Las variantes del juego

La clase CommandParser

```
public class CommandParser {
    private final static Command[] commands=
      {new Help(), new Quit(), new Reset(),
       new Run(), new Replace(0)};

    public static Command parseCommand(String line, …) {
      //Quitar blancos y dividir en tokens
      if (tokens.length==0 || tokens.length>2) ...
      Command cm;
      for (Command c:commands) {
           cm=c.parse(tokens);
           if (cm!=null) return cm;
      }
```

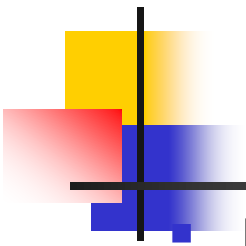
La clase abstracta **Command**

```
public abstract class Command {
    ...
    public abstract void execute(...);
    public abstract Command parse(String[] tokens, …);
    public String helpText() {...}
}
```

La clase **Reset**

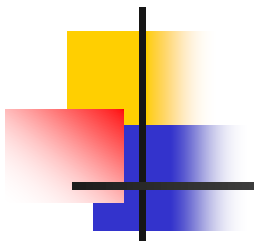
```
public class Reset extends Command {
    ...
    public void execute(Game g, ...) {
       g.executeReset();
    }

    public Command parse(String[] s) {
       if (s.length==1 &&
           s[0].equalsIgnoreCase("RESET"))
           return new Reset();
       else return null;
    }
    ...
}
```



La clase `Replace`

```
public class Replace extends Command {
    private int n;
    ...
    public void execute(Game g, ...) {
       g.executeReplace();
    }
```



La clase Replace

```
    public Command parse(String[] s) {
        if (s.length!=2) return null;
        if (!s[0].equalsIgnoreCase("REPLACE")) return null;

        //Escaneamos s[1] en una variable m

        if (m no es entero) return null;
        else return new Replace(m);
    }
    ...
}
```

La clase **Controller**

```
public void run() {
    while (!game.isFinished()) {
        ...
        System.out.print("Command > ");
        String[] words =
            scanner.nextLine().toLowerCase().
                trim().split(“ s+”);
        Command command =
            CommandParser.parseCommand(words, this);
        if (command != null)
            command.execute(game, this);
        else ...
```

Particularidades de la Pr. 2

- La clase **MoveCommand** es como la clase **Replace**, pero en lugar de tener un atributo entero tiene un atributo de tipo **Direction**
- Cada llamada **execute()** del método **run()** de la clase **Controller**:

  command.execute(game, this);

  se transforma en:

  game.reset() si **command** fue **new Command()**

  game.undo() si **command** fue **new Undo()**

  //Análogamente para **new Redo()**, **new Exit()**

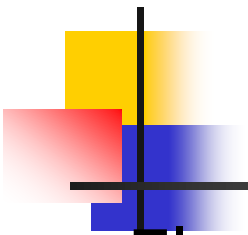
  game.move(dir) si **command** fue **new MoveCommand(dir)**

  game.newGame(gameRules, seed, boardSize, initCells)

  si **command** fue **new Play(gameType)**, donde **gameRules=gameType.getGameRules()**

La clase **Game**

- Aparte de los atributos que tenía antes, ahora tiene un atributo **`GameRules gr`**, más las pilas de **`ues`** y **`res`**
- Los métodos **`undo()`** y **`redo()`** de esta clase fueron explicados en las transparencias **`undo/redo`**
- El método **`move(dir)`**:
  - Vacía la pila de **`redo's`**
  - Guarda el estado actual (construyendo un **`GameState`** a partir de (1) la matriz de números del tablero –que la obtiene del atributo **`Board board`** de **`Game`**; y (2) el **`score`** –que lo obtiene del atributo **`score`** de **`Game`**) en la pila de **`ues`**
  - Le manda a **`board`** que ejecute el movimiento (como en la Pr. 1), pero pasándole las reglas del juego: **`board.executeMove(dir, gr);`**
  - Actualiza **`highest`** (ahora llamado **`valorGanador`**), añade una nueva celda si ha habido movimiento, comprueba si se ha ganado o se ha perdido. Todo esto último lo hacen métodos de **`GameRules`** invocados por el atributo **`gr`**

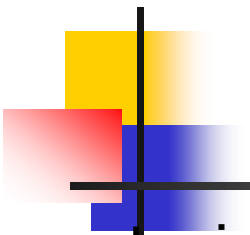


La clase **Game**

- El método **`newGame(GameRules gameRules, …)`**:
  - Da valor a los atributos **`myRandom`**, **`size`**, **`initCells`** a partir de los parámetros que aparecen en los puntos suspensivos
  - Da valor al atributo **`gr`** de **`Game`**:

```
gr = gameRules;
```

  - Resetea el juego



La interfaz **GameRules**

- La interfaz **`GameRules`** tiene los siguientes métodos:
  - **abstract methods** (consultar el enunciado de la práctica): **`void addNewCellAt(…)`**, **`int merge(…)`**, **`int getWinValue(…)`**, **`boolean win(…)`**
  - **default methods** (consultar el enunciado): **`boolean lose(…)`**, **`Board createBoard(…)`**, **`void addNewCell(…)`**, **`void initBoard(…)`**
  - **static methods** (opcionales): métodos para obtener un nuevo valor de celda, 1 nueva posición libre, n nuevas posiciones libres

Implementaciones de **GameRules**

- La clase **Rules2048**:
  - **void addNewCellAt(board, pos, rand)** deja en la posición **pos** de **board** un valor aleatorio generado con **rand**
  - **int merge(cell, otherCell)** es como el antiguo **doMerge**(). Recuerda que ahora el **doMerge(cell, gameRules)** de **Cell** lo hace **gameRules.merge(…)**
  - **int getWinValue(board)** busca el valor máximo del tablero
  - **boolean win(board)** busca 2048 en el tablero

Implementaciones de **GameRules**

- La clase **RulesInverse**:
  - **void addNewCellAt(board, pos, rand)** como el de la clase **Rules2048**, pero el valor aleatorio generado con **rand** se obtiene con otros parámetros
  - **int merge(cell, otherCell)** como el de la clase anterior, pero los puntos obtenidos son otros
  - **int getWinValue(board)** busca el valor mínimo del tablero
  - **boolean win(board)** busca 1 en el tablero

Implementaciones de **GameRules**

- La clase **RulesFib**:
  - **void addNewCellAt(board, pos, rand)** como los anteriores
  - **int merge(cell, otherCell)** como los anteriores. Para averiguar si dos números de Fibonacci son consecutivos utiliza **nextFib()** del enunciado de la práctica
  - **int getWinValue(board)** busca el valor máximo del tablero
  - **boolean win(board)** busca 144 en el tablero