Tercera práctica
Implementación de la interacción de objetos

Segunda sesión: Implementar en Java y Ruby las operaciones principales del sistema propuesto, partiendo de los diagramas UML de comunicación o secuencia que se proporcionan en swad en la carpeta /Diagramas. A continuación se proporciona una descripción de los métodos involucrados en estas operaciones. Es importante tener en cuenta que la implementación que se haga de las mismas debe seguir escrupulosamente los diagramas.

NOTAS:

- Si en algunos de los diagramas hay alguna operación que no está ya implementada, tendrás que implementarla.
- En el Diagrama de Clases en la clase `Player` se ha cambiado el valor devuelto por `computeGoldCoinsValue(t : NapakalakiGame.Treasure []) : float` siendo un `float` y no un `int`.

En Napakalaki:

- **initGame(players:String[ ])** ***(Diagrama: initGame)***

  Se encarga de solicitar a `CardDealer` la inicialización de los mazos de cartas de Tesoros y de Monstruos, de inicializar los jugadores proporcionándoles un nombre y de iniciar el juego con la llamada a `nextTurn()` para pasar al siguiente turno (que en este caso será el primero).

- **nextTurn():boolean** ***(Diagrama: nextTurn)***

  Esta operación usa el método `nextTurnIsAllowed()`, para verificar si el jugador activo (`currentPlayer`) cumple con las reglas del juego para poder terminar su turno.
  En caso el caso que `nextTurnIsAllowed()` devuelva `true`, se actualiza el jugador activo al siguiente jugador y se le solicita a `CardDealer` el siguiente monstruo al que se enfrentará ese jugador (`currentMonster`).
  En caso de que el nuevo jugador activo esté muerto, por el combate en su anterior turno o porque es el primer turno, se inicializan sus tesoros siguiendo las reglas del juego. La inicialización de los tesoros se encuentra recogida en el diagrama subordinado ***initTreasures***.

- **discardVisibleTreasure(treasures:Treasure[]):void** ***(Diagrama: discarVisibleTreasure)***

  Operación encargada de eliminar los tesoros visibles indicados de la lista de tesoros visibles del jugador. Al eliminar esos tesoros, si el jugador tiene un mal rollo pendiente, se indica a éste dicho descarte para su posible actualización.

- **discardHiddenTreasure(treasures:Treasure[]):void ()** ***(Diagrama: igual que el anterior)***

  Análoga a la operación anterior aplicada a tesoros ocultos. Realizar el correspondiente diagrama de secuencia.

- **developCombat()** ***(Diagrama: developCombat)***

  Operación responsabilidad de la única instancia de `Napakalaki`, la cual pasa el control al jugador actual (`currentPlayer`) para que lleve a cabo el combate con el monstruo que le ha tocado (`currentMonster`). El método correspondiente de la clase Player para llevar a cabo dicha responsabilidad es `combat(currentMonster:Monster): CombatResult`, cuyo algoritmo general (también reflejado en el diagrama y responsabilidad de Player) es :

```
Si el nivel de jugador > nivel del monstruo {
      Se actualiza el nivel y tesoros del jugador y
                  se invoca al operación applyPrize (Diagrama: applyPrize)
      Devuelve Win si el jugador no gana la partida (nivel del jugador <10).
      Devuelve WinAndWinGame si el jugador gana la partida.
   }
Si el nivel del jugador es <= nivel del monstruo {
            Se lanza el dado
            Si sale 5 ó 6, {
                  No pasa nada y se devuelve LooseAndEscape
            } en otro caso  {
                  Se analiza en qué consiste el mal rollo
                  Si el jugador muere {
                        Se invoca a la operación die  (Diagrama: die)
                        Se devuelve LoseAndDie
                              } en otro caso {
                              Se invoca a  la operación applyBadConsequence
                                    (Diagrama: applyBadConsequence)
                              Se devuelve Lose
                           }
             }
  }
```

- **buyLevels(visible: Tresure[ ], hidden: Treasure[ ]):boolean** ***(Diagrama: buyLevels)***

  Esta operación permite comprar niveles antes de combatir con el monstruo actual. Para ello, a partir de las listas de tesoros (pueden ser tanto ocultos como visibles) se calculan los niveles que puede subir el jugador en función del número de piezas de oro que sumen. Si al jugador le está permitido comprar la cantidad de niveles resultantes (no se puede comprar niveles si con ello ganas el juego), entonces se produce el mencionado incremento. Independientemente de si se ha podido comprar niveles o no, los tesoros empleados para ello pasan al mazo de descartes.

- **makeTreasuresVisible(treasures:Treasure[1..*]):void** ***(Diagrama: makeTreasuresVisible)***

  Operación en la que el jugador pasa tesoros ocultos a visibles, siempre que pueda hacerlo según las reglas del juego, para ello llama al método, para ello desde Player se ejecuta el método: ***canMakeTreasureVisible(treasures:Treasure):boolean***

En Player:

- **applyPrize(currentMonster : Monster) : void** ***(Diagrama: applyPrize)***

  Cuando el jugador gana el combate, esta operación es la encargada de aplicar el buen rollo al jugador, sumando los niveles correspondiente y robando los tesoros indicados en el buen rollo del monstruo.

- **die():void** ***(Diagrama: die)***

  Cuando el jugador muere en un combate, esta operación es la encargada de dejarlo sin tesoros, ponerle el nivel a 1 e indicar que está muerto, en el atributo correspondiente.

- **applyBadConsequence(b: BadConsequence)** ***(Diagrama: applyBadConsequence)***

  Cuando el jugador ha perdido el combate, no ha podido huir y no muere, hay que almacenar el mal rollo que le impone el monstruo con el que combatió. Para ello, decrementa sus niveles según indica el mal rollo y guarda una copia de un objeto badConsequence ajustada a los tesoros que puede perder según indique el mal rollo del monstruo y de los tesoros que disponga el jugador, la operación encargada de hacer esto es `adjustToFitTreasureList` de la clase badConsequence, este es el mal rollo pendiente `(pendingbadConsequence)` que el jugador almacenará y que deberá cumplir descartándose de esos tesoros antes de que pueda pasar al siguiente turno.

- **initTreasures() : void** ***(Diagrama: initTreasures)***

  Cuando un jugador está en su primer turno o se ha quedado sin tesoros ocultos o visibles, hay que proporcionarle nuevos tesoros para que pueda seguir jugando. El número de tesoros que se les proporciona viene dado por el valor que saque al tirar el dado:

  - `Si (dado == 1) roba un tesoro.`
  - `Si (1 < dado< 6) roba dos tesoros.`
  - `Si (dado == 6) roba tres tesoros.`

En BadConsequence:

- **adjustToFitTreasureLists(v :Treasure [], h : Treasure []) : badConsequence**

  Recibe como parámetros los tesoros visibles y ocultos de los que dispone el jugador y devuelve un nuevo objeto mal rollo que contiene una lista de tipos tesoros visibles y ocultos de los que el jugador debe deshacerse. El objeto de mal rollo devuelto por `adjustToFitTreasureList` solo contendrá tipos de tesoros y en una cantidad adecuada a los que el jugador puede llegar a deshacerse en función de los que dispone.