Auxiliar Revolucionaria 1

Patrones de diseño 2: Ejercicios resueltos y propuestos

Profesor: Alexandre Bergel
Auxiliares: Juan-Pablo Silva
Ignacio Slater
Semestre: Primavera 2019

1. Definiciones necesarias

Sea el juego de cartas *Yu-Gi-Oh!* definido como la tupla $Y = \ldots$
Sería interesante (?) dar una definición matemática para el juego, pero no lo voy a hacer.

Yu-Gi-Oh! es un *TCG* con varios tipos de cartas, pero para los siguientes ejercicios nos centraremos en 2 tipos en particular, *cartas de monstruos* y *cartas de hechizos*. La definición completa de las reglas del juego se pueden encontrar en https://img.yugioh-card.com/uk/rulebook/Rulebook_v9_en.pdf

2. Template method

* Ejercicio 1 Zonas de juego

Si bien existen muchas diferencias entre estos dos tipos de cartas, por ahora consideraremos que solo se diferencian en la zona del campo en la que son jugadas.

Defina las cartas de monstruos y de hechizos para que puedan jugarse en su zona correspondiente siguiendo buenas metodologías de diseño.

* Ejercicio 2 Niveles de monstruos

Además de los tipos de cartas del ejercicio 1 se tienen distintos niveles de monstruos que requerirán sacrificios para ser jugados. Los niveles son:

- **Niveles 1 a 4:** Se pueden jugar sin necesidad de hacer sacrificios.
- **Niveles 5 y 6:** Necesitan de un sacrificio para ser jugados.
- **Niveles 7 y superiores:** Necesitan de 2 sacrificios para jugarse.

Modifique el código para permitir esta mecánica.

** Ejercicio 3 Invocaciones de monstruos (Propuesto)

Extienda el código anterior para permitir dos tipos de invocaciones de monstruos, en modo de defensa boca abajo y boca arriba en modo de ataque.

3. Null Object

* Ejercicio 4 Efectos

Además de los tipos ya definidos, todas las cartas de hechizos tienen efectos, y las cartas de monstruos pueden o no tener algún efecto. Extienda la implementación para permitir esta mecánica.

*** Ejercicio 5 Efectos específicos (Propuesto)

Actualmente la implementación tiene efectos pero estos no hacen mucho. Considere los siguientes efectos específicos:

- Utilizar el efecto de una carta puede anular el efecto de otra.
- El efecto de una carta puede ocupar espacios de la zona de monstruos o hechizos con *tokens* que no pueden ser movidos o sacrificados y que no pueden realizar ninguna acción.

Extienda el código para implementar estas funcionalidades.

4. Composite

** Ejercicio 6 Cartas de efecto continuo

Considere ahora que existen efectos que afectan a otras cartas y que se mantienen a lo largo del juego, en este caso se definen 2:

- **Cartas de efecto continuo:** Son cartas que tienen un efecto sobre una o más cartas en juego.
- **Cartas de equipo:** Son cartas que pueden ser equipadas por un monstruo y que tienen un efecto sobre este.

Como estas cartas tienen efectos sobre otras y se necesita saber qué cartas son las que las están afectando, se necesita guardar referencias a estas. Esto puede causar que haya cartas con referencias a otras que a su vez tienen referencias a otras. Implemente esta nueva funcionalidad de las cartas.

**** Ejercicio 7 Cartas de campo

Se le solicita ahora que agregue un nuevo tipo de carta al juego.

Las cartas de campo son cartas que se juegan en una zona propia del *game mat* y que tienen efecto sobre todo el resto de las cartas.

Extienda el código para agregar esta mecánica:

¡CUIDADO! Si se implementa de la misma manera que en el ejercicio anterior se puede caer en referencias circulares (por ejemplo si hay 2 cartas de campo en juego).

5. Singleton

* Ejercicio 8 Instancia única del efecto nulo

Como todos los efectos nulos son iguales, no tiene sentido crear múltiples instancias del mismo.

Modifique la implementación para asegurarse de que no se pueda tener más de una instancia de efectos nulos.

** Ejercicio 9 Extender otros efectos

Extienda la implementación del ejercicio 5 para que utilice *singleton pattern*.

6. Flyweight factory

** Ejercicio 10 Eliminar la duplicación de objetos

En el juego puede suceder que varias cartas tengan el mismo efecto, en dicho caso crear varias veces el mismo objeto sería un uso ineficiente de memoria.

Modifique la implementación para que ningún efecto se pueda crear más de una vez.

*** Ejercicio 11 Inicialización *lazy* de efectos (Propuesto)

Existe la posibilidad de que los efectos de algunas cartas nunca sean utilizados, en ese caso también sería un uso ineficiente de memoria crear esos efectos.

Extienda la implementación anterior para que los efectos se creen solamente si se necesitan.

7. Soluciones

Solución del ejercicio 1

Lo primero que necesitamos hacer es crear una interfaz y una clase abstracta que represente todos los tipos de cartas, así:

```java
// com.github.islaterm.yugi.card.ICard
public interface ICard {
  void playToMat(GameMat gameMat);
}

// com.github.islaterm.yugi.card.AbstractCard
public abstract class AbstractCard implements ICard {
  @Override
  public abstract void playToMat(GameMat gameMat);
}
```

Lo que estamos haciendo al definir el método como abstracto es relegar la labor de decidir cómo se jugarán las cartas a las implementaciones particulares de cada tipo de carta. Esto es lo que se conoce como *template method pattern*.

Ahora, las implementaciones particulares de cada método serán:

```java
// com.github.islaterm.yugi.card.monster.BasicMonsterCard
public class MonsterCard extends AbstractCard {

  @Override
  public void playToMat(@NotNull GameMat gameMat) {
    gameMat.addMonster(this);
  }
}

// com.github.islaterm.yugi.card.SpellCard
public class SpellCard extends AbstractCard {
  @Override
  public void playToMat(@NotNull GameMat gameMat) {
    gameMat.addSpell(this);
  }
}
```

Solución del ejercicio 2

Este problema también puede ser resuelto utilizando un *template*. Para esto necesitaremos extender el modelo de cartas de monstruos que llevamos hasta ahora.

```java
// com.github.islaterm.yugi.card.monster.IMonsterCard
public interface IMonsterCard extends ICard {
  void addTribute(IMonsterCard card);
}
```

```java
// com.github.islaterm.yugi.card.monster.AbstractMonsterCard
public abstract class AbstractMonsterCard extends AbstractCard implements IMonsterCard {
  private Set<IMonsterCard> tributes = new HashSet<>();

  @Override
  public void addTribute(IMonsterCard card) {
    this.tributes.add(card);
  }

  @Override
  public void playToMat(GameMat gameMat) {
    if (enoughSacrifices()) {
      gameMat.addMonster(this);
      tributes.forEach(gameMat::removeMonster);
    }
  }

  protected abstract boolean enoughSacrifices();
}
```

Ahora, con esas clases definidas se pueden implementar los tres tipos de cartas de monstruos que necesitamos.

```java
// com.github.islaterm.yugi.card.monster.BasicMonsterCard
public class BasicMonsterCard extends AbstractMonsterCard {
  @Override
  protected boolean enoughSacrifices() {
    return tributes.isEmpty();
  }
}
// com.github.islaterm.yugi.card.monster.OneTributeMonsterCard
public class OneTributeMonsterCard extends AbstractMonsterCard {
  @Override
  protected boolean enoughSacrifices() {
    return tributes.size() == 1;
  }
}
// com.github.islaterm.yugi.card.monster.TwoTributesMonsterCard
public class TwoTributesMonsterCard extends AbstractMonsterCard {
  @Override
  protected boolean enoughSacrifices() {
    return tributes.size() == 2;
  }
}
```

Solución del ejercicio 4

Para resolver este problema debemos crear una clase que represente los efectos de los monstruos y hechizos.

Adicionalmente, necesitamos una forma de definir el caso en que un monstruo no tenga efecto, para esto último utilizaremos *null object pattern*.

Primero, necesitaremos una interfaz para los efectos y permitir que las cartas puedan tener dichos efectos:

```java
// com.github.islaterm.yugi.effect.IEffect
public interface IEffect {
  void use();
}
// com.github.islaterm.yugi.card.ICard
void useEffect();
// com.github.islaterm.yugi.card.AbstractCard
@Override
public void useEffect() {
  cardEffect.use();
}
```

Ahora, podemos implementar los efectos concretos:

```java
// com.github.islaterm.yugi.effect.Effect
public class Effect implements IEffect {
  @Override
  public void use() {
    System.out.println("Soy un efecto owo");
  }
}
// com.github.islaterm.yugi.effect.NullEffect
public final class NullEffect implements IEffect {
  @Override
  public void use() {
    System.out.println("No hago nada :D");
  }
}
```

La clase `NullEffect` se define como `final`, esto significa que la clase no puede ser extendida. Esto ayuda a entender mejor el propósito de la clase.

Con los efectos definidos podemos crear los constructores respectivos de las cartas para definir sus efectos.

```java
// com.github.islaterm.yugi.card.monster.BasicMonsterCard
public BasicMonsterCard(IEffect effect) {
  cardEffect = effect;
}
// Los otros tipos de monstruos son análogos
// com.github.islaterm.yugi.card.SpellCard
public SpellCard(Effect effect) {
  cardEffect = effect;
}
```

Note que el constructor de las cartas de hechizos no puede recibir efectos nulos porque no tendría sentido.

Solución del ejercicio 6

Para resolver este ejercicio utilizaremos *composite pattern*, este patrón se puede utilizar en cualquier estructura que deba recorrerse de forma recursiva y que pueda modelarse como un árbol.

En este caso, la estructura de cartas que tenemos es un árbol, donde cada carta puede referenciar a 0 o más cartas, para aplicar este patrón se necesita crear una interfaz común para los objetos que foman parte de la estructura. En nuestro caso esa interfaz ya la hemos definido en `ICard`, pero habrá que agregarle un método que pueda recorrer el árbol, así:

```
// com.github.islaterm.yugi.card.ICard
void applyEquippedEffects();
// com.github.islaterm.yugi.card.AbstractCard
protected Set<ICard> equippedCards = new HashSet<>();

@Override
public void applyEquippedEffects() {
  for (ICard card : equippedCards) {
    card.useEffect();
    card.applyEquippedEffects();
  }
}
```

De esta forma los efectos se aplicarán recursivamente y terminarán una vez que se llegue a una carta que no tenga ninguna otra carta equipada.

Solución del ejercicio 8

Para asegurarse de que no se pueda crear más de una instancia de un objeto se necesita ocultar el constructor de la clase. Esto implica que la clase sólo puede ser instanciada desde dentro de ella, por lo que es necesario que el método para obtener la instancia de la clase sea estático. Esto es lo que se conoce como *singleton pattern*.

Modifiquemos la clase `NullEffect` para que sea un *singleton*.

```
// com.github.islaterm.yugi.effect.NullEffect
public class NullEffect implements IEffect {
  private static IEffect instance;

  private NullEffect() {
  }

  @Override
  public void use() {
    System.out.println("No hago nada :D");
  }

  public static IEffect getInstance() {
    if (instance == null) {
      instance = new NullEffect();
    }
    return instance;
  }
}
```

Solución del ejercicio 10

Para evitar crear un mismo objeto dos veces ya se vio que se puede utilizar un *singleton*, pero si se tienen muchos objetos que se requiere que sean únicos entonces implementar este patrón agregaría mucha complejidad al código.

En vez de eso, se puede utilizar *factory pattern* para unificar y simplificar la creación de dichos objetos. Ahora, si además de eso el *factory* guarda los objetos que ha creado para no crearlos de nuevo, estaremos utilizando *flyweight factory pattern*.

```java
// com.github.islaterm.yugi.effect.FlyweightEffectFactory
public class FlyweightEffectFactory {
  private Map<String, IEffect> createdEffects = new HashMap<>();

  public IEffect getEffect(String effectId) {
    if(!createdEffects.containsKey(effectId)) {
      createdEffects.put(effectId, new Effect());
    }
    return createdEffects.get(effectId);
  }
}
```