Memoria Práctica 4 de Análisis y Diseño de Software

Guillermo Julián Moreno - Víctor de Juan Sanz

Ejercicio 1

En este ejercicio, la implementación del método curarHeridas, después de valorar diversas opciones acabamos optando por incluir un campo en la clase criatura que fuese prob_curar, que en elfos y orcos sería disntinto de 0 y en otros, dicha probabilidad sería 0. Hemos desarrollado esto así porque da una mayor generalidad y portabilidad. A la hora de crear una clase nueva, si queremos que se cure simplemente tenemos que definir en la factoría de ese tipo de criaturas la probabilidad de curarse y todas tienen el mismo método curar heridas. Otra opción a valorar era el uso de una interfaz (PrimerosNacidos por ejemplo) y las criaturas que tuvieran ese don divino de curarse implementarían esa interfaz con un método curarHeridas() (y el consiguiente aplicarHeridas() que llamase a curarHeridas()) y nos encontrábamos con el problema de que esa solución es menos apliable. Por ejemplo, si quisiéramos añadir un nuevo tipo de criaturas que se curasen tendríamos que copiar el código de curarHeridas() y de aplicarHeridas(). Además, si decidiéramos cambiar como se aplican las heridas y en vez de aplicarlas todas juntas se aplican de 1 en 1, habría que cambiar el código de aplicarHeridas() en todas las clases de criatura, es por ello que decidimos implementar esta funcionalidad con el uso de la variable. Además, para facilitar la posible creación de otras criaturas en un futuro, hemos implementado 2 constructores distintos en función de si ese tipo de criaturas se pueden curar o no (con 3 argumentos o con 4).

Sobre la creación de unidades:

Hemos implementado una clase para generar números aleatorios entre 2 dados, para simplificar un poco el código de las factorías y hacerlo más legible. Antes:

```
public Elfo crearCriatura() {
        Random r = new Random(Calendar.getInstance().getTimeInMillis());
        return new Elfo(r.nextInt(3)+1,r.nextInt(3)+1,2);
}
```

Ahora:

```
public Elfo crearCriatura() {
        return new Elfo(RandomUtils.randBetween(2, 3), RandomUtils.randBetween(2, 3), 2);
}
```

Ejercicio 2

Para la creación de las tropas y de los ejércitos, hemos implementado el constructor de ejército de la siguiente manera:

```
public abstract class Ejercito<C extends Criatura> {
        protected ArrayList<Tropa<C>> tropas;
        Random rnd = new Random(Calendar.getInstance().getTimeInMillis());

        public Ejercito(
                        Map<Class<? extends CriaturaFactoria<? extends C>>, List<Integer>> tropasMap)
                        throws ClassNotFoundException, InstantiationException,
                        IllegalAccessException {
                tropas = new ArrayList<Tropa<C>>();

                for (Class<? extends CriaturaFactoria<? extends C>> c : tropasMap
                                .keySet()) {
                        CriaturaFactoria<? extends C> factoria = c.newInstance();
                        List<Integer> tropasList = tropasMap.get(c);

                        for (Integer numTropas : tropasList) {
                                Tropa<C> tropa = new Tropa<C>(factoria, numTropas);
                                tropas.add(tropa);
                        }
                }
        }
```

Recibe como argumento un Map con claves factorías y una lista de enteros con cuantas tropas tiene que crear de cada tipo. Después con el método newInstance() podemos pasarlo como argumento al creador de tropas.

Tener este constructor un tanto complejo en la clase abstracta Ejército nos permite tener mucho más simplificadas todas las subclases.

```
public EjercitoLibre(Map<Class<? extends CriaturaFactoria<? extends CriaturaLibre>>, List<Integer>>
    tropasMap) throws ClassNotFoundException, InstantiationException, illegalAccessException {
        super(tropasMap);
}
```

De esta forma, al trabajar con tipados conseguimos que no se puedan crear tropas con tipos de criaturas distintas e incluso que no se puedan crear guerreros libres en tropas oscuras. En la clase batalla creamos el coMap (Mapa de criaturas oscuraas) que después pasaremos como argumento al creador del ejército.

```
Map<Class<? extends CriaturaFactoria<? extends CriaturaOscura>>, List<Integer>> coMap = new HashMap<Class<?
    extends CriaturaFactoria<? extends CriaturaOscura>>, List<Integer>>();
```

De esta forma nos aseguramos que el map de criaturas oscuras solo le podemos pasar algo que herece de CriaturaFactoría que cree criaturas que hereden de CriaturaOscura. Si le pasamos una factoría que criaturas libres, dará un error de compilación. De esta forma evitamos el uso de excepciones y nos aseguramos una mejor funcionalidad (ya que en ejecución no se pdorucirán errores de este tipo que hubiese que depurar).

Como anécdota de la mejora de esta implementación: al crear la clase orco, Víctor copiaro y pegó código de la calse elfo olvidándose de moficiar el extends por lo que estábamos tratando orco como una criatura libre y al hacer el main e intentar crear un ejército de orcos dentro del ejército oscuro nos encontramos con que no compilaba y el log nos decía donde estaba el error por lo que no tardamos nada en darnos cuenta ni arreglarlo.

Ejercicio 3

A la hora de la batalla no nos hemos complicado y hemos hecho batallas simples en las que cada tropa de un ejército ataca a una tropa aleatorai del otro. La incorporación de la funcionalidad curar heridas está ya implementada dentro del método aplicarHeridas()

Ejercicio 4

No hemos tenido que modificar nada, ya que simplemente con crear las factorías correspondientes ElfoNoldorFactoría y OrcoUrukHaiFacotria que devuelven un elfo y un orco respectivamente. En la clase batalla (donde se encuantran la creación de los ejércitos) podemos escribir la sentencia :

```
coMap.put(OrcoUrukHaiFactoria.class, Arrays.asList(30));
```

que creará 30 guerreros Uruk Hai (que serán instancias de la clase orco.