# Quinta Práctica
# Implementación de la modificación de un diagrama de clases introduciendo herencia e interfaces.

**Competencias específicas de la quinta práctica**

- Interpretar correctamente los diagramas de clases del diseño en UML en los que haya estos dos distintos mecanismos de reutilización de código: herencia de clases e interfaces.
- Modificar el código de un programa orientado a objetos en Java y otro en Ruby de acuerdo con las modificaciones realizadas en un diagrama de clases en el que se añaden clases con relación de herencia, interfaces y clases que las implementan.
- Modificar la Interfaz Gráfica de Usuario (IGU) de Java de forma libre para profundizar en el conocimiento de las librerías AWT y SWING

## Programación y objetivos de la cuarta práctica

## **Tiempo requerido:** 2 sesiones (4 horas)

Planificación y objetivos:
Comienzo: 12 de enero de 2014

| Sesión | Semanas | Objetivos |
|---|---|---|
| Primera: Modelo en Java y Ruby | 12-16 de enero | • Interpretar e implementar en Java y Ruby un diagrama de clases en el que haya relaciones de herencia entre clases.<br>• Interpretar e implementar en Java y Ruby un diagrama de clases en el que haya interfaces y relaciones entre éstas y sus clases. |
| Segunda: IGU Java | 19-23 de enero | • Modificar la interfaz gráfica de usuario para adaptarla a la nueva funcionalidad del modelo<br>• Hacer todos los cambios que se consideren en la IGU para hacerla más amigable y atractiva al usuario<br>• Probar de forma exhaustiva el programa final |

**Entrega (finalizadas las dos sesiones):**

- Fecha: Antes del día 10 de febrero a las 16:00 h.
- Entregables: Un archivo zip denominado P5.zip, con el siguiente contenido:
  - Una carpeta denominada P5Java con el proyecto desarrollado, exporta el proyecto desde Netbeans a un zip, llamado P4Java.
  - Otra carpeta llamada P5Ruby, con todos los ficheros .rb o el proyecto netbeans generado, en este caso hacerlo de la misma forma que se hizo con Java.
- Forma de entrega: Sólo un miembro del grupo sube a SWAD el archivo P5.zip (pestaña "Evaluación", enlace "Mis trabajos", zona de "Actividades"), antes de que termine el plazo indicado por el profesor para cada grupo.

# Quinta práctica

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

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.

- Se introduce 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 y no ha conseguido huir, pero no muere al aplicarse el “mal rollo”, y
  - al tirar de nuevo el dado, obtiene un 6.
- 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.
  - El nivel de combate de un jugador sectario será el nivel de combate de su condición de jugador más el nivel que indica su carta de sectario multiplicado por el número de sectarios en juego.
- Las piezas de oro en los tesoros de un jugador sectario valdrán el doble cuando las canjee por niveles.
- 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.
- Un jugador sectario nunca podrá dejar de serlo.

## B) Tareas a realizar en Java sobre el modelo (primera sesión)

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

   - Herencia: Se ha creado una subclase de Player llamada CultistPlayer.

- Interfaces: Se ha añadido la interfaz Card, que es implementada por las clases Treasure, Monster y Cultist (clase nueva que representa la carta de Sectario).
- 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 modificar el constructor.
- Se ha añadido un constructor de copia a la clase Player para generar el CultistPlayer a partir del jugador.
- El constructor de la clase CultistPlayer llamará al nuevo constructor de copia de Player. Asimismo, incrementará en uno la variable de clase totalCultistPlayers.
- Se han añadido los métodos getOponentLevel y shouldConvert en Player y son sobreescritos en CultistPlayer.
- Se ha añadido una asociación de CultistPlayer a Cultist, cuyo rol es myCultistCard.
- Se ha añadido una asociación de CardDealer a Cultist, con rol unusedCultist. Se han añadido además los métodos nextCultist(), de visibilidad pública, y shuffleCultists() e initCultistCardDeck(), privado.

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.
- Reimplementa el método initMonsterCardDeck para que se añadan al mazo de monstruos las 6 cartas de monstruos nuevas del documento cartasMonstruosCONSectarios.pdf.

3. Implementación de métodos.

   Implementa todos los métodos que has añadido y modifica otros existentes según las siguientes indicaciones:

- Implementa el método shuffleCutists() de la clase CardDealer y modifica el método initCards() de la misma clase para que incluya una llamada a initCultistCardDeck().

- Implementa los métodos getBasicValue() y getSpecialValue() definidos en la interfaz Card de la siguiente manera:
  - Monster: getBasicValue() llama al método getCombatLevel() y getSpecialValue() devuelve el getLevel() + levelChangeAgainstCultistPlayer.
  - Treasure: devuelven el bonus mínimo para getBasicValue() y bonus máximo para getSpecialValue(), llamando a los correspondiente consultores.
  - Cultist: getBasicValue() devuelve gainedLevels y getSpecialValue() devuelve getBasicValue()*CultistPlayer.getTotalCultistPlayers().
- 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 lo que devuelve el método getSpecialValue() de su carta de sectario (Cultist).

- 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 en caso de que no haya podido huir pero siga vivo:
    - Se envíe a sí mismo el mensaje shouldConvert()
    - Si el mensaje devuelve true devuelva el valor LoseAndConvert

- 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.
    - Llame al constructor de CultistPlayer pasándole en el jugador actual y la carta Cultist robada.
    - Reemplace tanto en la lista de jugadores (players) como en la variable currentPlayer al jugador actual por el nuevo CultistPlayer.
- En el método combat() de Player se modificará para que, en vez de preguntar el nivel del monstruo al que se enfrenta al propio monstruo con el método getLevel(), se preguntará a sí mismo con el método getOponentLevel(). El método getOponentLevel() debe definirse en Player y redefinirse en CultistPlayer para hacer uso de getBasicValue() y getSpecialValue() de la clase Monster devolviendo respectivamente el nivel del monstruo al que se enfrenta un jugador y un jugador sectario.
- El método computeGoldCoinsValue() se redefine en CultistPlayer de manera que las piezas de oro dupliquen su valor a la hora de canjearlas por niveles.

## C) Tareas a realizar en Ruby (primera sesión)

- Sigue los pasos indicados en el apartado B que tengan una traducción directa en Ruby. En concreto, todo tiene equivalente en Ruby excepto las interfaces.
- Implementa la interfaz Card simulándola mediante un módulo con el mismo nombre en el que los métodos de la interfaz sí estén implementados, pero sólo con el siguiente código:

```
raise NotImplementedError.new
```

## D) Tareas de programación a realizar en Java sobre la GUI y pruebas del programa (segunda sesión)

### 1. Diseño e implementación de la vista asociada a la clase Cultist: CultistView.

Proceder de manera similar a como se ha hecho con la vista asociada a Treasure, realizando el diseño gráfico de la vista asociada a la clase Cultist e implementando el método **setCultist**.

## 2. Modificación de la vista asociada a la clase Player

a) Añade a la vista del jugador la vista de su carta de sectario.

**Importante**: Para hacerlo basta con arrastrar desde la lista de ficheros del proyecto (zona izquierda del IDE) el archivo CultistView.java hacia la zona donde se está diseñando la vista del jugador; como si fuese un componente más. De igual modo, darle un nombre significativo a dicho componente.

b) Modifica el método **setPlayer** para incluir la orden para actualizar la vista asociada a su carta de sectario (si es el caso) mediante el método setCultist.

## 2. Mejora de la interfaz

De forma libre se pueden hacer todos los cambios que se deseen en la IGU para hacerla más amigable y atractiva al usuario.

## 3. Prueba de la aplicación

Realizad las pruebas del programa necesarias jugando varias veces hasta estar seguros de que el programa funciona correctamente según las especificaciones (reglas del juego) en todas las posibles casuísticas que se os puedan ocurrir.