Tecnología de la Programación I

Presentación de la Práctica 2 (parte II)

(Basado en la práctica de Alberto Núñez y Miguel V. Espada)

Ana M. González de Miguel (ISIA, UCM)

Índice

1. Herencia y Polimorfirmo

- ✓ En la práctica 1 hemos replicado código en los objetos del juego y en las listas de objetos.
- ✓ Esto lo vamos a resolver en la práctica 2 usando una de las herramientas de POO: **herencia**. Para ello se creará una jerarquía de clases que herede de *GameObject*.
  - La clase abstracta *GameObject* tendrá los atributos y métodos básicos para controlar la posición en el tablero y una referencia a la clase *Game*. Esta clase se da en el anexo.
  - De *GameObject* heredarán *Ship* y *Weapon*. La primera clase representará las naves del juego, mientras que la segunda representará los objetos que causan daño. Implementa el método *move*() en la clase *Weapon* usando un atributo *Move* (ver al final del anexo).

- La clase *EnemyShip*, que hereda de la clase *Ship*, representará todas las naves enemigas, incluyendo el Ovni, con un atributo *Move*. Entre otros elementos, esta clase deberá gestionar los puntos que se obtienen al ser destruidas por el jugador (*game* recibe los puntos en el método *onDelete*).
- De forma similar, la clase *UCMShip* también hereda de *Ship* y representará la nave que controla el jugador. Incorpora atributos *points*, *hasShockWave* y *canShootLaser*.
- La clase *AlienShip* es probablemente una de las clases más complicadas de la práctica. En particular, esta clase – junto con la clase *Ovni* – hereda de *EnemyShip* y representará las naves del juego que se mueven con un comportamiento grupal, es decir, todas en la misma dirección. Su movimiento seguirá las mismas reglas que fueron definidas en la práctica 1. Para representar este movimiento, se utilizará un atributo estático que represente el número de naves que deben realizar un determinado movimiento. Así, todos los objetos de esta clase compartirán el mismo atributo.

- Considerar al menos los siguientes atributos en *AlienShip*

```
protected static int REMAINING_ALIENS = 0;
private static boolean IS_IN_FINAL_ROW;
private static int SHIPS_ON_BORDER;
protected int cyclesToMove;
```

- En la constructora de *AlienShip:*

```
REMAINING_ALIENS += 1;
SHIPS_ON_BORDER = 0;
```

- En el método *move*():

```
Si (cyclesToMove == 0)
   Actualizar cyclesToMove
   Hacer el movimiento a la izquierda o a la derecha
   Actualizar IS_IN_FINAL_ROW
   Si está en el borde, SHIPS_ON_BORDER = REMAINING_ALIENS
Sino si (SHIPS_ON_BORDER > 0) y !IS_IN_FINAL_ROW
   Aumentar la fila
   Cambiar el sentido del movimiento
   SHIPS_ON_BORDER -= 1
Sino cyclesToMove--
```

- Intuitivamente, las clases *RegularShip* y *DestroyerShip* heredarán de *AlienShip*, implementando su comportamiento. Cambia el atributo *Bomb* de *DestroyerShip* por un boolean *canShootBomb*. Esta clase implementa el método *computerAction*().
- Finalmente, las clases *Bomb* y *UCMShipLaser* heredan de *Weapon*, representando el movimiento de los disparos realizados por las naves *DestroyerAlien* y *UCMShip*, respectivamente. Por un atributo *DestroyerShip* en *Bomb* y un atributo *UCMShip* en *UCMShipLaser*.
- La clase *Shockwave* también heredará de *Weapon*; aunque no se mueva y tenga una forma de atacar diferente también lo podemos considerar como un arma. Si más adelante tuviéramos otros *power-ups* refinaremos la jerarquía de clases para tener armas con diferentes comportamientos.

✓ Es importante remarcar que en esta versión de la práctica, la mayor parte de la lógica estará distribuida en los propios elementos del juego, quedando la clase *Game* liberada de esta tarea.

- ✓ Adicionalmente, podemos utilizar la herencia para refactorizar el código de las listas.
- ✓ En esta práctica, utilizaremos una clase – llamada *GameObjectBoard* (ver el anexo) – que será la encargada de gestionar la lista de elementos de tipo *GameObject*.
- ✓ La clase *Game* solo tendrá un atributo de tipo *GameObjectBoard* y otro de tipo *UCMShip*.
- ✓ Con estos dos atributos se gestionarán todos los elementos del juego, incluidos el Ovni, los disparos del jugador y las bombas lanzadas por las naves enemigas.
- ✓ El hecho de emplear la palabra *board* no quiere decir que ahora vayamos a utilizar como tablero una matriz para gestionar los objetos; seguiremos usando una lista donde estarán todos los objetos de juego.

2. Detalles de Implementación

- ✓ Hay varios aspectos que van a cambiar radicalmente la estructura del código:
  - Sólo tenemos un contenedor para todos los objetos.
  - Desde el *Game* y el *board* sólo manejamos abstracciones de los objetos, por lo que no podemos distinguir quién es quién.
  - Toda la lógica del juego estará en los objetos de juego. Cada clase concreta sabe sus detalles acerca de cómo se mueve, como ataca, como recibe ataque y cuando realiza computer actions.
- ✓ Estas refactorizaciones son complejas y os vamos a dar ayuda; empecemos por la clase *Game*, la cual se encarga de demasiadas tareas en la práctica 1. En esta nueva versión, al igual que hemos hecho con el controlador, vamos a delegar su comportamiento a las demás clases. Veamos un fragmento de código.

```
public class Game implements IPlayerController {
....
GameObjectBoard board;
private UCMShip player;
....
public Game (Level level, Random random){
   this.rand = random;
   this.level = level;
   initializer = new BoardInitializer();
   initGame();
}

public void initGame () {
   currentCycle = 0;
   board = initializer.initialize (this, level);
   player = new UCMShip(this, DIM_X/2, DIM_Y-1);
   board.add(player);
}
....
```

```
public boolean aliensWin() {
   return !player.isAlive () || AlienShip.haveLanded();
}

private boolean playerWin () {
   return AlienShip.allDead();
}

public void update() {
   board.computerAction(); // eliminar la llamada computerAction() de
los métodos execute de las subclases de Command
   board.update();
   currentCycle += 1;
}
...
```

- El nuevo *game* mantiene una referencia al *player* y al *board* donde se almacenan los objetos de juego. Cuando tiene que hacer alguna acción, la delega a la clase correspondiente. Podríamos decir que el *Game* no hace absolutamente nada salvo delegar.
- De hecho el código de la clase *Game* es tan pequeño que os lo vamos a dar en los anexos casi completamente. Igual tu código difiere en alguna parte o en el nombre de algún método o atributo, puedes modificar el nuestro, o el tuyo, como te sea más cómodo; lo importante es que entiendas bien su función.

- ✓ En el *Game* usamos una clase auxiliar para inicializar el juego. El *BoardInitializer* se encarga de añadir los objetos de juego en el juego dependiendo del nivel.
- ✓ Si nos fijamos en la declaración del *Game* vemos que implementa un interfaz *IPlayerController*.
- ✓ Esta interfaz no es 100% necesaria, pero nos ayuda a abstraer qué métodos son los necesarios para tratar la comunicación con el jugador. En realidad es lo que se conoce como *mixin*; es una forma de incluir métodos de una clase en otra, sin que exista relación de herencia entre ellas.

```
public interface IPlayerController {

// Player actions
public boolean move (int numCells);
public boolean shootMissile();
public boolean shockWave();

// Callbacks
public void receivePoints(int points);
public void enableShockWave();
public void enableMissile();

}
```

- ✓ Vemos que hay dos tipos de métodos:
  - Las acciones que puede hacer el jugador. Estas llamadas le llegan a través de los comandos del controlador.
  - Los **callbacks** que son las acciones de retorno. Por ejemplo cuando se habilita el *shockWave*, después de matar al Ovni. Callback es un término muy común para denotar este tipo de funciones, que se llaman al final de unas determinadas acciones.
- ✓ *Game* debe implementar estos métodos de la interfaz, aunque recuerda que no debe hacer nada más que delegar al objeto que le corresponda.
- ✓ Vamos a usar otro interfaz para encapsular los métodos relacionados con los ataques. La clase *GameObject* implementa el interfaz *IAttack*.

- La idea es que todos los objetos del juego deben tener la posibilidad de atacar o ser atacados. Por ello, tenemos que implementar la posibilidad de recibir ataque por cada uno de los proyectiles.

```
public interface IAttack {
    default boolean performAttack(GameObject other) {return false;};
    default boolean receiveMissileAttack(int damage) {return false;};
    default boolean receiveBombAttack(int damage) {return false;};
    default boolean receiveShockWaveAttack(int damage) {return false;};
}
```

- Por defecto, *performAttack* devuelve false lo que quiere decir que el objeto de juego no hace daño (sólo hacen daño el misil, las bombas y el shockwave).

- ✓ En las clases *Weapon* y *Shockwave* debes implementar el método *performAttack.*
- ✓ Implementa el método *receiveMissileAttack* en las clases *EnemyShip y Bomb.*
- ✓ Implementa el método *receiveBombAttack* en las clases *UCMShip y UCMShipLaser.*

- ✓ Los destroyers no hacen daño, lo que hacen es lanzar bombas que son las que realmente hacen daño. Si un objeto de juego puede atacar, tendrá que sobreescribir el método e implementar la lógica de su ataque.
- ✓ De igual manera todos los objetos pueden recibir daño de otro objeto. Por defecto no lo reciben. Pero, por ejemplo, los aliens y las bombas sí reciben daño del misil, o el UCMShip y el misil sí reciben ataques de la bomba.
- ✓ Deberás rellenar esos métodos. Al igual que en el *Game* esos métodos tienen muy poca lógica, toda la funcionalidad se delega a los objetos de juego.

- ✓ Los objetos de juego que hacen acciones aleatorias, como tirar bombas, deberán implementar otro interfaz *IExecuteRandomActions* donde se encapsulan como métodos estáticos los métodos para saber si ejecuta o no la acción. Este código también te lo damos para que lo uses donde sea necesario (ver el anexo).
- ✓ Uno de los objetos que implementa *IExecuteRandomActions* es el Ovni. Recomendamos siempre tener un ovni en el tablero, con atributo de estado para saber si está enable o disable. Así podemos gestionar la aparición de un nuevo ovni desde el mismo objeto ya que consistirá simplemente en habilitarlo, en lugar de añadir un nuevo objeto.

3. Extensión del Juego

- ✓ En esta versión de la práctica vamos a hacer una pequeña ampliación:
  - **Nave explosiva**. Las naves de tipo común pueden transformarse en una nave explosiva durante la partida. Si eso ocurre – esta nave – tras ser destruida, quita un punto de vida a todos los objetos que hay en una casilla adyacente (horizontal, vertical y diagonalmente). La probabilidad de que una nave común se transforme en nave explosiva es del 5% y se calculará en la acción computer action. El movimiento, los puntos de daño y los puntos obtenidos al ser destruida, serán los mismos que los de la nave común. Sin embargo, se representará en pantalla con la letra E.

- **SuperMisil**. El jugador puede "comprar" un supermisil por 20 puntos. Para ello se deberá incluir el comando adicional comprar supermisil, el cual recibe un único argumento cuyo valor tiene que ser supermisil. En la interfaz de usuario se deberá mostrar el número de supermisiles que éste posee durante la partida, comenzando con cero supermisiles al inicio del juego. Además, se deberá modificar el comando shoot para que acepte un parámetro nuevo, en este caso supermisil, de forma que la nave pueda lanzar uno de los supermisiles comprados. En el caso de que el jugador ejecute este comando y no disponga de supermisiles, se mostrará por pantalla el correspondiente mensaje de error, de igual forma que si al comprar el supermisil, no se dispone de los puntos necesarios. El comportamiento del disparo normal será el mismo que el de la práctica 1. El supermisil actúa de forma similar al misil, salvo que, en este caso, causa dos puntos de daño a las naves enemigas. Una vez impacte con un proyectil enemigo o una nave, aunque ésta tenga un único punto de vida, el supermisil desaparece.

4. Anexos de la Práctica 2

```
package pr2.game;
import java.util.Random;
import pr2.game.Level;
import pr2.game.GameObjects.AlienShip;
import pr2.game.GameObjects.BoardInitializer;
import pr2.game.GameObjects.GameObject;
import pr2.game.GameObjects.IPlayerController;
import pr2.game.GameObjects.UCMShip;
import pr2.game.GameObjects.Lists.GameObjectBoard;

public class Game implements IPlayerController {

public final static int DIM_X = 9;
public final static int DIM_Y = 8;
private int currentCycle;
private Random rand;
private Level level ;
```

```
GameObjectBoard board;
private UCMShip player;
private boolean doExit;
private BoardInitializer initializer;

public Game (Level level, Random random){
   this.rand = random;
   this.level = level;
   initializer = new BoardInitializer();
   initGame();
}

public void initGame () {
   currentCycle = 0;
   board = initializer.initialize(this, level);
   player = new UCMShip(this, DIM_X / 2, DIM_Y - 1);
   board.add(player);
}
```

```
public Random getRandom() {
   return rand;
}
public Level getLevel() {
   return level;
}
public void reset() {
   initGame();
}
public void addObject(GameObject object) {
   board.add(object);
}
public String positionToString(int x, int y) {
   return board.toString(x, y);
}
public boolean isFinished() {
   return playerWin() || aliensWin() || doExit;
}
```

```
public boolean aliensWin() {
   return !player.isAlive () || AlienShip.haveLanded();
}
private boolean playerWin () {
   return AlienShip.allDead();
}
public void update() {
   board.computerAction();
   board.update();
   currentCycle += 1;
}
public boolean isOnBoard(int x, int y) {
   return x >= 0 && y >= 0 && x < DIM_X && y < DIM_Y;
}
public void exit() {
   doExit = true;
}
```

```
public String infoToString() {
   return "Cycles: " + currentCycle + "\n" +
   player.stateToString() +
   "Remaining aliens: " + (AlienShip.getRemainingAliens()) + "\n";
}
public String getWinnerMessage () {
   if (playerWin()) return "Player win!";
   else if (aliensWin()) return "Aliens win!";
   else if (doExit) return "Player exits the game";
   else "This should not happen"
}
// TODO implementar los métodos del interfaz IPlayerController

}
```

```
package pr2.game.GameObjects;

import pr2.game.Game;
import pr2.game.Level;
import pr2.game.GameObjects.Lists.GameObjectBoard;

public class BoardInitializer {
   private Level level ;
   private GameObjectBoard board;
   private Game game;

public GameObjectBoard initialize(Game game, Level level) {
   this.level = level;
   this.game = game;
   board = new GameObjectBoard(Game.DIM_X, Game.DIM_Y);
   initializeOvni () ;
   initializeRegularAliens () ;
   initializeDestroyerAliens () ;
   return board;
}
```

```
private void initializeOvni () {
   // TODO implement
}
private void initializeRegularAliens () {
   // TODO implement
}
private void initializeDestroyerAliens () {
   // TODO implement
}
}
```

```
package pr2.game.GameObjects;
import pr2.game.Game;

public abstract class GameObject implements IAttack {
   protected int x;
   protected int y;
   protected int live;
   protected Game game;

   public GameObject( Game game, int x, int y, int live) {
      this.x = x;
      this.y = y;
      this.game = game;
      this.live = live;
   }
   public int getX() {
      return x;
   }
   public int getY() {
      return y;
   }
```

```
public boolean isAlive() {
   return this.live > 0;
}
public int getLive() {
   return this.live;
}
public boolean isOnPosition(int x, int y) {
   return this.x == x && this.y == y;
}
public void getDamage (int damage) {
   this.live = damage >= this.live ? 0 : this.live - damage;
}
public boolean isOut() {
   return !game.isOnBoard(x, y);
}
public abstract void computerAction();
public abstract void onDelete();
public abstract void move();
public abstract String toString();
}
```

```
package pr2.game.GameObjects;
import java.util.Random;
import pr2.game.Game;
import pr2.game.Level;

public interface IExecuteRandomActions {
   static boolean canGenerateRandomOvni(Game game){
      return game.getRandom().nextDouble() <
game.getLevel().getOvniFrequency();
   }

   static boolean canGenerateRandomBomb(Game game){
      return game.getRandom().nextDouble() <
game.getLevel().getShootFrequency();
   }
}
```

```
package pr2.game.GameObjects.Lists;
import pr2.game.GameObjects.GameObject;

public class GameObjectBoard {
   private GameObject[] objects;
   private int currentObjects;

   public GameObjectBoard (int width, int height) {
      // TODO implement
   }
   private int getCurrentObjects () {
      // TODO implement
   }
   public void add(GameObject object) {
      // TODO implement
   }
   private GameObject getObjectInPosition (int x, int y) {
      // TODO implement
   }
```

```
private int getIndex(int x, int y) {
   // TODO implement
}
private void remove(GameObject object) {
   // TODO implement
}
public void update() {
   // TODO implement
}
private void checkAttacks(GameObject object) {
   // TODO implement
}
public void computerAction() {
   // TODO implement
}
private void removeDead() {
   // TODO implement
}
public String toString(int x, int y) {
   // TODO implement
}
```

```
package pr2.game;

public enum Level {
    EASY(4, 2, 0.2, 3, 0.5, 1),
    HARD(8, 4, 0.3, 2, 0.2, 2),
    INSANE(12, 4, 0.5, 1, 0.1, 3);

    private int numRegularAliens;
    private int numDestroyerAliens;
    private int numCyclesToMoveOneCell;
    private double ovniFrequency;
    private double shootFrequency;
    private int numRowsOfRegularAliens;
```

```
private Level(int numRegularAliens,
              int numDestroyerAliens,
              double shootFrequency,
              int numCyclesToMoveOneCell,
              double ovniFrequency,
              int numRowsOfRegularAliens)
{
   this.numRegularAliens = numRegularAliens;
   this.numDestroyerAliens = numDestroyerAliens;
   this.shootFrequency = shootFrequency;
   this.numCyclesToMoveOneCell = numCyclesToMoveOneCell;
   this.ovniFrequency = ovniFrequency;
   this.numRowsOfRegularAliens = numRowsOfRegularAliens;
}
```

```
public int getNumRegularAliens() {
   return numRegularAliens;
}
public int getNumDestroyerAliens() {
   return numDestroyerAliens;
}
public Double getShootFrequency() {
   return shootFrequency;
}
public int getNumCyclesToMoveOneCell() {
   return numCyclesToMoveOneCell;
}
public Double getOvniFrequency() {
   return ovniFrequency;
}
public int getNumRowsOfRegularAliens() {
   return numRowsOfRegularAliens;
}
```

```
public int getNumRegularAliensPerRow() {
   return numRegularAliens / numRowsOfRegularAliens;
}
public int getNumDestroyerAliensPerRow() {
   return getNumDestroyerAliens();
}
public static Level fromParam(String param) {
   for (Level level : Level.values())
      if (level.name().equalsIgnoreCase(param)) return level;
   return EASY;
}
public Double getTurnExplodeFreq(){
   return 0.05;
}
}
```

```
package pr2.game.GameObjects;

public enum Move {
   LEFT, RIGHT, UP, DOWN, NONE;

   public Move flip() {
      if(this == LEFT) return RIGHT;
      if(this == RIGHT) return LEFT;
      return this;
   }
}
```