Práctica 4:

Ampliación del sistema incluyendo herencia y polimorfismo

Modificando el diseño de nuestro juego

En esta práctica se modificará el diseño del juego para añadir una serie de elementos, tales como la herencia entre clases, que te permitirá practicar sobre una base de conceptos de programación orientada a objetos mayor. Observarás además cómo el diseño del juego mejora en algunos aspectos con las modificaciones que se introducen en esta práctica.

Se proporciona un diagrama de clases del proyecto que solo incluye las clases involucradas (las nuevas y las que sufren modificaciones) en esta práctica. Los elementos no recogidos en dicho diagrama continúan sin cambios respecto a las prácticas anteriores.

La parte relativa a las barajas de cartas se implementará únicamente en Java. Este último apartado no se realizará en Ruby.

Dado

La clase *Dice* solo es modificada en cuanto a que se añade un nuevo método de clase. Para evitar confusiones, en el diagrama de clases suministrado solo se ha incluido ese nuevo método. El resto de la clase permanecerá inalterada respecto a lo que ya tienes implementado.

Este método devolverá la dirección de movimiento preferente (primer parámetro) con una probabilidad proporcional al valor de inteligencia suministrado (tercer parámetro). En el caso de no generar como resultado la dirección indicada por el primer parámetro, se elegirá una al azar de las válidas (contenidas en el segundo parámetro).

Armas y escudos

Probablemente hayas observado que las clases *Weapon* y *Shield* tienen muchos aspectos en común. En ambos casos producen un efecto dependiendo de un valor entero y ese efecto deja de producirse cuando se agotan los usos disponibles.

Crea una clase *CombatElement* que generalice ambas clases y contenga todos los elementos comunes. Generaliza también el ataque y la protección, ya que aunque se desea que esos métodos sigan existiendo en *Weapon* y *Shield* respectivamente, es posible definirlos delegando en un método genérico llamado *produceEffect* de *CombatElement.*

Observa que la clase *CombatElement* se ha especificado como abstracta en el diagrama de clases.

Jugadores y Monstruos

Estas dos clases también tienen muchos aspectos en común y es posible generalizarlos creando una clase denominada *LabyrinthCharacter*. Esta contendrá y gestionará todos los atributos de instancia que tenían los monstruos y muchos de los de los jugadores.

Observa el diagrama de clases suministrado y entiende como se ha producido esta generalización. Verás que *LabyrinthCharacter* dispone de algunos consultores y modificadores para que las clases

derivadas puedan tener acceso a algunos de los atributos de esta clase que necesitan.

Observa que en esta generalización, la clase *LabyrinthCharacter* también se ha especificado como abstracta. Ten en cuenta que algunos métodos también se han especificado como abstractos.

Nuevo tipo de jugador

En esta práctica se añade una nueva regla al juego. Aparecen los *FuzzyPlayer*, que no son más que jugadores que se mueven, atacan y defienden utilizando el azar y no de forma tan determinista como los jugadores existentes hasta ahora. Un *FuzzyPlayer* solo puede crearse a partir de otro jugador y dicha creación se produce como resultado de la resurrección del mismo.

Es importante que reflexiones sobre el hecho de que al resucitar una instancia de *Player* deberá ser sustituida por una de *FuzzyPlayer* . En el código no se produce la transformación de un objeto en otro sino que se crea uno nuevo y este deberá sustituir al antiguo en todos los lugares donde sea referenciado. Comprueba los lugares donde guardas referencias a jugadores y añade la funcionalidad necesaria para realizar las actualizaciones necesarias relacionadas con la resurrección y conversión de jugadores. Para ello puedes añadir métodos (no aparecen en el diagrama de clases) que implementen esta funcionalidad. Consulta con tu profesor si tienes dudas relacionas con este aspecto.

El constructor de *FuzzyPlayer* se basa en el uso de los constructores copia añadidos a las clases *LabyrinthCharacter* y *Player*. En Ruby, dado que no es posible utilizar el mecanismo de sobrecarga, en vez utilizar esos constructores copia, añade a *LabyrinthCharacter* y *Player* métodos de instancia que permitan copiar todas las características de otro objeto pasado como parámetro al objeto que ejecuta ese método; haz que el constructor de *FuzzyPlayer* delegue en esos métodos.

La energía de ataque de un *FuzzyPlayer* se calcula sumando a la aportación de las armas el resultado de usar el método *intensity* del dado sobre su fuerza. La intensidad defensiva también recurre al resultado de aplicar el mismo método *intensity* sobre la inteligencia para sumar el resultado a la aportación de los escudos.

El movimiento de este nuevo tipo de jugadores se produce de la siguiente forma:

- Se calcula la dirección de movimiento igual que en la clase *Player*.
- Se pasa esa dirección como primer parámetro al nuevo método de la clase *Dice* añadido en esta práctica y se delega en ese método para generar el resultado.

La representación en forma de cadena de caracteres de estos objetos será idéntica a la de los jugadores de los que ya dispones pero anteponiendo la cadena “Fuzzy” al resto de información.

En el resto de aspectos se comportan exactamente de la misma forma que los jugadores ya existentes en tu juego.

No es necesario intentar evitar conversiones de *FuzzyPlayer* a *FuzzyPlayer*. No supone un problema que se produzcan si un *FuzzyPlayer* muere y es resucitado.

Barajas de cartas (solo Java)

En este apartado se introduce otro cambio en el juego. El jugador, cuando reciba armas y escudos como premio, en vez de generarlos él mismo los obtendrá de sendas barajas.

Como ambas barajas tienen el mismo diseño e implementación, exceptuando el tipo de elemento que contendrán, se recurrirá a una clase paramétrica *CardDeck*. Esta tendrá toda la funcionalidad de una baraja y solo dejará sin implementar el método que crea las armas/escudos y los introduce en la baraja.

Las barajas de cartas funcionarán de la siguiente forma:

- El método *addCard* (en singular) añade **una** carta del tipo correspondiente al contenedor disponible para tal fin.
- El método *addCards* (en plural) añade **todas** las cartas del tipo correspondiente que contendrá la baraja al contenedor disponible para tal fin. En un caso se añadirán armas y en el otro se añadirán escudos. Así, *addCards* utilizará internamente el método anterior *addCard* para añadir cada una de las cartas.
  El hecho de que las barajas ya definan a priori en el método *addCards* las cartas que contendrán (aunque no el orden) sin posibilidad de añadir/quitar cartas es una decisión de diseño intencionada para reducir la complejidad de este apartado.
- El método *nextCard* proporciona la siguiente carta que corresponda. Esta funcionalidad la realiza de la siguiente forma:
  - Si no hay cartas en la colección, se llama al método *addCards* (para que la baraja tenga contenido) *y* se baraja.
  - Independientemente del paso anterior, se selecciona la primera carta, se elimina la misma de la colección de cartas y se devuelva la carta seleccionada.

Para la operación de barajado, consulta el método *Collections.shuffle*.

Comprueba si puedes añadir al parámetro *T* en la clase *CardDeck* la siguiente restricción:

<T extends CombatElement>

Asegúrate de entender la razón.

Comprobando lo aprendido hasta ahora

Una vez finalizada esta práctica y las anteriores deberías saber y entender los siguientes conceptos:

- Los indicados en el guion anterior.
- Herencia.
- Visibilidad.
- Implementar una clase que “deriva” de otra.
- Saber qué métodos y atributos se pueden usar en la clase “derivada” de entre los que tiene la “superclase” y cómo usarlos.
- Polimorfismo y Ligadura dinámica.
- Saber redefinir un método heredado y cuándo es necesaria esta técnica.
- Tipo estático vs. Tipo dinámico.
- Dada una variable que referencia a un objeto concreto, saber qué mensajes se le puede enviar, y qué métodos se ejecutarían en cada caso.
- Uso de clases paramétricas en Java.