Segunda Práctica
Implementación del diseño de una estructura de clases

Competencias específicas de la segunda práctica

- Interpretar correctamente diagramas de clases del diseño en UML.
- Implementar el esqueleto de clases (cabecera de la clase, declaración de atributos y declaración de métodos) en Java y en Ruby, y relacionarlas adecuadamente a nivel de implementación.
- Implementar algunos de los métodos simples de las clases.

A) Programación de la segunda práctica

Objetivos
• Ubicar las clases y relaciones implementadas en la primera práctica dentro del diagrama de clases UML dado. • Identificar y definir las nuevas clases. • Declarar los atributos básicos de cada clase. • Declarar los atributos de referencia (implementan asociaciones entre clases) de cada clase. • Declarar los métodos constructores, consultores y modificadores de cada clase. • Declarar otros métodos que aparezcan en el diagrama de clases UML. • Implementar todos los métodos constructores. • Conocer y utilizar el patrón de diseño Singleton. • Implementar los métodos con funcionalidad simple especificados.

B) Primera Sesión:

B.1) Implementa en Java el diagrama de clases proporcionado como DClasesNapakalaki.pdf, siguiendo las siguientes directrices:

1) Declara todos los atributos básicos teniendo en cuenta, además de su tipo primitivo, su visibilidad (si son private, package, protected o public), y su ámbito (si son de instancia o de clase (static)).

2) Declara todos los métodos teniendo en cuenta: parámetros, valor de retorno, visibilidad y ámbito de acceso.

3) Identifica los atributos de referencia a partir de las asociaciones existentes entre las clases. Utiliza la clase ArrayList para guardar colecciones de objetos.

4) Implementa los constructores de todas las clases prestando atención a que los objetos

queden correctamente inicializados.

5) Declara cada una de las nuevas clases que aparecen en el diagrama, teniendo en cuenta cuáles de ellas son **<<singleton>> o <<enumeration>>** .

Se dice que una clase es un *singleton* cuando sólo puede tener una instancia. Para conseguirlo utilizamos el patrón de diseño singleton. Una de las formas de implementarlo es la siguiente (suponiendo que Miclase es la clase singleton):

```
public class MiClase {

   private static final MiClase instance = new MiClase();

  // El constructor privado asegura que no se puede instanciar
  // desde otras clases
  private MiClase() { }

  public static MiClase getInstance() {
     return instance;
  }
}
```

Más información: http://en.wikipedia.org/wiki/Singleton_pattern

B.2) Siguiendo lo dispuesto en el apartado B.1, implementa en Ruby el diagrama de clases proporcionado. Ten en cuenta que en Ruby:

- No hay declaración explícita de variables.
- Los atributos y métodos solo admiten tres controles de acceso: public, protected y private.
- Los atributos de instancia se pueden crear en cualquier método de instancia y anteponiendo a su nombre @, lo recomendable es crearlas e inicializarlas dentro del método initialize. Plantéate si es necesario utilizar algún modificador de acceso (attr_accesor, attr_reader o attr_writer) y en tal caso, cuál es el más adecuado.
- Los atributos de clase se pueden crear en cualquier lugar del código asociado a la clase y anteponiendo a su nombre @@.
- Los métodos de clase se declaran usando el nombre de la clase o self para indicar que solo ella puede ejecutar el método,

```
class Ejemplo
def Ejemplo.métodoDeClase
  o
def self.métodoDeClase
```

- El patrón Singleton en Ruby ya está implementado en el módulo Singleton y lo único que tenemos que hacer es usarlo, por ejemplo, si la clase MiClase es un

singleton tenemos que:

- En MiClase incluir el módulo Singleton de la siguiente forma:

```
include Singleton
```

- Donde necesitemos acceder a la instancia de MiClase:

```
mc = MiClase.instance
```

- Para trabajar con colecciones puedes usar Array.

Nota: Tanto en Java como en Ruby, revisa las clases BadConsequence, Monster y Prize desarrolladas en la práctica anterior, para que estén conforme con lo que se indica en el diagrama de clases proporcionado.

C) Métodos básicos:

Implementa los siguientes métodos básicos, que describimos en lenguaje natural, en Java y en Ruby. Dejaremos para la siguiente sesión los métodos más complejos.

1) En la clase **Player**

- **isDead():boolean**

  Devuelve `true` si el jugador está muerto, `false` en caso contrario.

- **getName():String**

  Devuelve el nombre del jugador.

- **getCombatLevel():int**

  Devuelve el nivel de combate del jugador, que viene dado por su nivel más los bonus que le proporcionan los tesoros que tenga equipados.

- **bringToLife():void**

  Devuelve la vida al jugador, modificando el atributo correspondiente.

- **incrementLevels(i:int):void**

  Incrementa el nivel del jugador en `i` niveles.

- **decrementLevels(i:int):void**

  Decrementa el nivel del jugador en `i` niveles.

- **setPendingBadConsequence(b:BadConsequence):void**

  Asigna el mal rollo al jugador, dándole valor a su atributo pendingBadConsequence.

- **dieIfNoTreasures():void**

  Cambia el estado de jugador a muerto, modificando el correspondiente atributo. (Esto ocurre cuando el jugador, por algún motivo, ha perdido todos sus tesoros.)

- **validState():boolean**

  Devuelve true cuando el jugador no tiene ningún mal rollo que cumplir y no tiene más de 4 tesoros ocultos, y false en caso contrario. Para comprobar que el jugador no tenga mal rollo que cumplir, utiliza el método isEmpty de la clase BadConsequence.

- **howManyVisibleTreasures(tKind:TreasureKind):int**

  Devuelve el número de tesoros visibles de tipo `tKind` que tiene el jugador.

- **getLevels():int**

  Devuelve el nivel del jugador.

- **setEnemy(Player enemy): void**

  Asigna valor al atributo que referencia al enemigo del jugador.

- **canISteal():boolean**

  Devuelve el valor de la variable canISteal, que indica si el jugador ha robado antes o no un tesoro a su enemigo.

- **haveStolen():void**

  Cambia el atributo canISteal a `false` cuando el jugador roba un tesoro.

- **canYouGiveMeATreasure(): boolean**

  Devuelve `true` si el jugador tiene tesoros para ser robados por otro jugador y `false` en caso contrario.

2) En la clase **BadConsequence**

- **isEmpty():boolean**

  Devuelve true cuando el conjunto de atributos del mal rollo indica que no se pierden tesoros. Plantéate qué valores deberán tener sus atributos.

3) En la clase **CardDealer**

- **initTreasureCardDeck()**

Inicializa el mazo de cartas de Tesoros (`unusedTreasures`) con todas las cartas de tesoros proporcionadas en el documento de cartas de tesoros.

- **initMonsterCardDeck()**

  Inicializa el mazo de cartas de monstruos (`unusedMonsters`), con todas las cartas de monstruos proporcionadas en el documento de cartas de monstruos. Se recomienda reutilizar el código desarrollado en la primera práctica para construir las cartas de monstruos. Usa el atributo de clase definido en la clase BadConsequence MAXTREASURES para los monstruos cuyo mal rollo sea pérdida de todos los tesoros visibles u ocultos.

- **shuffleTreasures()**

  Baraja el mazo de cartas de tesoros `unusedTreasures.`

- **shuffleMonsters()**

  Baraja el mazo de cartas de monstruos `unusedMonsters.`

- **giveTreasureBack(t:Treasure)**

  Introduce en el mazo de descartes de tesoros (`usedTreasures`) el tesoro `t.`

- **giveMonsterBack(m:Monster)**

  Introduce en el mazo de descartes de monstruos (`usedMonsters`) al monstruo `m.`

4) En la clase **Dice**

- **nextNumber():int**

  Genera un número aleatorio entre 1 y 6 (ambos incluidos).

5) En la clase **Monster**

- **getLevelsGained():int**

  Devuelve el número de niveles ganados proporcionados por su buen rollo.

- **getTreasuresGained():int**

  Devuelve el número de tesoros ganados proporcionados por su buen rollo.

6) **Completa lo que falta de la clase Treasure,** teniendo en cuenta que todos sus métodos son los consultores básicos.

7) **Define el enumerado CombatResult** de la misma forma que se definió TreasureKind.