

---

# Cuadernillo de prácticas Tecnología de la Programación

---



Curso: 2017/2018

Departamento de Ingeniería del Software e Inteligencia Artificial  
Departamento de Sistemas Informáticos y Computación  
Facultad de Informática  
UNIVERSIDAD COMPLUTENSE DE MADRID



---

# Práctica 2: Mejora y extensión de la aplicación 2048

---

**Fecha de entrega:** 18 de diciembre de 2017, 09:00

**Objetivo:** Herencia, polimorfismo, clases abstractas, interfaces.

## 1. Introducción

En esta práctica, modificamos la aplicación desarrollada en la práctica anterior de tres formas.

- Primero, como se explica en la sección 2, refactorizamos<sup>1</sup> el código de la práctica anterior, eliminando parte del código del método `run` del controlador y distribuyendo su funcionalidad entre un conjunto de clases. Esto supone aplicar el que se conoce como *patrón command*.
- Segundo, como se explica en la sección 3, añadimos dos nuevos comandos a la aplicación. Al hacerlo, el alumno debería convencerse de que incorporar el patrón *command* a la aplicación ayuda a obtener un diseño fácilmente extensible y flexible.
- Por último, como se explica en la sección 4, añadimos la posibilidad de jugar dos variantes del juego 2048, a la vez que se mantiene la posibilidad de jugar al juego original.

---

<sup>1</sup>Refactorizar consiste en cambiar la estructura del código (se supone que para mejorarlo) sin cambiar lo que hace.

## 2. Refactorización de la solución de la práctica anterior

### 2.1. Descripción

El patrón *command* es un patrón de diseño<sup>2</sup> muy conocido. En esta práctica no necesitas conocer de este patrón más que lo que se explica aquí. Para aplicarlo, cada comando del juego se representa por una clase diferente, que llamamos `ExitCommand`, `MoveCommand`, `ResetCommand` y `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*<sup>3</sup> `CommandParser`, 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 `CommandParser` 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 `CommandParser` un objeto de la subclase que representa a ese comando y que puede, a su vez, ejecutar el comando.

### 2.2. Implementación

El código de la clase abstracta `Command` es el siguiente:

```
package tp.pr2.control.commands;

import tp.p2.control.Controller;
import tp.p2.logic.multigames.Game;

public abstract class Command {

    private String helpText;
    private String commandText;
    protected final String commandName;

    public Command(String commandInfo, String helpInfo){
        commandText = commandInfo;
        helpText = helpInfo;
        String[] commandInfoWords = commandText.split("\\s+");
        commandName = commandInfoWords[0];
    }
}
```

<sup>2</sup>Los patrones de diseño de software en general, y el patrón *command* en particular, se estudian en la asignatura Ingeniería del Software.

<sup>3</sup>Una clase utilidad es aquella en la que todos los métodos son estáticos.

```

    }

    public abstract void execute(Game game, Controller controller);
    public abstract Command parse(String[] commandWords, Controller controller);
    public String helpText(){return "    " + commandText + ": " + helpText;}
}

```

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.

Aquellas subclases de `Command` que corresponden a comandos sin parámetros — es decir, comandos formados por una sola palabra; de momento, todas menos la clase `MoveCommand` son de esta categoría — no heredan directamente de `Command` sino de una clase intermedia `NoParamsCommand`. Esta clase implementa el método `parse` haciendo uso del atributo `commandName` de la clase `Command`. Por ello las clases que derivan de `NoParamsCommand` solo necesitan implementar el método `execute`. Las subclases de `Command` que corresponden a comandos con parámetros — repetimos, de momento solo una — derivan directamente de `Command` y necesitan atributos para guardar el valor de sus parámetros. La clase `MoveCommand` tiene pues un atributo de tipo `Direction`.

La clase `CommandParser` contiene la siguiente declaración e inicialización de atributo:

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

Este atributo se usa en los dos siguientes métodos de `CommandParser`:

- `public static Command parseCommand(String[] commandWords, Controller controller)`, que, a su vez, invoca el método `parse` de cada subclase de `Command` como repetidamente se ha explicado más arriba,
- `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`.

La razón de que se pase el controlador, como argumento, al método `parseCommand` y este, a su vez, al método `parse`, así como al método `execute`, es poder invocar el método `setNoPrintGameState()` del controlador, cuando sea necesario.

### 2.3. Ejemplos de ejecución

Como se ha dicho, ya que esta primera mejora consiste en refactorizar el código de la práctica anterior, el comportamiento de la aplicación no debe cambiar. Sin embargo, la mejora nos da la oportunidad de cambiar algo, de forma sencilla, el mensaje de ayuda:

```

Command > help
The available commands are:
  help: print this help message.
  reset: start a new game.
  exit: terminate the program.
  move <direction>: execute a move in one of the directions: up, down, left, right.

```

### 3. Incorporación de nuevos comandos

#### 3.1. Descripción

Queremos añadir los siguientes nuevos comandos a nuestra aplicación:

- **undo**, que restablece el juego al estado que tenía antes del último movimiento,
- **redo**, que permite ejecutar de nuevo un comando previamente realizado.

El comando **undo** solo tiene que permitir **deshacer** los **últimos 20 movimientos**. Un **intento de deshacer** más produce un **mensaje de error**. El comportamiento de la aplicación a este respecto debería ser el mismo independientemente de cuántos movimientos se hayan ejecutado.

#### 3.2. Implementación

La incorporación de estos nuevos comandos requiere

- la creación de dos nuevas subclases **UndoCommand** y **RedoCommand**, herederas en último término de **Command**,
- la incorporación de dos métodos públicos **undo()** y **redo()**, a la **clase Game**,
- la incorporación de dos nuevos objetos comando al **atributo array availableCommands** de la clase **CommandParser**.

#### Cómo recuperar el estado previo

En el juego 2048, es difícil implementar los comandos **undo** y **redo** guardando información sobre los movimientos que se han hecho, pues estos no determinan de forma unívoca la situación del tablero. Es más fácil almacenar una representación compacta del estado previo del juego que consista en el estado del tablero y entonces implementar el comando **undo** recuperando simplemente el estado previo almacenado. Para hacer todo esto, creamos una clase **GameState** (cuyos objetos son *objetos valor*, como los de la **clase MoveResult** de la práctica anterior) con **dos atributos score y highest** de tipo **int** y un **atributo boardState** de tipo **int[][]**. Asimismo añadimos los siguientes métodos a la clase **Board**:

- **public int[][] getState()**, que produce la representación compacta a partir del estado del tablero actual,
- **public void setState(int[][] aState)**, que establece el estado del tablero actual a partir de la representación compacta pasada como argumento.

y los siguientes métodos a la clase **Game**:

- **public GameState getState()**, que devuelve el estado actual del juego, invocando, para ello, el método **getState** de la clase **Board**,
- **public void setState(GameState aState)**, que restablece el juego al que determina el estado pasado como argumento, invocando, para ello, el método **setState** de la clase **Board**.

Por último, evidentemente necesitamos también una clase que defina la estructura de datos que usamos para almacenar las sucesiones de representaciones compactas de estados. Esta clase debe tener, al menos, el siguiente atributo:

- `private static final int CAPACITY = 20;`

inicializado a 20, como se ha dicho, y los siguientes métodos públicos:

- `public GameState pop()`, que devuelve el último estado almacenado,
- `public void push(GameState state)`, que almacena un nuevo estado,
- `public boolean isEmpty()`, que devuelve si la estructura de datos está vacía.

No es necesario ningún método público más. Otra posibilidad es definir una estructura de datos genérica, similar a la clase `ArrayAsList`, de la práctica anterior, en cuyo caso el tipo devuelto por `pop()` y el tipo del parámetro de `push()` sería `Object` en lugar de `GameState`.

### Ejemplos de ejecución

La siguiente traza muestra el uso de los comandos `undo` y `redo`. Para que el ejemplo no sea muy largo, en la ejecución que se muestra se ha tomado 3 en lugar de 20 como valor de `CAPACITY`.

```
$> java tp.pr2.Game2048 4 2
```

```

-----
|      |      |  2  |      |
-----
|      |      |      |      |
-----
|      |  2  |      |      |
-----
|      |      |      |      |
-----
highest: 2          score: 0

```

```
Command > move up
```

```

-----
|  2  |  2  |  2  |      |
-----
|      |      |      |      |
-----
|      |      |      |      |
-----
|      |      |      |      |
-----
highest: 2          score: 0

```

```
Command > move left
```

```

-----
|  4  |  2  |      |      |
-----
|      |      |      |      |
-----
|      |      |  2  |      |
-----
|      |      |      |      |
-----
highest: 4          score: 4

```

Command > move down

```

-----
|  2  |   |   |   |
-----
|   |   |   |   |
-----
|   |   |   |   |
-----
|  4  |  2  |  2  |   |
-----
highest: 4      score: 4

```

Command > move right

```

-----
|   |   |  2  |  2  |
-----
|   |   |   |   |
-----
|   |   |   |   |
-----
|   |   |  4  |  4  |
-----
highest: 4      score: 8

```

Command > move right

```

-----
|   |   |   |  4  |
-----
|   |  2  |   |   |
-----
|   |   |   |   |
-----
|   |   |   |  8  |
-----
highest: 8      score: 20

```

Command > undo  
Undoing one move...

```

-----
|   |   |  2  |  2  |
-----
|   |   |   |   |
-----
|   |   |   |   |
-----
|   |   |  4  |  4  |
-----
highest: 4      score: 8

```

Command > undo  
Undoing one move...

```

-----
|  2  |   |   |   |
-----
|   |   |   |   |
-----
|   |   |   |   |
-----
|  4  |  2  |  2  |   |
-----
highest: 4      score: 4

```

Command > undo  
Undoing one move...



```

-----
|  4  |  2  |      |      |
-----
|      |      |      |      |
-----
|      |      |  2  |      |
-----
|      |      |      |      |
-----
highest: 4          score: 4

```

```

Command > undo
Command > Undo is not available

```

```

redo
Redoing one move...

```

```

-----
|  2  |      |      |      |
-----
|      |      |      |      |
-----
|      |      |      |      |
-----
|  4  |  2  |  2  |      |
-----
highest: 4          score: 4

```

```

Command > redo
Redoing one move...

```

```

-----
|      |      |  2  |  2  |
-----
|      |      |      |      |
-----
|      |      |      |      |
-----
|      |      |  4  |  4  |
-----
highest: 4          score: 8

```

```

Command > redo
Redoing one move...

```

```

-----
|      |      |      |  4  |
-----
|      |  2  |      |      |
-----
|      |      |      |      |
-----
|      |      |      |  8  |
-----
highest: 8          score: 20

```

```

Command > redo
Nothing to redo

```

```

Command > exit
Game over.....

```

La siguiente traza muestra otros aspectos de los comandos `undo` y `redo`.

```
$> java tp.pr2.Game2048 4 2
```

```

-----
|      |      |      |      |
-----
|      |  2  |      |      |
-----

```

```

-----
|   |   |   |   |
-----
|  2  |   |   |   |
-----
highest: 2          score: 0

```

Command > move up

```

-----
|  2  |  2  |   |   |
-----
|   |   |   |   |
-----
|   |   |   |   |
-----
|   |   |  2  |   |
-----
highest: 2          score: 0

```

Command > move left

```

-----
|  4  |  2  |   |   |
-----
|   |   |   |   |
-----
|   |   |   |   |
-----
|  2  |   |   |   |
-----
highest: 4          score: 4

```

Command > undo

Undoing one move...

```

-----
|  2  |  2  |   |   |
-----
|   |   |   |   |
-----
|   |   |   |   |
-----
|   |   |  2  |   |
-----
highest: 2          score: 0

```

Command > move right

```

-----
|   |   |   |  4  |
-----
|   |   |   |   |
-----
|   |   |   |   |
-----
|  4  |   |   |  2  |
-----
highest: 4          score: 4

```

Command > redo

Nothing to redo

Command > undo

Undoing one move...

```

-----
|  2  |  2  |   |   |
-----
|   |   |   |   |

```

```

-----
|   |   |   |   |
-----
|   |   | 2 |   |
-----
highest: 2          score: 0

Command > undo
Undoing one move...

-----
|   |   |   |   |
-----
|   | 2 |   |   |
-----
|   |   |   |   |
-----
| 2 |   |   |   |
-----
highest: 2          score: 0

Command > exit
Game over.....

```

## 4. Extensión de la aplicación con nuevos juegos

### 4.1. Descripción

Añadimos a nuestra aplicación la capacidad de jugar a las siguientes dos variantes del juego 2048:

- En la primera variante, los números del tablero son números de la sucesión de Fibonacci. Las células creadas inicialmente y las generadas después de cada jugada que haya producido algún movimiento de células, tienen el valor 1 con probabilidad del 90 % y valor 2 con probabilidad del 10 %. La operación de fusión funde células adyacentes que tengan valores que correspondan a números consecutivos de la sucesión de Fibonacci, dejando en una de ellas el número siguiente de la sucesión, que es la suma de estos dos, y vaciando la otra. Como en el juego 2048 original, el número de puntos que se obtiene de una fusión es igual al valor de la célula no vacía resultante de la fusión. El juego se gana cuando se alcanza una célula con el valor 144; se pierde cuando se alcanza un estado en el juego que no permite nuevos movimientos. En ambos casos, el juego termina mostrando un mensaje apropiado. Una implementación de esta variante del juego sobre una interfaz gráfica de usuario puede jugarse en la siguiente URL:  
<http://themoreyouknow.github.io/fib/>.
- En la segunda variante, los números del tablero son potencias de dos, como en el juego 2048 original. Las células inicialmente creadas y las generadas después de cada jugada que haya producido algún movimiento de células, tienen valor 2048 con probabilidad del 90 % y valor 1024 con probabilidad del 10 %. La operación de fusión funde células adyacentes que tengan el mismo valor, dejando en una de ellas la mitad del valor que tenía y vaciando la otra. El número de puntos que se obtiene de una fusión se calcula como sigue: 2 puntos tras la fusión de dos células con valor 2048, 4 puntos por la fusión de dos células con valor 1024, 8 puntos con dos de valor 512, etc. El juego se gana cuando se obtiene una célula con valor 1 y se pierde cuando se alcanza un estado en el juego que no permite nuevos movimientos. En ambos

casos, el juego termina mostrando un mensaje apropiado. Una implementación de esta variante del juego sobre una interfaz gráfica de usuario puede jugarse en la siguiente URL:

<https://jonastermeau.com/files/others/2048%5E0/>

Recuerda que:

- En el juego 2048 original, los números del tablero son potencias de dos. Las células inicialmente creadas y las generadas después de cada jugada que haya producido algún movimiento de células, tienen valor 2 con probabilidad del 90 % y valor 4 con probabilidad del 10 %. La operación de fusión funde células adyacentes que tengan el mismo valor, dejando en una de ellas el doble del que tenía y vaciando la otra. El número de puntos que se obtiene de una fusión es igual al valor de la célula resultante no vacía. El juego se gana cuando se obtiene una célula con valor 2048; se pierde cuando se alcanza un estado en el juego que no permite nuevos movimientos. En ambos casos, el juego termina mostrando un mensaje apropiado. Una implementación de este juego sobre una interfaz gráfica de usuario puede jugarse en la siguiente URL: <http://gabrielecirulli.github.io/2048/>

## 4.2. Implementación

### La interfaz `GameRules`

La implementación de la aplicación multi-juego se deduce de la presentación de los juegos que acabamos de dar. Definimos una interfaz `GameRules` que tiene los métodos siguientes:

- `void addNewCellAt(Board board, Position pos, Random rand)`, que incorpora una célula con valor aleatorio en la posición `pos` del tablero `board`,
- `int merge(Cell self, Cell other)`, que fusiona dos células y devuelve el número de puntos obtenido por la fusión,
- `int getWinValue(Board board)`, que devuelve el mejor valor del tablero, según las reglas de ese juego, comprobándose si es un valor ganador y se ha ganado el juego,
- `boolean win(Board board)`, que devuelve si el juego se ha ganado o no,
- `boolean lose(Board board)`, que devuelve si el juego se ha perdido o no.

Observa que la **condición de que se ha perdido en el juego** (que sucede cuando ningún movimiento ni fusión es posible) es la misma para los tres juegos. Sin embargo, queremos permitir la posibilidad de que futuros juegos tengan una implementación diferente de la condición de juego perdido, manteniendo el hecho de que los tres que presentamos tienen la misma. Una forma de llevar a cabo esto es usar la construcción **default implementation** de las interfaces de Java 8. Para ello declaramos el método `lose` usando la **palabra clave default** y proporcionamos una **implementación por defecto del mismo**, como parte de la interfaz. Se trata de algo similar a lo que ocurre cuando se proporciona un método concreto con su implementación, dentro de una clase abstracta<sup>4</sup>. De la misma forma que un método

<sup>4</sup>Las dos principales diferencias entre las clases abstractas y las interfaces de Java 8 es que estas últimas no pueden contener estado (es decir, atributos) y todos sus métodos son públicos.

concreto de una clase abstracta puede ser reescrito si así lo requiere una subclase, un default method de una interfaz de Java 8 puede ser implementado si lo requiere una clase implementadora de la interfaz.

Por las mismas razones que para el método lose, decidimos añadir tres default methods más a la interfaz:

default  
methods

- Board createBoard(int size), cuya implementación por defecto crea y devuelve un tablero size × size,
- addNewCell(Board board, Random rand), cuya implementación por defecto elige una posición libre de board e invoca el método addNewCellAt() para añadir una célula en esa posición,
- default void initBoard(Board board, int numCells, Random rand), cuya implementación por defecto inicializa board eligiendo numCells posiciones libres, e invoca el método addNewCellAt() para añadir nuevas células en esas posiciones.

### Implementación de la interfaz GameRules

Los que no son default methods de la interfaz GameRules tendrán una implementación distinta en cada una de las tres clases que implementan esta interfaz, a saber, Rules2048, RulesFib, y RulesInverse. Por cierto, para el juego de Fibonacci, puede ser útil el siguiente código:

```
package tp.pr2.util;

public class MyMathsUtil {

    // convert from long to int since we will not need to use large numbers
    public static int nextFib(int previous) {
        double phi = (1 + Math.sqrt(5)) / 2;
        return (int) Math.round(phi * previous);
    }
}
```

esto se  
puede usar  
para  
fibonacci

2048

141

La implementación del método getWinValue en las clases Rules2048 y RulesFib devuelve simplemente el valor máximo del tablero, mientras que en la clase RulesInverse devuelve el mínimo. Para obtener estos valores se puede implementar un método apropiado en la clase Board. El juego llamará a este método cuando sea necesario. Observa que si has implementado la clase MoveResults con un atributo que guarda el máximo del tablero después de cada movimiento, debes eliminar ahora ese atributo de esa clase pues ya no se usará. De forma análoga, este valor no debería ser tampoco un atributo de la clase GameState ya que, como se ha dicho, un método de Board lo proporciona cada vez que se necesite. Por último, en el método toString() de la clase Game, se usará el término “mejor valor” en lugar de “highest” ya que, en el juego inverso, el valor que interesa es el menor, no el mayor.

La clase Game tendrá un atributo GameRules currentRules que guarda las reglas del juego actual. Una instancia de la clase GameRules (que será una instancia de alguna de las tres clases que implementan GameRules) se pasa al juego cuando se crea y se copia en el atributo currentRules en la constructora de la clase Game. Observa que, aunque gran parte de los métodos de GameRules son invocados por métodos de la clase Game, es el método doMerge de la clase Cell el que invocará el método merge() de GameRules. Para

cuando se crea el objeto Rules2048 en la constructora de Game, se copia el objeto Rules2048 en el atributo currentRules

facilitar esto, el juego debe pasar el valor de su atributo `currentRules` al tablero y el tablero debe pasárselo entonces a las células. Por tanto, el método `executeMove()` de la clase `Board` y el método `doMerge()` de la clase `Cell` tendrán ahora un parámetro más, siendo sus declaraciones respectivas las siguientes:

- `public MoveResults executeMove(Direction dir, GameRules rules)`
- `public int doMerge(Cell neighbourCell, GameRules rules)`

Observa que se ha cambiado el tipo devuelto del método `doMerge` que pasa de `boolean` a `int` (número de puntos). Se entiende que si devuelve 0, no ha habido fusión.

Al arrancar la aplicación, se juega al juego 2048 original, como en la práctica anterior. Para poder jugar a otros juegos, se añade un nuevo comando que permite al usuario cambiar de juego.

### Incorporación del comando play

Añadimos un nuevo comando `play <game>`, donde `<game>` puede ser una de las siguientes cadenas: `original`, `fib` o `inverse`. Para definir el nuevo comando, creamos una nueva subclase de `Command` llamada `PlayCommand`. En la sección 3, ya hemos visto los cambios que supone añadir un nuevo comando a la aplicación.

Recuerda que, cuando arranca la aplicación de la práctica anterior, dos (o incluso tres) parámetros se pasan por línea de comandos para especificar el tamaño del tablero, el número de células iniciales y, si hace el caso, la semilla que se usa en la generación de números pseudo-aleatorios.

Cuando se cambia de juego con el comando `play`, la aplicación debe dar al usuario la posibilidad de proporcionar cada uno de estos tres valores y, si el usuario simplemente pulsa `return` para cualquiera de ellos, deben usarse los valores por defecto 4, 2, y la semilla elegida por el sistema, respectivamente. La clase `PlayCommand` necesitará atributos para almacenar cada uno de estos tres valores. Además, de la misma forma que la clase `MoveCommand` tiene un atributo de tipo `Direction` para guardar la dirección del comando `move`, la clase `PlayCommand` necesita un atributo para guardar el parámetro del tipo de juego del comando `play`. Este atributo será `GameType gameType`, donde `GameType` es un tipo enumerado con valores `ORIG`, `FIB` e `INV`.

El hecho de que se use una semilla nueva cada vez que se teclee el comando `play` se traduce en que se crea un nuevo objeto `Random` en un método de la clase `Game` cada vez que se elija este comando. Para uniformizar el tratamiento que se da al juego cuando arranca la aplicación y el que se quiere dar al juego seleccionado vía el comando `play`, se debe pasar una semilla al constructor de la clase `Game` (en la práctica anterior, el argumento del constructor de la clase `Game` podía ser la semilla o el objeto `Random` creado usando la semilla). El constructor de la clase `Game` tendrá entonces la forma:

- `public Game(GameRules rules, long seed, int dim, int initCells)`

### 4.3. Ejemplos de ejecución

La siguiente traza muestra el uso del comando `play`.

```
$> java tp.pr2.Game2048 4 2
```

```
-----
```

si el  
usuario  
no  
introduce  
nada

```

|   |   |   |   |
|---|
|  2  |   |   |  2  |
|---|
|   |   |   |   |
|---|
|   |   |   |   |
|---|
best value: 2          score: 0

```

Command > move left

```

|---|
|   |   |   |   |
|---|
|  4  |   |   |   |
|---|
|   |   |   |  2  |
|---|
|   |   |   |   |
|---|
best value: 4          score: 4

```

Command > help

The available commands are:

```

help: print this help message.
reset: start a new game.
exit: terminate the program.
undo: undo the last command.
redo: redo the last undone command.
move <direction>: execute a move in one of the directions: up, down, left, right.
play <game>: start a new game of one of the game types: original, fib, inverse.

```

Command > play fib

Please enter the size of the board:

Using the default size of the board: 4

Please enter the number of initial cells:

Using the default number of initial cells: 2

Please enter the seed for the pseudo-random number generator:

Using the default seed for the pseudo-random number generator: 924

```

|---|
|   |   |   |  1  |
|---|
|   |   |   |   |
|---|
|   |   |   |   |
|---|
|   |   |   |  1  |
|---|
best value: 1          score: 0

```

Command > move up

```

|---|
|   |   |   |  2  |
|---|
|   |   |   |   |
|---|
|   |  2  |   |   |
|---|
|   |   |   |   |
|---|
best value: 2          score: 2

```

Command > move left

```

|---|
|  2  |   |   |   |
|---|

```

```

-----
|   |   |   |   |
-----
|  2  |   |   |   |
-----
|  1  |   |   |   |
-----
best value: 2      score: 2

```

Command > move down

```

-----
|   |   |   |   |
-----
|   |   |   |  1 |
-----
|  2  |   |   |   |
-----
|  3  |   |   |   |
-----
best value: 3      score: 5

```

Command > move down

```

-----
|   |   |  1  |   |
-----
|   |   |   |   |
-----
|   |   |   |   |
-----
|  5  |   |   |  1 |
-----
best value: 5      score: 10

```

Command > play inverse

Please enter the size of the board: 3

Please enter the number of initial cells: 2

Please enter the seed for the pseudo-random number generator:

Using the default seed for the pseudo-random number generator: 625

```

-----
|   |   |   |
-----
| 2048 | 2048 |   |
-----
|   |   |   |
-----
best value: 2048    score: 0

```

Command > move left

```

-----
|   |   | 2048 |
-----
| 1024 |   |   |
-----
|   |   |   |
-----
best value: 1024    score: 2

```

Command > move right

```

-----
|   |   | 2048 |
-----
| 2048 |   | 1024 |
-----
|   |   |   |

```



```

-----
best value: 1024      score: 2

Command > play original
Please enter the size of the board: 6
Please enter the number of initial cells: 4
Please enter the seed for the pseudo-random number generator:
Using the default seed for the pseudo-random number generator: 226

```

```

-----
|   |   |   |   |   |   |
-----
|  2 |   |   |   |   |  4 |
-----
|   |   |   |   |   |   |
-----
|  2 |   |   |   |   |   |
-----
|   |   |   |   |  2 |   |
-----
|   |   |   |   |   |   |
-----
best value: 4      score: 0

```

Command > move up

```

-----
|  4 |   |   |   |  2 |  4 |   |
-----
|   |   |   |   |   |   |   |
-----
|   |   |   |   |   |   |   |
-----
|   |   |   |   |   |   |   |
-----
|   |   |   |   |   |   |   |
-----
|   |   |   |  2 |   |   |   |
-----
best value: 4      score: 4

```

Command > move down

```

-----
|   |   |   |   |   |   |   |
-----
|   |   |   |   |   |   |   |
-----
|   |   |   |   |   |   |   |
-----
|   |   |   |   |   |   |   |
-----
|  2 |   |   |   |   |   |   |
-----
|  4 |   |   |  2 |  2 |  4 |   |
-----
best value: 4      score: 4

```

```

Command > exit
Game over.....

```

La siguiente traza muestra el uso del comando `play` y algunos mensajes de error.

```
$> java tp.pr2.Game2048 4 2
```

```

-----
|   |   |   |   |   |
-----
|   |   |   |   |   |

```

```

-----
|  2  |   |   |   |
-----
|   |   |   |  2 |
-----
best value: 2          score: 0

```

Command > move it move it  
Unknown direction for move command

Command > move  
Move must be followed by a direction: up, down, left, right

Command > move left

```

-----
|   |   |   |   |
-----
|  2  |   |   |   |
-----
|  2  |   |   |   |
-----
|  2  |   |   |   |
-----
best value: 2          score: 0

```

Command > undoit  
Unknown command. Use 'help' to see the available commands

Command > help  
The available commands are:  
 help: print this help message.  
 reset: start a new game.  
 exit: terminate the program.  
 undo: undo the last command.  
 redo: redo the last undone command.  
 move <direction>: execute a move in one of the directions: up, down, left, right.  
 play <game>: start a new game of one of the game types: original, fib, inverse.

Command > undo  
Undoing one move...

```

-----
|   |   |   |   |
-----
|   |   |   |   |
-----
|  2  |   |   |   |
-----
|   |   |   |  2 |
-----
best value: 2          score: 0

```

Command > play something else  
Unknown game type for play command

Command > play  
Play must be followed by a game type: original, fib, inverse

Command > play fib  
 Please enter the size of the board: 4 6  
 Please provide a single positive integer or press return  
 Please enter the size of the board: -4  
 The size of the board must be positive  
 Please enter the size of the board:  
 Using the default size of the board: 4  
 Please enter the number of initial cells: 4600  
 Please enter the seed for the pseudo-random number generator:  
 Using the default seed for the pseudo-random number generator: 353  
 The number of initial cells must be less than the number of cells on the board

```
Command > why?  
Unknown command. Use 'help' to see the available commands  
  
Command > exit  
Game over.....
```

## 5. Entrega de la práctica

La práctica debe entregarse utilizando el mecanismo de entregas del campus virtual, no más tarde de la fecha y hora indicada en la cabecera de la práctica. Debes subir un fichero comprimido (.zip) que contenga al menos lo siguiente<sup>5</sup>. No incluyas los ficheros que resultan de la compilación (los del directorio `bin`).

- Directorio `src` con el código fuente de todas las clases de la práctica.
- Directorio `doc` con la documentación de la práctica generada con `javadoc`, si así lo requiere el profesor.
- Fichero `alumnos.txt` donde se indicará el nombre de los componentes del grupo.

---

<sup>5</sup>Puedes incluir también opcionalmente los ficheros de información del proyecto de Eclipse.