Práctica 2: Space Invaders refactored, parte I y II

Fecha de entrega: 5 de Diciembre de 2019, 9:00

Objetivo: Herencia, polimorfismo, clases abstractas, interfaces, excepciones y gestión de ficheros

1. Introducción

El objetivo de esta práctica consiste, fundamentalmente, en aplicar los mecanismos que ofrece la POO para mejorar el código desarrollado hasta ahora. En particular, en esta versión de la práctica incluiremos las siguientes mejoras:

- Primero, como se explica en la sección 2, refactorizamos[1] el código de la práctica anterior. Para ello, modificaremos parte del controlador distribuyendo su funcionalidad entre un conjunto de clases.
- Vamos a hacer uso de la herencia para reorganizar los objetos de juego. Hemos visto que hay mucho código repetido en los distintos tipos de aliens o entre el misil y las bombas. Por ello, vamos a crear una estructura de clases que nos permita extender fácilmente la funcionalidad del juego.
- Una vez refactorizada la práctica, vamos a añadir nuevos objetos al juego.
- La herencia también nos va a permitir redefinir cómo almacenamos la información del estado del juego. En la práctica anterior, al no usar herencia, debíamos tener una lista para cada conjunto de objetos. Sin embargo, en esta versión de la práctica, podremos usar una sola estructura de datos para todos los objetos de juego.
- Vamos a incluir la posibilidad de modificar cómo se representa el tablero. Para ello, crearemos otro *printer* con el que podamos ver el juego de forma *serializada*.

[1] Refactorizar consiste en cambiar la estructura del código (se supone que para mejorarlo) sin cambiar su funcionalidad.

- Utilizaremos la serialización poder grabar y cargar partidas en disco. Para ello, ampliaremos el juego para gestionar ficheros de entrada y salida.

- También vamos a incluir el manejo y tratamiento de excepciones. En ocasiones, existen estados en la ejecución del programa que deben ser tratados convenientemente. Además, cada estado debe proporcionar al usuario información relevante como, por ejemplo, errores producidos al procesar un determinado comando. En este caso, el objetivo es dotar al programa de mayor robustez, así como mejorar la interoperabilidad con el usuario.

Todos los cambios comentados anteriormente se llevarán a cabo de forma progresiva. El objetivo principal es extender la práctica de una manera robusta, preservando la funcionalidad en cada paso que hagamos y modificando el mínimo código para ampliarla.

Importante: Hemos divido el enunciado en varias partes que iremos publicando progresivamente, en esta primera parte nos centraremos en el primer item de la lista. Solo habrá una entrega para toda la práctica completa.

2. Refactorización de la solución de la práctica anterior

Hay una regla no escrita en programación que dice *Fat models and skinny controllers*. Lo que viene a decir es que el código de los controladores debe ser mínimo y, para ello, deberemos llevar la mayor parte de la funcionalidad a los modelos. Una manera de adelgazar el controlador es utilizando el patrón *Command* que, como veremos, permite encapsular acciones de manera uniforme y extender el sistema con nuevas acciones sin modificar el controlador.

El cuerpo del método `run` del controlador va a tener - más o menos - este aspecto. Tu código no tiene que ser exactamente igual, pero lo importante es que veas que se asemeja a esta propuesta.

```
while (!game.isFinished()){
  System.out.println(PROMPT);
  String[]  words = in.nextLine().toLowerCase().trim().split ("\\s+");

        Command command = CommandGenerator.parse(words);
        if (command != null) {
          if (command.execute(game))
              System.out.println(game);
        }
        else {
          System.out.format(unknownCommandMsg);
        }
}
```

Básicamente, mientras el juego no termina, leemos un comando de la consola, lo parseamos, lo ejecutamos y, si la ejecución es satisfactoria y ha cambiado el estado del juego, lo repintamos. Este mismo controlador nos valdría para diferentes versiones del juego o incluso para diferentes juegos. En la próxima sección vamos a ver cómo funciona el patrón *Command*.

2.1. Patrón Command

El patrón *command* es un patrón de diseño[2] muy conocido. En esta práctica no necesitas conocer de este patrón más de lo que se explica aquí. Para aplicarlo, cada comando del juego se representa por una clase diferente, que llamamos MoveCommand, ShootCommand, UpdateCommmand, ResetCommand, HelpCommand,... y que heredan de una clase abstracta Command. Las clases concretas invocan métodos de la clase Game para ejecutar los comandos respectivos.

En la práctica anterior, para saber qué comando se ejecutaba, el método run del controlador contenía un switch - o una serie de if's anidados - cuyas opciones correspondían a los diferentes comandos. Con la aplicación del patrón *command*, para saber qué comando ejecutar, el método run del controlador divide en palabras el texto proporcionado por el usuario (*input*) a través de la consola, para a continuación invocar un método de la *clase utilidad*[3] CommandGenerator, al que se le pasa el *input* como parámetro. Este método le pasa, a su vez, el *input* a un objeto *comando* de cada una de las clases concretas (que, como decimos, son subclases de Command) para averiguar cuál de ellos lo acepta como correcto. De esta forma, cada subclase de Command busca en el *input* el texto del comando que la subclase representa.

Aquel objeto *comando* que acepte el input como correcto devuelve al CommandGenerator otro objeto *comando* de la misma clase que él. El parseador pasará, a su vez, el objeto *comando* recibido al controlador. Los objetos *comando* para los cuales el input no es correcto devuelven el valor null. Si ninguna de las subclases concretas de comando acepta el input como correcto, es decir, si todas ellas devuelven null, el controlador informa al usuario de que el texto introducido no corresponde a ningún comando conocido. De esta forma, si el texto proporcionado por el usuario corresponde a un comando del sistema, el controlador obtiene del parseador CommandGenerator un objeto de la subclase que representa a ese comando y que puede, a su vez, ejecutar el comando.

Implementación

El código de la clase abstracta Command es el siguiente:

```
public abstract class Command {

  protected final String name;
  protected final String shortcut;
  private final String details ;
  private final String help;


  protected static final String incorrectNumArgsMsg = "Incorrect number of arguments";
  protected static final String incorrectArgsMsg = "Incorrect argument format";

  public Command(String name, String shortcut, String details, String help){
    this.name = name;
    this.shortCut = shortCut;
    this.details  = details;
    this.help = help;
  }
```

[2]Los patrones de diseño de software en general, y el patrón *command* en particular, se estudian en la asignatura Ingeniería del Software.

[3]Una clase utilidad es aquella en la que todos los métodos son estáticos.

```
  public abstract boolean execute(Game game);

  public abstract Command parse(String[] commandWords);

  protected boolean matchCommandName(String name) {
    return this.shortcut.equalsIgnoreCase(name) ||
        this.name.equalsIgnoreCase(name);
  }

  public String helpText(){
    return details + " : " + help + "\n";
  }
}
```

De los métodos abstractos anteriores, execute se implementa invocando algún método con el objeto game pasado como parámetro y ejecutando alguna acción más. El método parse se implementa con un método que parsea el texto de su primer argumento (que es el texto proporcionado por el usuario por consola, dividido en palabras) y devuelve:

- o bien un objeto de una subclase de Command, si el texto que ha dado lugar al primer argumento se corresponde con el texto asociado a esa subclase,
- o el valor null, en otro caso.

La clase CommandGenerator contiene la siguiente declaración e inicialización de atributo:

```
private static Command[] availableCommands = {
  new ListCommand(),
  new HelpCommand(),
  new ResetCommand(),
  new ExitCommand(),
  new ListCommand(),
  new UpdateCommand(),
  new MoveCommand(),
  new ShockwaveCommand()
};
```

Este atributo se usa en los dos siguientes métodos de CommandGenerator:

- public static Command parseCommand(String[] commandWords), que, a su vez, invoca el método parse de cada subclase de Command, tal y como se ha explicado anteriormente,
- public static String commandHelp(), que tiene una estructura similar al método anterior, pero invocando el método helpText() de cada subclase de Command. Este método es invocado por el método execute de la clase HelpCommand.

Una de las ventajas de usar el patrón *Command* es que es muy fácil y natural implementar el "Crtl-Z" o undo. Al encapsular los comandos como objetos los podemos apilar y desapilar, yendo para adelante y para atrás en la historia del juego. No lo vamos a implementar en esta práctica pero seguro que durante tu carrera lo utilizas más de una vez.

2.2. Herencia y polimorfismo

Quizás la parte más frustrante y mayor fuente de errores de la primera práctica es tener que replicar código en los objetos del juego y en las listas de objetos. Esto lo vamos a resolver usando la herramienta básica de la programación orientada a objetos: la *herencia*. Para ello, se creará una jerarquía de clases que heredan 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.
- De GameObject heredará Ship y Weapon. La primera clase representará las naves del juego, mientras que la segunda representará los objetos que causan daño.
- La clase EnemyShip, que hereda de la clase Ship, representará todas las naves enemigas, incluyendo el Ovni. Entre otros elementos, esta clase deberá gestionar el movimiento de las naves y los puntos que se obtienen al ser destruidas por el jugador.
- De forma similar, la clase UCMShip también hereda de Ship y representará la nave que controla el jugador.
- 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.
- Intuitivamente, las clases RegularAlien y DestroyerAlien heredarán de AlienShip, implementando su comportamiento.
- Finalmente, las clases Bomb y UCMMissile heredan de Weapon, representando el movimiento de los disparos realizados por las naves DestroyerAlien y UCMShip, respectivamente.
- 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. De esta forma se consigue un diseño escalable, permitiendo añadir nuevos elementos en el juego de forma sencilla.

Adicionalmente, podemos utilizar la herencia para refactorizar el código de las listas. En esta práctica, utilizaremos una clase – llamada GameObjectBoard – 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.

2.3. 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 distiguir 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();
    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 fijaos 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.

El Game debe implementar estos métodos del 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). Los destroyers no hace 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.

En el anexo además de parte de la implementación del GameObject también encontrarás las cabeceras de la clase GameObjectBoard, donde se encapsula el comportamiento de la lista de objetos de juego. 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.

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

3.1. Incorporación de nuevos objetos de 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.

  Si has hecho bien la parte I y II de la práctica esta extensión te tiene que resultar sencilla.

4. Anexos a 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;

public interface IPlayerController {

    // PLAYER ACTIONS
    public boolean move (int numCells);
    public boolean shootLaser();
    public boolean shockWave();

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

}
```

```
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;

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;};
}
```

```
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;
    }
}
```