

        ListView<String> inscriptos = new ListView<>();

        Button btnAgregar = new Button("Agregar");
        btnAgregar.setOnAction(_ -> {
            inscriptos.getItems().add(nombre.getText());
        });

        Button btnQuitar = new Button("Quitar");
        btnQuitar.disableProperty().bind(
            Bindings.isEmpty(
                inscriptos.getSelectionModel().getSelectedItems()
            )
        );
        btnQuitar.setOnAction(_ -> inscriptos.getItems().remove(
                inscriptos.getSelectionModel().getSelectedIndex()
        ));

        stage.setScene(new Scene(new VBox(
                new HBox(lbl, nombre, btnAgregar),
                inscriptos,
                btnQuitar
        )));
        stage.show();
    }
}
```

Criterios de corrección:

```
Stage
  |
  +-- Scene
        |
        +-- VBox
              |
              +-- HBox
              |     |
              |     +-- Label
              |     +-- TextField (nombre)
              |     +-- Button (Agregar)
              |
              +-- ListView (inscriptos)
              |
              +-- Button (Quitar)
```

Verificar que usen los términos correctos en la explicación:

- Cuando el usuario presiona el botón “Agregar”, se ejecuta el handler asociado al evento `onAction`, que agrega el texto del TextField `nombre` a la lista de items del ListView `inscriptos`.
- Cuando el usuario selecciona un nombre en el ListView, el botón “Quitar” se habilita (si hay una selección) o se deshabilita (si no hay selección) gracias al binding con la propiedad `disableProperty`.
- Al presionar el botón “Quitar”, el nombre seleccionado se elimina de la lista, mediante el handler asociado al evento `onAction`.