Prueba Tecnica Academia Epidata - Programa de capacitación en ReactJS.Java

Ejercicio

Resuelva los siguientes ejercicios en los campos de texto provistos.
Recuerde que no es necesario resolver todos los ejercicios.

Consideraciones

1. Si bien el enunciado hace referencia a Java, se puede resolver con cualquier otro lenguaje de programación orientado a objetos ACLARANDO QUÉ LENGUAJE FUE USADO. En el caso de lenguajes multiparadigma, utilizar solo los mecanismos orientados a objetos.
2. Se recomienda que los ejercicios sean resueltos en un editor local para evitar perdida de datos por fallas de Internet. No es necesario que sea una IDE, se puede resolver en un editor de texto simple.
3. Es importante la legibilidad del código, será leído por un humano.
4. Solo implemente lo pedido en los enunciados. Asuma que el resto del código existe.

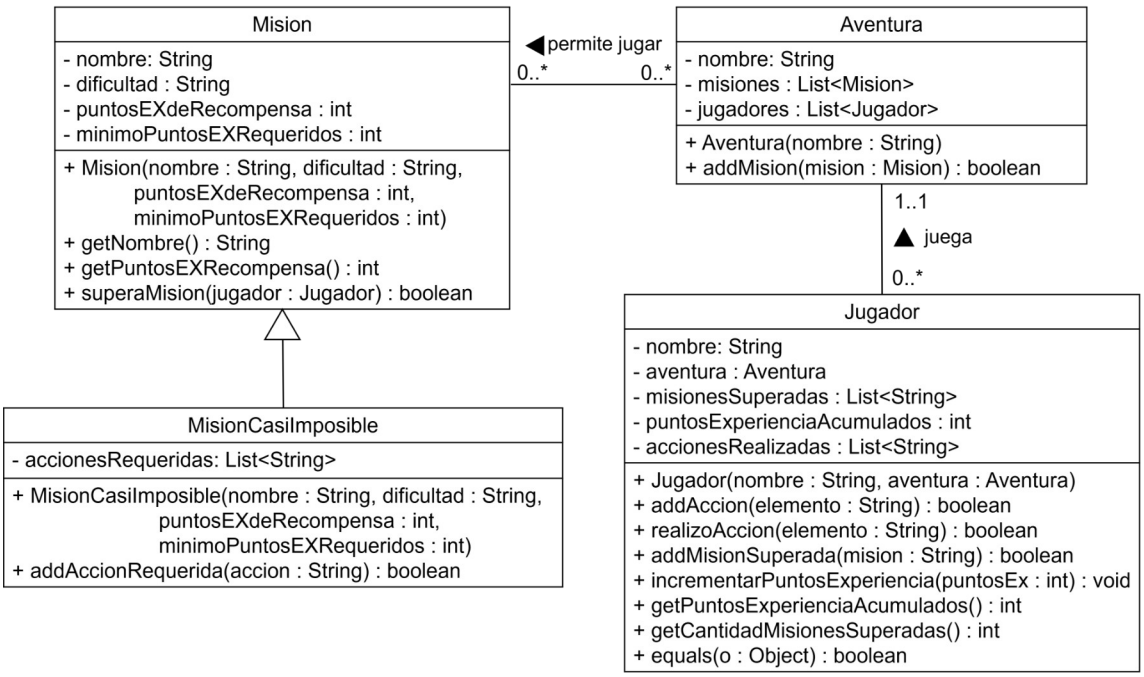
IMPORTANTE: Utilice el lenguaje orientado a objetos con el que se sienta más cómodo. El objetivo de la evaluación no es evaluar los conocimientos de un lenguaje en particular, sino el conocimiento del paradigma de programación orientada a objetos.

¿Qué lenguaje de programación utilizarás para resolver los ejercicios?

Java

Narrativa

Se tiene un sistema que modela un juego. Hasta el momento, el sistema administra jugadores y misiones. Cada jugador conoce su nivel de experiencia, acciones que realizó y misiones que superó. Por su lado, las misiones registran el nombre, dificultad, puntos de experiencia de recompensa y puntos de experiencia requeridos para realizar la misión. Además las misión es responsable de saber si un jugador supera la misión o no, basado solamente en los puntos de experiencia.
El administrador de juego desea agregar otro tipo de misiones, llamadas misiones casi imposibles, donde además de los puntos de experiencia, se desea realizar otros controles adicionales sobre el jugador. El diseñador modificó el diagrama y nos dejó instrucciones para implementar esta nueva característica del sistema.



NOTA

Si bien el enunciado hace referencia a Java, se puede resolver con cualquier otro lenguaje de programación orientado a objetos ACLARANDO QUÉ LENGUAJE FUE USADO. En el caso de lenguajes multiparadigma, utilizar solo los mecanismos orientados a objetos.

Ejercicio 1)

Tenemos un diagrama de clases preliminar de un sistema y hay que implementar la clase **`MisionCasiImplosible`** de acuerdo con el diagrama. El resto de las clases serán implementadas por otros desarrolladores.

`addAccionRequerida(elemento: String): boolean` solo agrega una acción requerida si esta no se encuentra en la lista de acciones requeridas. Tener en cuenta que el método debe retornar **`true`** si la acción pudo ser agregada y **`false`** en caso contrario.

Una vez que se haya implementado la clase, se ejecutará el siguiente código, el cual no debe tener errores de ejecución:

```
MisionCasiImposible m1 = new MisionCasiImposible("Mision CI", "difícil", 100, 200);
m1.addAccionRequerida("llegar al castillo");
```

```
public boolean addAccionRequerida(String accion){
    for(String action : this.accionesRequeridas){
        if (action.equals(accion)) return false;
    }
    this.accionesRequeridas.add(accion);
    return true;
}
```

Ejercicio 2)

En principio, para superar una **`Mision`**, el Jugador tiene que haber acumulado más puntos de experiencia que el **`minimoPuntosEXRequeridos`** requerido por la misión. Para ello, definieron el siguiente método en la clase **`Mision`**:

```
public boolean superaMision(Jugador jugador){
      return minimoPuntosEXRequeridos < jugador.getPuntosExperienciaAcumulados();
}
```

Ahora, para superar una **`MisionCasiImposible`** se requiere cumplir más condiciones. En este caso, el **`Jugador`** no solo debe haber acumulado más puntos de experiencia que el mínimo requerido por la misión, sino que también debe haber realizado todas las acciones requeridas por la misión (representadas por **`accionesRequeridas: List<String>`**). En caso de que la segunda condición no se cumpla, para superar la **`MisionCasiImposible`** la cantidad de acciones que le faltan cumplir al jugador debe ser menor a la cantidad de misiones superadas por el **`Jugador.`**

Tenga en cuenta el siguiente método de la clase **`Jugador`**:

```
public boolean realizoAccion(String elemento){
      return accionesRealizadas.contains(elemento);
}
```

Implemente los métodos necesarios para incorporar la modificación solicitada. Indique a qué clases pertenecen los métodos implementados.

```java
// Class : misionCasiImposible
public boolean superarMision(Jugador jugador){
    // 1ra condicion : el jugador debe acumular mas puntos de experiencia que el minimo requerido
    if(this.minimoPuntoExRequeridos > jugador.getPuntosExperienciaAcumulados()){
        return false;
    }
    // 2da condicion : el jugador debe haber realizado todas las acciones requeridas de la mision
    boolean todasAccionesRealizadas = true;
    int cantidadNoSuperadaPorElJugador = 0;
    for(String action : this.accionesRequeridas){
        if (jugador.realizoAccion(action) == false){
            todasAccionesRealizadas = false;
            cantidadNoSuperadaPorElJugador++;
        }
    }
    // 3ra condicion : si 2da condicion = false, acciones que le faltan cumplir debe ser menor a
    // cantidad de misiones superadas para superarMision.
    if(todasAccionesRealizadas == false){
        if (cantidadNoSuperadaPorElJugador < jugador.accionesRealizadas.length()){
            return true;
        }
        return false;
    }
    return true;
}
```

Ejercicio 3a)

Una vez implementados las **Misiones** y **MisionesCasiImposibles**, se quiere implementar la funcionalidad correspondiente a agregar un **Jugador** a una **Aventura**. Para ello quieren implementar el método **public boolean agregarJugador(jugador: Jugador): boolean**. Para poder agregar un **Jugador** a la **Aventura**, este no debe haber sido previamente agregado.

Implementar esa funcionalidad teniendo en cuenta:

Si el jugador ya existía:
    Retornar **false**

Si el jugador pudo ser agregado:
    Por cada misión de la aventura:
        Si el jugador **superaMision**:
            Agregarle al jugador el nombre de la misión superada.
            Si la misión pudo ser agregada (**addMisionSuperada** retornó **true**), incrementar los puntos de experiencia del jugador de acuerdo a los puntos retornados por la misión.
    Retornar **true**

```java
// Class : Aventura
public boolean agregarJugador(Jugador jugador){
    // check si el jugador ya existe en la aventura
    for(Jugador player : this.jugadores){
        if (jugador.equals(player)) return false;
    }
    // agregamos el jugador y actualizamos sus misiones superadas e incrementamos puntos de ex.
    this.jugadores.add(jugador);
    for(Mision mision : this.misiones){
        if (mision.superaMision(jugador)) {
            //agregar el nombre de la mision superada al jugador e incrementamos los puntos de experiencia del jugador
            jugador.addMisionSuperada(mision.nombre);
            jugador.incrementarPuntosExperiencia(mision.puntosEXdeRecompensa);
        }
    }
    return true;
}
```

Ejercicio 3b)

Describa que fue necesario tener en cuenta para que el código desarrollado en el punto 3a pueda soportar tanto misiones y misiones casi imposible. ¿Cómo se determina que lógica utilizar para saber si se superó la misión?

```
// Usamos herencia de clases, para que la clase 'misionCasiImposible' pueda heredar todos los atributos y metodos de la clase 'Mision'
// ademas podemos ver que sobreescribimos el metodo 'superaMision' dado que superarMision de 'Mision' != superarMision de 'misionCasiImposible'
// por los requerimientos a cumplir del ejercicio 2). De esta forma cubrimos el soporte para ambas clases en el ejercicio 3.a).
```

Ejercicio 4)

En el sistema encontramos el siguiente método en el que no fueron muy claros con los nombres de métodos y variables.

```
public List<String> metodoSinNombre(){
      List<String> variableSinNombre = new ArrayList<>();
      for(Mision m : misiones){
            boolean booleanSinNombre = false;
            for(Jugador j : jugadores)
                  if(m.superaMision(j))
                        booleanSinNombre = true;
            if(!booleanSinNombre)
                  variableSinNombre.add(m.getNombre());
      }
      return variableSinNombre;
}
```

```
public static void main(String[] args) {

      Mision m1 = new Mision("buscando la olla del duende","facil",100,0);
      Mision m2 = new Mision("lobo está?","intermedio",200,100);

      Mision m5 = new MisionCasiImposible("san jorge un poroto","dificil",500,300);
      ((MisionCasiImposible)m5).addAccionRequerida("derrotar 10 dragones")

      Aventura j1 = new Aventura("D&D");
      j1.addMision(m1);
      j1.addMision(m5);
      j1.addMision(m2);

      Jugador player1 = new Jugador("Juan",j1);

      Jugador player2 = new Jugador("Sebastián",j1);
      player2.addAccion("derrotar 10 dragones");

      j1.addJugador(player1);
      j1.addJugador(player2);

      System.out.println(j1.metodoSinNombre());

}
```

¿Cuál es la salida de ejecutar dicho código? NOTA: Java imprime las listas de string con el siguiente formato: [String1, String2, String3]

```
["lobo esta?", "san jorge un poroto"]
```

Documente la funcionalidad del metodoSinNombre

```
// El metodo recorre todos las misiones y para cada una de ellas si hay un jugador (de la bolsa de jugadores) que no supera dicha mision en
// particular entonces agrego el nombre de la mision a la lista de misiones que luego retorna.
// Dicho de otra forma retorna las misiones que aun no fueron superadas por todos los jugadores.
// "lobo esta?" se encuentra en la lista porque ambos jugadores no superan el minimoDePuntosdeExRequeridos (inician con 0 ambos jugadores).
// "san jorge un poroto" de la Class misionCasiImposible tambien lo agrega ya que tiene que cumplir mas condiciones pero, dado que no cumple
// superar mas puntos de experiencia que el minimo requerido (condi. 1) entonces tambien lo agrega.
```

Ejercicio 5) - Un poco de algoritmia

Dada una lista de enteros y un entero objetivo se debe retornar la lista de todos pares, sin repetir, de índices tal que la suma de los enteros en la lista con esos índices sea igual al valor objetivo.

Por ejemplo, dado:

```
L = [2, 6, 8 , 3, 5, 0, 2, 6]
T = 8
```

el método debe devolver:

```
[(0, 1), (0, 7), (1, 6), (2, 5), (3, 4), (6, 7)]
```

El primer elemento `(0,1)` es porque `L[0] + L[1] = 2 + 6 = 8`.
El segundo elemento `(0,7)` es porque `L[0] + L[7] = 2 + 6 = 8`.
El tercer elemento `(1,6)` es porque `L[1] + L[6] = 6 + 2 = 8`.
El cuarto elemento `(2,5)` es porque `L[2] + L[5] = 8 + 0 = 8`.
...

Nótese que el par `(1,0)` también suma 8, pero cómo se encuentra el par `(0,1)` se considera repetido.

```java
public static List<List<Integer>> metodo(List<Integer> list, int obj){
    List<List<Integer>> output = new ArrayList<List<Integer>>();
    for (int i=0; i<list.size(); i++){
        for (int j=i+1; j<list.size(); j++){
            // pares sin repetir
            int sum = list.get(i) + list.get(j);
            if(sum == obj){
                List<Integer> par = new ArrayList<Integer>();
                par.add(i);
                par.add(j);
                output.add(par);
            }
        }
    }
    System.out.println(output);
    return output;
}
```