Cuarta Práctica
Diseño e implementación usando herencia

Competencias específicas de la cuarta práctica

- Interpretar correctamente los diagramas de clases del diseño en UML en los que haya herencia de clases.
- Modificar el código de un programa orientado a objetos en Java y en Ruby de acuerdo con las modificaciones realizadas en un diagrama de clases en el que se añaden clases con relación de herencia.
- Mejorar el diseño de un sistema utilizando mecanismos de reutilización de código.

Programación y objetivos de la cuarta práctica
Tiempo requerido: 2 sesiones (4 horas)

Planificación y objetivos:

Sesión	Semanas	Objetivo general
Primera: Java y Ruby	Grupos de los lunes, jueves y viernes: semana del 28 de noviembre Grupos de los miércoles: 7 de diciembre Grupos de los martes: 13 de diciembre	Interpretar e implementar en Java y Ruby un diagrama de clases que permita el uso de jugadores sectarios
Segunda: Java y Ruby	Grupos de los lunes y viernes: semana del 5 de diciembre Grupos de los miércoles y jueves: semana del 12 de diciembre Grupos de los martes: 20 de diciembre	Mejorar el diseño e implementar en Java y Ruby el diagrama de clases actual considerando la herencia como técnica de reutilización de código
La práctica se desarrollará en grupo, con la misma composición que para las otras prácticas.		

Objetivos específicos[1]:

1.	Ser capaz de modificar un diagrama de clases a partir de un cambio en las reglas del juego que introduzcan una relación “es-un”
2.	Ser capaz de realizar las modificaciones en el código pertinentes para implementar cambios en un diagrama de clases que impliquen el uso de herencia

1 Los relacionados con la programación, se entiende que deben cumplirse en Java y en Ruby

3.	Saber especificar la redefinición de un método
4.	Saber cómo redefinir correctamente un método que amplíe lo que haga el de su superclase
5.	Ser capaz de implementar una clase abstracta

Entrega y examen de la cuarta práctica: semana del 9 de enero cada grupo en su sesión.

Cuarta práctica

El supuesto que vamos a seguir es el mismo que en las prácticas anteriores: el juego de cartas **Napakalaki**. La práctica se desarrollará en los mismos grupos que se formaron para las prácticas anteriores.

PRIMERA SESIÓN:

A) Descripción del problema: Modificación de las reglas del juego

Se incorporan las siguientes modificaciones al juego:

- Aparece un subtipo de jugador: el jugador sectario (*CultistPlayer*). Un jugador se convierte en sectario cuando se cumplen las siguientes condiciones:
  - ha perdido el combate contra un monstruo
  - obtiene un 6 al lanzar de nuevo el dado
- Cuando un jugador adquiere la condición de sectario se le asigna una carta de este tipo (*Cultist*), que se proporcionan junto a este guión. Su carta de sectario afectará al número de niveles que se le añadirán al jugador para combatir contra los futuros monstruos.
- Asimismo se proporcionan nuevos monstruos que incluyen un aumento o reducción del nivel de combate del monstruo cuando se enfrenta a jugadores que sean sectarios.
- Si un jugador va a robar un tesoro a su enemigo, y éste es sectario, se lo robará eligiéndolo aleatoriamente de los tesoros visibles.
- Un jugador sectario nunca podrá dejar de serlo.

En Java no se pueden redefinir métodos privados, por ello, para dar una solución uniforme en ambos lenguajes, declara como *protected* los métodos que vayas a necesitar redefinir.

Si en algún método de *CultisPlayer* se requiere acceder a algún atributo privado de *Player*, y no existe un mecanismo para ello, deberás añadir un consultor con visibilidad protected en la clase Player.

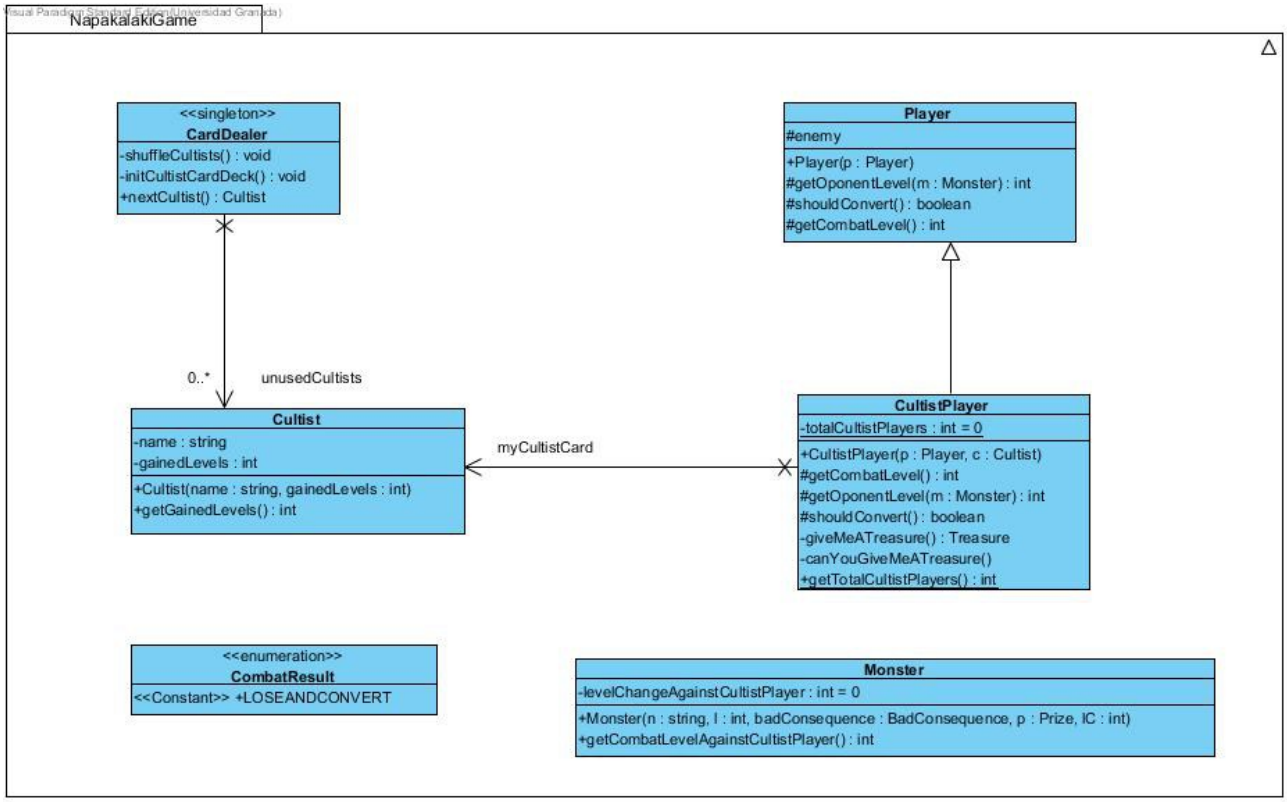

Figura 1: Diagrama clases

B) Tareas a realizar en Java

1. Definición de nuevas clases y métodos e implementación de constructores: define las clases/métodos necesarios según el diagrama de clases proporcionado en la Figura 1. Para ello ten en cuenta que:

- Herencia: Se ha creado una subclase de *Player* llamada *CultistPlayer*.
- Se ha añadido un nuevo atributo en la clase *Monster* para indicar un incremento/decremento de su nivel cuando combate con un sectario (*levelChangeAgainstCultistPlayer*) y por tanto debes añadir un nuevo constructor usándolo en el método *initMonsterCardDeck* de la clase *CardDealer* para crear los nuevos monstruos.
- Se debe cambiar la implementación del constructor anterior de *Monster* para que inicialice la variable *levelChangeAgainstCultistPlayer* con valor *0* para todos los monstruos anteriores.
- Se ha añadido un constructor de copia (superficial) a la clase *Player* para crear un jugador copiando los atributos de otro.
- El constructor de la clase *CultistPlayer* debe llamar al nuevo constructor de copia de *Player* para generar el *CultistPlayer* a partir del jugador. Asimismo, debe incrementar en uno la variable de clase *totalCultistPlayers* y dar valor al atributo *myCultistCard*.
- Se han añadido los métodos *getOponentLevel* y *shouldConvert* en *Player* y son sobreescritos en *CultistPlayer*.
- Se ha creado una clase nueva, *Cultist*, que representa la carta de Sectario.
- Se ha añadido una asociación de *CultistPlayer* a *Cultist*, cuyo rol es *myCultistCard*.

- Se ha añadido una asociación de *CardDealer* a *Cultist*, con rol *unusedCultists*. Se han añadido además los métodos *nextCultist*, de visibilidad pública, y *shuffleCultists* e *initCultistCardDeck*, privados.

2. Creación de nuevas cartas.

- Implementa el método *initCultistCardDeck* de la clase *CardDealer* para que se añadan al mazo de sectarios las 6 cartas de sectarios que aparecen en el documento *cartasSectarios.pdf*.
- Añade al método *initMonsterCardDeck* de la clase *CardDealer* el código necesario para que se añadan al mazo de monstruos las 6 cartas de monstruos nuevas del documento *cartasMonstruosCONSectarios.pdf*.

3. Implementación de métodos.

A continuación damos indicaciones para implementar algunos de los métodos:

- Implementa el método *shuffleCultists* de la clase CardDealer para que baraje las cartas de Cultist.
- Modifica el método *initCards* de *CardDealer* para que incluya una llamada a *initCultistCardDeck*.
- Implementa el método *getCombatLevelAgainstCultistPlayer* de la clase *Monster* para que devuelva la suma del nivel del monstruo (*getLevel*) con el valor del atributo *levelChangeAgainstCultistPlayer*.
- En el método *shouldConvert* de la clase *Player* se lanzará el dado y devolverá true si se obtiene un 6 y false en caso contrario. Su redefinición en *CultistPlayer* devolverá siempre false.
- El método *getCombatLevel* se redefine en la clase *CultistPlayer* para calcular su nivel de combate sumando el *getCombatLevel* de *Player* más un 70% del mismo, más lo que devuelva el ḿetodo *getGainedLevels* de su carta de sectario (*Cultist*) multiplicado por el número de sectarios en la partida. La parte decimal del resultado debe truncarse.
- Se debe añadir el valor *LOSEANDCONVERT* al enumerado *CombatResult* para reflejar la situación de un jugador que pierde el combate contra el monstruo y debe convertirse a sectario.
- El método *combat* de la clase *Player* debe modificarse para que una vez que el jugador activo haya combatido y haya perdido:
    - Se envíe a sí mismo el mensaje *shouldConvert*
    - Si el mensaje *shouldConvert* devuelve true devuelva el valor *LOSEANDCONVERT*
- El método *combat* de *Player* también se modificará para que, en vez de preguntar al propio monstruo al que se enfrenta su nivel de combate con el método *getCombatLevel*, el jugador se pregunte a sí mismo con el método *getOponentLevel*.
- El método *getOponentLevel* debe definirse en Player y redefinirse en *CultistPlayer*. En *Player* hará uso de *getCombatLevel* y en *CultistPlayer* de *getCombatLevelAgainstCultistPlayer*, ambos métodos de la clase *Monster*.
- El método *developCombat* de la clase *Napakalaki* debe modificarse de tal forma que si *combat* de *Player* devuelve *LOSEANDCONVERT* realice las siguientes operaciones:

    - Pida a la única instancia de *CardDealer* una carta de *Cultist* (*nextCultist*).
    - Llame al constructor de *CultistPlayer* pasándole el jugador actual y la carta *Cultist* anterior.
    - Reemplace al jugador actual por el nuevo *CultistPlayer* tanto en la lista de jugadores (*players*) como en la variable *currentPlayer* y en el atributo *enemy* de los jugadores que corresponda.
- Crea un consultor protected para la variable *enemy* en la clase *Player* para que pueda ser consultada también por *CultistPlayer*.
- El método *giveMeATresure* se redefine en *CultistPlayer* de manera que se devuelva un tesoro visible elegido al azar.
- El método *canYouGiveMeATreasure* también se redefine para consultar si el enemigo tiene tesoros visibles para ser robados.

4. Modificación de la interfaz de usuario

- En la estructura condicional múltiple que procesa el resultado del combate en GameTester, añade la opción (“case”) siguiente:

```
   case LOSEANDCONVERT :
    System.out.println ("\n\n Has perdido el combate, y te
has convertido en sectario");
    System.out.println ("\n No obstante, tienes que cumplir
el mal rollo");
    currentPlayer = game.getCurrentPlayer();
    break;
```

C) Tareas a realizar en Ruby

- Sigue los mismos pasos indicados en el apartado B, adaptando la nomenclatura de los atributos y métodos a Ruby La adaptación de la interfaz de usuario la puedes tomar del siguiente texto:

```
when NapakalakiGame::CombatResult::LOSEANDCONVERT then
 puts "\n\n Has perdido el combate, y te has convertido
        en                  sectario"
 puts "\n No obstante, tienes que cumplir el mal rollo"
 currentPlayer=@game.getCurrentPlayer()
```

SEGUNDA SESIÓN:

A) Descripción del problema: Rediseño e implementación del juego usando mecanismos de herencia

El diseño realizado hasta ahora no ha contemplado el potencial del mecanismo de herencia. El objetivo de esta sesión es rediseñar el concepto de BadConsequence en base a los tres tipos de mal rollo que tenemos. En clase de teoría haremos un estudio de diferentes alternativas de diseño para modelarlo. Una vez encontrada la mejor estructura de herencia, será ésta la que se terminará de diseñar e implementará en prácticas.

B) Tareas a realizar

- Completa el diagrama de clases de la figura 2, redistribuyendo métodos y atributos en las clases.
- Implementa en Java y Ruby el diagrama final obtenido. Ten en cuenta que en Ruby no existen las clases ni métodos abstractos.

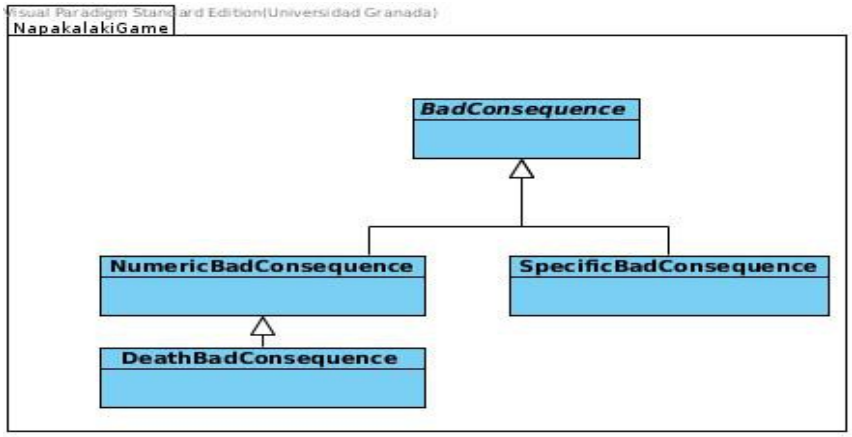

Figura 2: Propuesta de rediseño relacionada con el mal rollo de los monstruos