Tercera práctica

Implementación de la interacción de objetos

Competencias específicas de la tercera práctica

- Interpretar correctamente diagramas de interacción en UML.
- Traducir los diagramas de interacción de UML a Java y a Ruby.

Programación de la tercera práctica

Tiempo requerido: tres sesiones (seis horas).

Planificación y objetivos:

Sesión	**Semana**	**Objetivos**
Primera	Del 10 al 16 de noviembre	• Implementar en Java y Ruby operaciones simples del diagrama de clases visto en la práctica anterior. Las operaciones implican respetar las normas del juego, aplicar estructuras de control de los lenguajes e implementar clases, atributos y métodos.
Segunda	Del 17 al 23 de noviembre	• Saber interpretar un diagrama de interacción de UML, de secuencia o de comunicación. Implica comprender el envío de mensajes y saber interpretar frames y condiciones de control en los diagramas. • Traducir diagramas de interacción a Java y Ruby.
Tercera	Del 24 al 30 de noviembre	• Probar todo el código desarrollado en Java y en Ruby. Implica saber usar el depurador y arreglar los errores detectados.
• El trabajo se realizará tanto en Java como en Ruby. • La práctica se desarrollará en grupo.		

Entrega y **examen** de la tercera práctica: Semana del 1 al 4 de diciembre, los grupos del lunes el 14 de diciembre.

- **Lugar:** Aula en la que se desarrolla la correspondiente sesión de prácticas.
- **Día:** El correspondiente a la sesión de cada grupo en el periodo indicado.
- **Duración**: 30 minutos, al comienzo de la sesión.
- **Tipo examen:** Realizar pequeñas modificaciones y añadidos sobre el propio código.
- Se pedirá entregar el código completo de la tercera práctica con las modificaciones pedidas en el examen.

A) Primera sesión: Implementa los siguientes métodos en Java y en Ruby

En la clase Napakalaki:

- **initPlayers(names:String[]):void**

  Inicializa el array de jugadores que contiene `Napakalaki`, creando tantos jugadores como elementos haya en *`names`*, que es el array de `String` que contiene el nombre de los jugadores.

- **nextPlayer():Player**

  Decide qué jugador es el siguiente en jugar.

  En primer lugar, se calcula el índice que ocupa el siguiente jugador en la lista de jugadores. Se pueden dar dos posibilidades:

  - Que sea la primera jugada, en este caso hay que generar un número aleatorio entre 0 y el número de jugadores menos 1. Este número indicará el índice que ocupa en la lista de jugadores el jugador que comenzará la partida.
  - Que no sea la primera jugada, en este caso el índice es el del jugador que se encuentra en la posición siguiente respecto al jugador actual. Hay que tener en cuenta que si el jugador actual está en la última posición de la lista, el siguiente será el que está en la primera posición.

  Una vez calculado el índice, se devuelve el jugador que ocupa esa posición.

- **getCurrentPlayer():Player**

  Devuelve el jugador actual (`currentPlayer`).

- **getCurrentMonster():Monster**

  Devuelve el monstruo en juego (`currentMonster`).

- **nextTurnIsAllowed():boolean**

  Método que comprueba si el jugador activo (`currentPlayer`) cumple con las reglas del juego para poder terminar su turno. Devuelve `false` si el jugador activo no puede pasar de turno y `true` en caso contrario, para ello usa el método de `Player` *`validState()`* `donde se realizan las comprobaciones pertinentes.`

- **endOfGame(result:CombatResult):boolean**

  Devuelve `true` si el parámetro result es `WINGAME` (valor del enumerado `CombatResult).` En caso contrario devuelve `false`.

- **setEnemies():void**

  Se asigna un enemigo a cada jugador. Esta asignación se hace de forma aleatoria teniendo en cuenta que un jugador no puede ser enemigo de sí mismo.

En la clase Player

- **canMakeTreasureVisible(t:Treasure):boolean**

  Comprueba si el tesoro *t* se puede pasar de oculto a visible según las reglas del juego.

- **giveMeATreasure():Treasure**

  Devuelve un tesoro elegido al azar de entre los tesoros ocultos del jugador.

En la clase BadConsequence

- **substractVisibleTreasure(t:Treasure)**

  Actualiza el mal rollo para que el tesoro visible *t* no forme parte del mismo. Es posible que esta actualización no implique cambio alguno, que lleve a eliminar un tipo específico de tesoro visible, o a reducir el número de tesoros visibles pendientes.

- **substractHiddenTreasure(t:Treasure)**

  Igual que el anterior, pero para los ocultos.

En la clase CardDealer **(puede que algunos ya los tengas implementados):**

- **nextTreasure():Treasure**

  Devuelve el siguiente tesoro que hay en el mazo de tesoros (`unusedTreasures`) y lo elimina de él. Si al iniciar el método el mazo unusedTreasures estuviese vacío, pasa el mazo de descartes (`usedTreasures`) al mazo de tesoros (unusedTreasures) y barájalo, dejando el mazo de descartes vacío.

- **nextMonster():Monster**

  Igual que la anterior pero con el mazo de monstruos.

B) Segunda sesión: Implementa en Java y Ruby las operaciones principales del sistema propuesto, partiendo de los diagramas UML de comunicación o secuencia que se encuentran en swad en la carpeta P3/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.

NOTA:

Si tras implementar los diagramas queda alguna operación sin implementar, tendrás que implementarla.

En la clase Napakalaki

- **initGame(players:String[ ])** ***(Archivo: initGame.pdf)***

  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, asignarle a cada jugador su enemigo y de iniciar el juego con la llamada a `nextTurn()` para pasar al siguiente turno (que en este caso será el primero).

- **nextTurn():boolean** ***(Archivo: nextTurn.pdf)***

  Esta operación usa el método `nextTurnAllowed()`, donde se verifica 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 le solicita a `CardDealer` el siguiente monstruo al que se enfrentará ese jugador (`currentMonster`) y se actualiza el jugador activo al siguiente jugador.
  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** ***(Archivo:discardVisibleTreasure.pdf)***

  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. Finalmente, se invoca a `dieIfNoTreasure()` para comprobar si se ha quedado sin tesoros y en ese caso pasar a estado de muerto. Los tesoros descartados se devuelven al CardDealer.

- **discardHiddenTreasure(treasures:Treasure[]):void ()** ***(Diagrama similar al anterior)***

  Análoga a la operación anterior aplicada a tesoros ocultos.
  Plantéate como ejercicio de refuerzo realizar el diagrama de comunicación correspondiente a esta operación.

- **developCombat()** ***(Archivo: developCombat.pdf)***

  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 de la clase Player con esa responsabilidad es *`combat(currentMonster:Monster): CombatResult`*, cuyo comportamiento general (también reflejado en el diagrama y responsabilidad de Player) es: si el nivel de combate del jugador supera al del monstruo, se aplica el buen rollo y se puede ganar el combate o el juego, en otro caso, el jugador pierde el combate y se aplica el mal rollo correspondiente.

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

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

En la clase Player:

- **applyPrize(currentMonster : Monster) : void** ***(Archivo: applyPrize.pdf)***

  Esta operación es la encargada de aplicar el buen rollo del monstruo vencido al jugador, sumando los niveles correspondientes y pidiendo al CardDealer que le dé el número de tesoros indicado en el buen rollo del monstruo. Esos tesoros se añaden a sus tesoros ocultos.

- **applyBadConsequence(b: BadConsequence)** ***(Archivo: applyBadConsequence.pdf)***

  Cuando el jugador ha perdido el combate, hay que considerar 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. Es decir, un objeto mal rollo que refleje el mal rollo indicado por el monstruo pero eliminando las condiciones que el jugador no pueda cumplir según los tesoros de que disponga (por ejemplo si el mal rollo del monstruo implica perder 2 tesoros visibles y el jugador sólo tiene 1, entonces el mal rollo pendiente será de sólo 1 tesoro visible). La operación encargada de hacer esto es `adjustToFitTreasureList` de la clase badConsequence. Éste 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.

- **stealTreasure():Treasure** ***(Archivo: stealTreasure.pdf)***

  Cuando el jugador decide robar un tesoro a su enemigo, este método comprueba que puede hacerlo (sólo se puede robar un tesoro durante la partida) y que su enemigo tiene tesoros ocultos para ser robados (`canYouGiveMeATreasure()`), en el caso que así sea éste le proporciona un tesoro que se almacenará con sus ocultos. El jugador no puede volver a robar otro tesoro durante la partida. En el caso que no se haya podido robar el tesoro por algún motivo se devuelve null.

- **discardAllTreasures()** ***(Archivo: discardAllTreasures.pdf)***

  El jugador se descarta de todos sus tesoros ocultos y visibles. Para cada tesoro que se descarta se hace uso de la operación `discardVisibleTreasure(t:Treasure)` o `discardHiddenTreasure(t:Treasure)` según corresponda, de esa forma se verifica si se cumple con algún mal rollo pendiente.

- **initTreasures() : void** ***(Archivo: initTreasures.pdf)***

  Cuando un jugador está en su primer turno o se ha quedado sin tesoros, 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 la clase BadConsequence:

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

  (No hay diagrama, tienes que plantearte cómo resolverlo)

  Recibe como parámetros los tesoros visibles y ocultos de los que dispone el jugador y devuelve un nuevo objeto mal rollo que se ajusta a las posibilidades del jugador. Los atributos de BadConsequence que debemos tener en cuenta para ajustar el el mal rollo que debe cumplir el jugador son nVisibleTreasures, nHiddenTreasures, specificVisibleTreasures y specificHiddenTreasures.

  Así, para un BadConsequence con los siguientes valores de esos atributos:

  nVisibleTreasures = 0
  nHiddenTreasures = 3
  specificVisibleTreasures = []
  specificHiddenTreasures = []

  si los parámetros recibidos son v=[ONEHAND, HELMET] y h=[SHOE], el mal rollo devuelto por la operación tendrá como estado:

  nVisibleTreasures = 0
  nHiddenTreasures = 1 (porque el jugador sólo puede llegar a perder un tesoro oculto)
  specificVisibleTreasures = []
  specificHiddenTreasures = []

  Pongamos otro ejemplo. Si el BadConsequence tuviese estado:

  nVisibleTreasures = 0
  nHiddenTreasures = 0
  specificVisibleTreasures = [BOTHHANDS]
  specificHiddenTreasures = [ONEHAND, ONEHAND, HELMET]

  y recibiese los parámetros v=[ONEHAND] y h=[ONEHAND, HELMET, ARMOR], el mal rollo devuelto por la operación tendría como estado:

  nVisibleTreasures = 0
  nHiddenTreasures = 0
  specificVisibleTreasures = []
  specificHiddenTreasures = [ONEHAND, HELMET]

C) Tercera sesión: Ahora que todas las clases del juego están implementadas, ha llegado el momento de probar la aplicación de Napakalaki tanto en Java como en Ruby. Para ello, es necesario jugar unas partidas a dicho juego. De este modo, todos los métodos desarrollados acabarán por ejecutarse y podréis comprobar que funcionan correctamente. Para ello, se requiere una interfaz de usuario con la que los jugadores tomen sus decisiones y jueguen sus cartas. Usando esta interfaz, podréis probar en profundidad el juego y detectar errores en tiempo de ejecución que deberéis depurar.

Para facilitar la tarea se proporcionan dos interfaces textuales, una en Java y otra en Ruby, que deberéis usar como interfaz de usuario de vuestras aplicaciones Napakalaki. El código de las interfaces de usuario está disponible en swad, en los archivos testerNapakalakiJava.zip y testerNapakalakiRuby.zip.

PDOO, curso 2015/16 Guión de la práctica 3

Realiza la siguientes tareas en Java y en Ruby:

1. Comprende las interfaces de usuario proporcionadas

Para comprender, a grandes rasgos, el funcionamiento de las interfaces trata de responder las siguientes cuestiones:

- ¿Para qué sirve y cómo funciona el método *manageMenu*?
- ¿Qué ventajas proporciona usar el enumerado *Command*?
- ¿Cuál es el método que conecta, principalmente, la interfaz con el modelo? Con modelo nos referimos al conjunto de clases que implementan la lógica de negocio de Napakalaki
- ¿Qué instrucción permite leer de teclado en Java? ¿y en Ruby?
- ¿Cómo se "construyen" las frases que debe aparecer en la interfaz de usuario?
- ¿Qué pasaría si se introduce algo que no sea un entero desde teclado? ¿Cómo se resuelve en Java? ¿Cómo se resuelve en Ruby?

No obstante, entender la interfaz facilitada no es requisito imprescindible para realizar correctamente esta práctica. Siguiendo una especificación común (como es el caso del diseño que hemos estado implementado) se pueden conectar módulos de software de modo que el resultado funcione sin problemas sin tener que estudiar los entresijos del software que no se ha desarrollado (en este caso la interfaz).

2. Integra las clases de la interfaz con el modelo

Asegúrate de que los nombres de paquetes en Java (módulos en Ruby), clases y métodos usados en el modelo coinciden exactamente con los utilizados en la interfaz proporcionada. Deberían coincidir si se han seguido fielmente los diagramas UML. Si no es así debes realizar las correcciones oportunas en el modelo.

3. Ejecuta la aplicación

Introduce por la interfaz de usuario los datos que se requieren para jugar a Napakalaki: equiparse, mostrar el monstruo, combatir, descartarse, comprar niveles, etc.

Trata de ser sistemático/a para utilizar todas las opciones que posibilita la interfaz, así evitarás que surjan errores inesperados más adelante.

4. Depurar el código

Probablemente, aparecerán errores durante el paso 3. Algunos errores de programación pueden ser evidentes y fáciles de solucionar, pero es posible que haya otros que no sepáis de dónde vienen. Para rastrear estos últimos, recomendamos utilizar el depurador.

NOTA IMPORTANTE: Tenéis que modificar el constructor de la clase BadConsequence para los malos rollos que suponen muerte, inicializando los siguientes atributos a los valores que se especifica:

levels = Player.MAXLEVEL

nVisibleTreasures = MAXTREASURES

nHiddenTreasures = MAXTREASURES