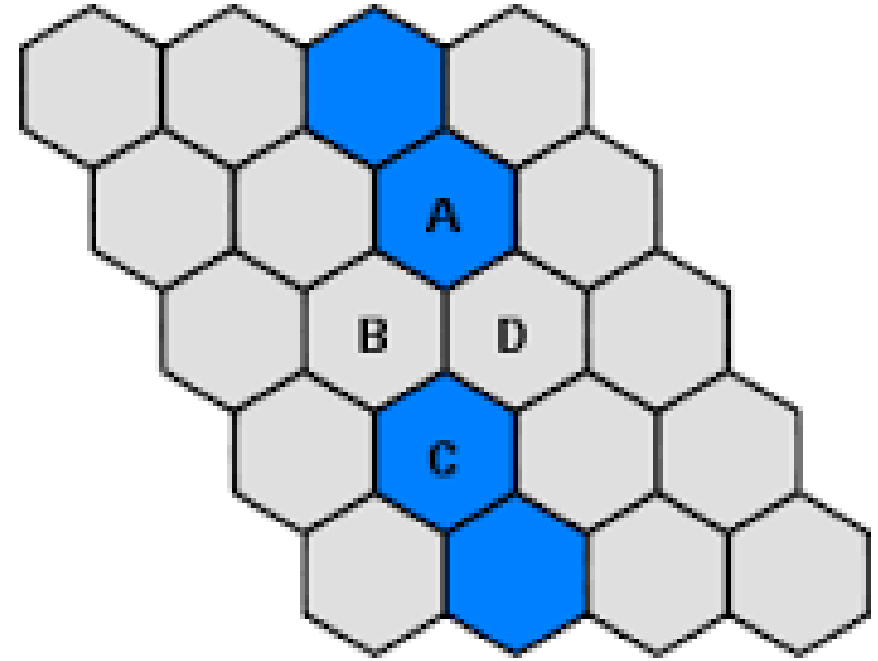
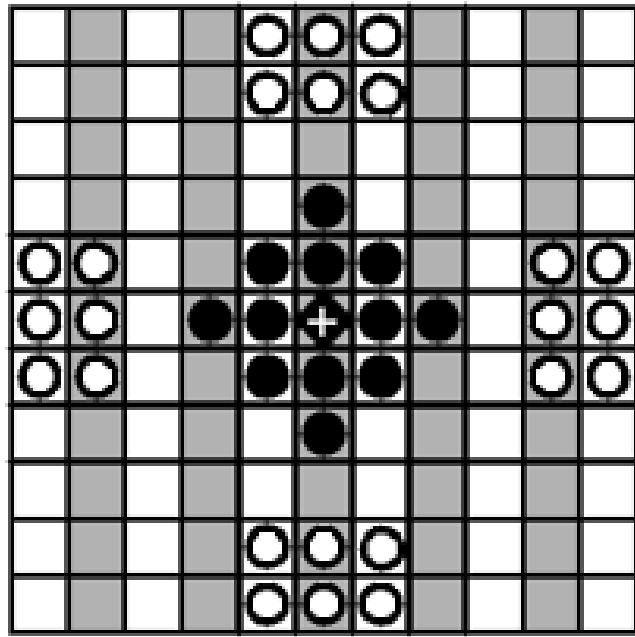


Juegos de Tablero (Board Games)

- ♦ Basecode para usar en todas la prácticas: descargar [basecode.zip](#) y usarlo para crear un nuevo proyecto en Eclipse
- ♦ Ejemplos de estas transparencias: descargar [misc.zip](#) y exrear en el directorio [src/es/ucm/fdi/tp](#)

¿Qué son los Juegos de Tablero?



- ♦ Fichas — Pieces (Counters)
- ♦ Tablero — Board
- ♦ Reglas de Juego — Game Rules

Piece: una Clase para Fichas

```
public class Piece implements java.io.Serializable {  
    private String id;  
    public Piece() {  
        id = generateId();  
    }  
    public Piece(String id) {  
        this.id = id;  
    }  
    public String getId() { return id; }  
    @Override  
    public int hashCode() { ... }  
    @Override  
    public boolean equals(Object obj) { ... }  
    @Override  
    public String toString() {  
        return id;  
    }  
}
```

Cada ficha (Piece) tiene un identificador, normalmente único en el contexto de un juego pero no tenemos que preocuparnos de este tema ahora ...

Podemos construir con un identificador asignado de una manera automática.

... o bien proporcionar el identificador.

Consultar el identificador

Fichas con el mismo id tienen que ser iguales (**equal**) y tener el mismo código de hash (**hashCode**).

La Interfaz Board

```
public interface Board extends java.io.Serializable {  
    public int getRows();  
    public int getCols();  
    public Piece getPosition(int row, int col);  
    public void setPosition(int row, int col, Piece p);  
    public boolean isFull();  
    public boolean isEmpty();  
    public void setPieceCount(Piece p, Integer n);  
    public Integer getPieceCount(Piece p);  
    public Board copy();  
}
```

consultar las dimensiones.

Consultar la ficha que está colocada en una posición

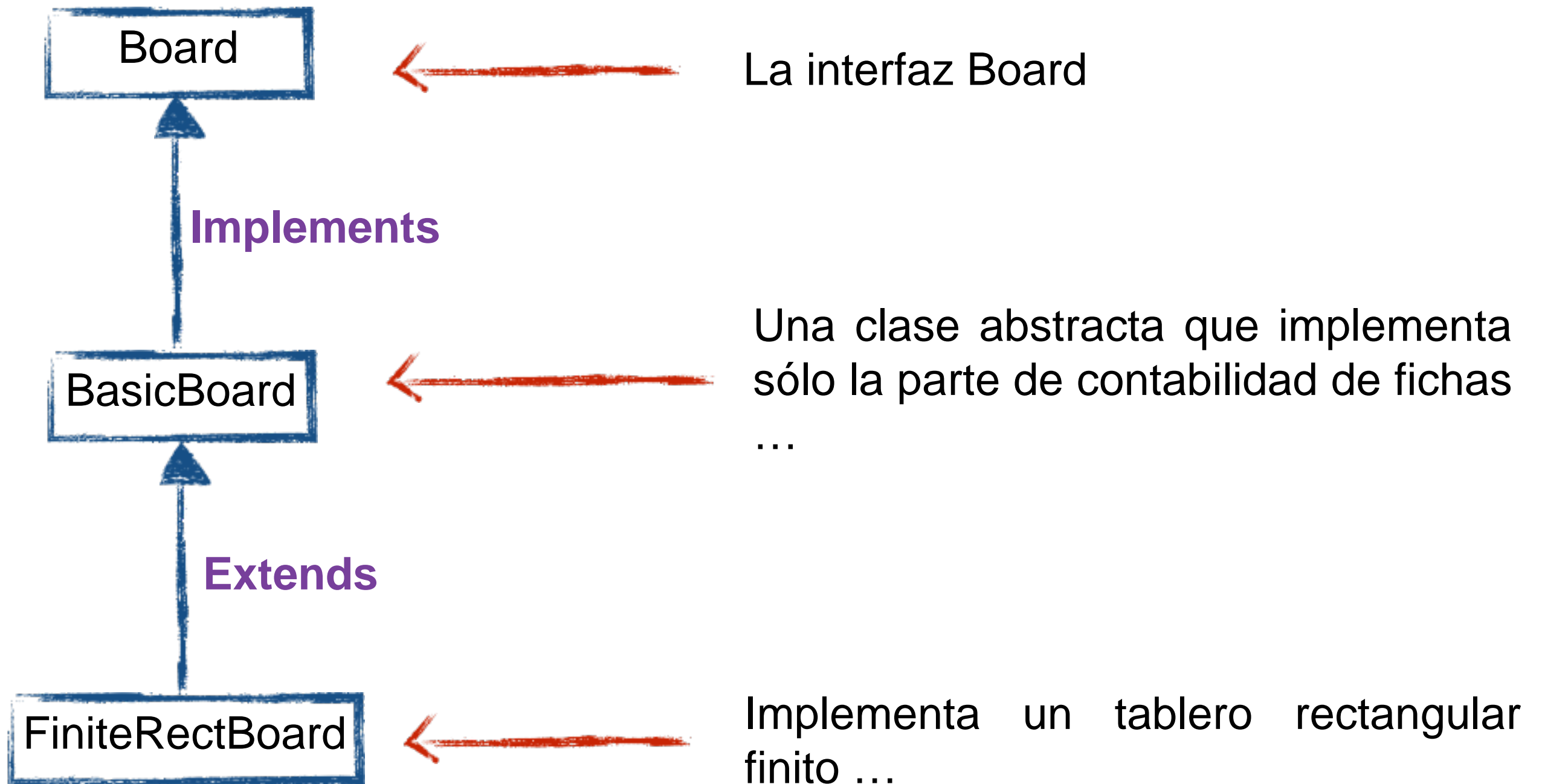
Colocar una ficha en una posición. Usamos **null** para posición vacía.

Consultar si el tablero está lleno o vacío.

“Contabilidad” de fichas, donde **null** significa que no hay “contabilidad” para esa ficha.

Crear una copia del tablero. Modificando uno no tiene que afectar al otro
...

Tableros en el Código Base



Usar las Fichas y el Tablero ...

```
public class ex1 {  
    private static void print(Board board) {  
        System.out.println(board);  
        System.out.println();  
    }  
  
    public static void main(String[] args) {  
        Piece x = new Piece("X");  
        Piece o = new Piece("O");  
        Board board = new FiniteRectBoard(3, 3);  
  
        print(board);  
  
        board.setPosition(1, 1, x);  
        board.setPosition(2, 2, o);  
        board.setPosition(1, 0, x);  
  
        print(board);  
  
        board.setPosition(2, 2, null);  
  
        print(board);  
    }  
}
```

Escribir un tablero en la consola (mediante el método toString).

Crear fichas

Crear un tablero

Añadir algunas fichas al tablero

Quitar una ficha del tablero

Construyendo un Juego ...

```
public class ex2 {
```

```
    ...  
    public static void main(String[] args) {  
        Scanner in = new Scanner(System.in);
```

```
        Piece x = new Piece("X");  
        Piece o = new Piece("O");  
        Board board = new FiniteRectBoard(5, 5);
```

```
        Piece turn = x;  
        boolean finished = false;  
        while ( !finished ) {  
            print(board);
```

```
            System.out.print("Type a move to place "+turn+":");  
            int row = in.nextInt();  
            int col = in.nextInt();
```

```
            board.setPosition(row, col, turn);
```

```
            turn = (turn.equals(x) ? o : x);
```

```
        }
```

```
    }
```

```
}
```

Crear fichas

Crear tablero

Pedir un movimiento al jugador

Ejecutar el movimiento

Cambiar turno

El Juego ConnectN

```
Piece x = new Piece("X");  
Piece o = new Piece("O");  
Board board = new FiniteRectBoard(5, 5);
```

Crear fichas

```
Piece turn = x;  
boolean finished = false;  
Piece winner = null;
```

Crear tablero

Elegir el primer jugador

```
while (!finished) {  
    print(board);
```

Pedir un movimiento al jugador

```
    System.out.print("Type a move to place " + turn + " :");  
    int row = in.nextInt();  
    int col = in.nextInt();
```

Se queremos hace otro juego muy parecido, tenemos que modificar las regels del juego (parte amarilla) — **modificación profunda del código!!**

```
    board.setPosition(row, col, turn);
```

Ejecutar

```
    if ( hasLine(board, turn) ) {  
        finished = true;  
        winner = turn;  
    } else if ( board.isFull() ) {  
        finished = true;  
    } else {  
        turn = (turn.equals(x) ? o : x);  
    }  
}
```

Decidir se ha terminado la partida

```
System.out.println("Game Over!" + (winner==null ? "No hay ganador" : winner) + ".");
```

Cambiar el turno (si no haya terminado)

Abstracción de las Reglas de Juego

```
GameRules rules = new ConnectNRules(5);
```

Crear las reglas del Juego

```
List<Piece> pieces = new ArrayList<>();  
pieces.add( new Piece("X") );  
pieces.add( new Piece("O") );
```

Todas la fichas van en una lista

```
Board board = rules.createBoard(pieces);
```

Pedir a las reglas
un tablero inicial

```
Piece turn = rules.initialPlayer(board, pieces);  
Piece winner = null;  
boolean finished = false;
```

Pedir a las reglas elegir
el primer jugador

```
while (!finished) {
```

```
    ...
```

```
}
```

```
...
```

En la siguiente transparencia ...

```
System.out.println("Game Over!" + (winner==null ? "Draw" : winner+" won")+".");
```

Abstracción de las Reglas de Juego

```
while (!finished) {  
    print(board);  
  
    System.out.print("Type a move to place " + turn + ":");  
    int row = in.nextInt();  
    int col = in.nextInt();  
  
    board.setPosition(row, col, turn);  
  
    Pair<State, Piece> x = rules.updateState(board, pieces, turn);  
  
    switch ( x.getFirst() ) {  
    case Won:  
        finished = true;  
        winner = x.getSecond();  
        break;  
    case Draw:  
        finished = true;  
        break;  
    case InPlay:  
        turn = rules.nextPlayer(board, pieces, turn);  
        break;  
    default:  
        throw ...  
    }  
}
```

Pedir un movimiento al jugador

Ejecutar el movimiento

Pedir a las reglas que calculen el siguiente estado del juego

Pedir a las reglas que elijan el siguiente jugador ...

¿Qué hemos conseguido?

```
GameRules rules = new ConnectNRules(5);
```

```
List<Piece> pieces = new ArrayList<>();  
pieces.add( new Piece("X") );  
pieces.add( new Piece("O") );
```

```
Board board = rules.createBoard(pieces);
```

```
Piece turn = rules.initialPlayer(board, pieces);  
Piece winner = null;  
boolean finished = false;
```

```
while (!finished) {  
    ...  
}  
...
```

```
System.out.println("Game Over!" + (winner==null ? "Draw" : winner+" won")+".");
```

Para cambiar a otro juego
(pero muy parecido), sólo
tenemos que cambiar las
reglas del juego:

`new TicTacToeRules()`

El resto no se toca!

Interfaz GameRule

```
public interface GameRules {  
    public String gameDesc();  
    public Board createBoard(List<Piece> pieces);  
    public Piece initialPlayer(Board board, List<Piece> pieces);  
    public int minPlayers();  
    public int maxPlayers();  
    public Pair<State, Piece> updateState(Board board, List<Piece> pieces, Piece turn);  
    public Piece nextPlayer(Board board, List<Piece> pieces, Piece turn);  
    public double evaluate(Board board, List<Piece> pieces, Piece turn);  
    public List<GameMove> validMoves(Board board, List<Piece> pieces, Piece turn);  
}
```

Para implementar otro juego, implementamos esta interfaz.

Abstracción de (ejecución de) Movimientos

```
while (!finished) {  
    print(board);  
  
    System.out.print("Type a move to place " + turn + ":");  
    int row = in.nextInt();  
    int col = in.nextInt();  
  
    board.setPosition(row, col, turn);  
  
    ...  
}
```

Pedir un movimiento al jugador

Un juego puede colocar un ficha en la posición (row,col), mientras que otro puede colocar la ficha y cambiar otras, etc.

Ejecutar el movimiento

Crear un movimiento y pedirle que se ejecute sobre el tablero. Cómo lo hace no importa ...

```
while (!finished) {  
    print(board);  
  
    System.out.print("Type a move to place " + turn + ":");  
    int row = in.nextInt();  
    int col = in.nextInt();  
  
    GameMove m = new ConnectNMove(row, col, turn);  
    m.execute(board, pieces);  
  
    ...  
}
```

Cualquier cambio en la "semántica" del movimiento no afecta a nuestro código

La Clase Abstracta GameMove

```
public abstract class GameMove implements java.io.Serializable {  
    private Piece piece;  
    protected GameMove(Piece piece) {  
        this.piece = piece;  
    }  
    public Piece getPiece() {  
        return this.piece;  
    }  
    public abstract void execute()  
}
```

```
public class ConnectNMove extends GameMove {  
    protected int row;  
    protected int col;  
    public ConnectNMove(int row, int col, Piece p) {  
        super(p);  
        this.row = row;  
        this.col = col;  
    }  
    @Override  
    public void execute(Board board, List<Piece> pieces) {  
        if (board.getPosition(row, col) == null) {  
            board.setPosition(row, col, getPiece());  
        } else {  
            throw new GameError("...");  
        }  
    }  
}
```

Abstracción de Jugadores

```
while (!finished) {  
    print(board);  
  
    System.out.print("Type a move to place " + turn + ":");  
    int row = in.nextInt();  
    int col = in.nextInt();  
    GameMove m = new ConnectNMove(row, col, turn);  
  
    m.execute(board, pieces);  
}
```

Esta parte es todavía demasiado concreta.

Un movimiento siempre se pide por consola, y siempre es una posición.

· Cuando programas, hazlo de una manera **Abstracta** y luego rellenas los **detalles** así permites a otros rellenas distintos **detalles** approve

```
Player p = new ConnectNConsolePlayer(new Scanner(System.in));  
// Player p = new ConnectNRandomPlayer();
```

```
...  
while (!finished) {  
    print(board);
```

```
    GameMove m = p.requestMove(turn, board, pieces, rules)  
    m.execute(board, pieces);
```

```
    ...  
}
```

Pedimos a un jugador un movimiento, cómo lo va a hacer no es importante ...

La Clase Abstracta Player

```
public interface Player implements java.io.Serializable {  
    abstract GameMove requestMove(Piece p, Board board, List<Piece> pieces, ...);  
}  
  
public class ConnectNConsolePlayer implements Player {  
  
    public class ConnectNRandomPlayer implements Player {  
  
        @Override  
        public GameMove requestMove(Piece p, Board board, List<Piece> pieces, ...) {  
            ...  
  
            int row = Utils.randomInt(rows);  
            int col = Utils.randomInt(cols);  
  
            while (true) {  
                if (board.getPosition(row, col) == null) {  
                    return createMove(row, col, p);  
                }  
                col = (col + 1) % board.getCols();  
                if (col == 0) {  
                    row = (row + 1) % board.getRows();  
                }  
            }  
        }  
    }  
}
```


Una Configuración más ...

```
GameRules rules = new ConnectNRules(5);
```

```
List<Piece> pieces = new ArrayList<>();  
pieces.add(new Piece("X"));  
pieces.add(new Piece("O"));
```

Que hace esta configuración?

```
Map<Piece,Player> ps = new HashMap<Piece,Player>();  
ps.put( pieces.get(0), new ConnectNConsolePlayer(new Scanner(System.in)));  
ps.put( pieces.get(1), new ConnectNRandomPlayer());
```

...

```
while (!finished) {
```

...

```
    GameMove m = ps.get(turn).requestMove(turn, board, pieces, rules);  
    m.execute(board, pieces);
```

...

```
}
```

...

Main, Control y Modelo

- ♦ Nuestro código tiene una mezcla de conceptos.
- ♦ Queremos seguir abstrayendo para aclarar estos conceptos, y sobretodo facilitar el reemplazamiento de unos componentes por otros sin tener que modificar el código.
- ♦ Separar la **Lógica** del Juego de su **Control**.
- ♦ El **Control** es el bucle que alterna entre los jugadores.
- ♦ El **Modelo** es todo lo que cambia (directamente) el estado de juego.
- ♦ **Main** es la parte que conecta todos lo componentes.

Controlador: Abstracción del Control

```
public class ConsoleCtrl {
    protected Game game;
    protected List<Piece> pieces;
    protected Map<Piece, Player> players;

    public ConsoleCtrl(Game game, List<Piece> pieces, List<Player> players, ...) {
        this.game = game;
        this.pieces = pieces;
        this.players = new HashMap<Piece, Player>();
        for (int i = 0; i < pieces.size(); i++) {
            this.players.put(pieces.get(i), players.get(i));
        }
    }

    public void makeMove(Player p) {
        game.makeMove(players.get(game.getTurn()));
    };

    public void start() {
        game.start(pieces);

        while (game.getState() == State.InPlay) {
            System.out.println(game);
            game.makeMove(players.get(game.getTurn()));
        }
    }
}
```

En el **basecode**, también pregunta (en la consola) si queremos si queremos seguir jugando ...

Modelo: Abstracción de la Lógica

```
public class Game implements Observable<GameObserver> {
```

```
    private Board board;  
    private List<Piece> pieces;  
    private GameRules rules;  
    private Piece turn;  
    private State state;  
    private Piece winner;
```

El estado interno incluye el tablero, las fichas, el estado del juego, etc.

```
    public Game(GameRules rules) {  
        this.rules = rules;  
        this.state = State.Starting;  
        this.turn = this.winner = null;  
    }
```

La constructora recibe las reglas de juego ...

```
    public void start(List<Piece> pieces) { ... }  
    public void restart() { ... }  
    public void stop() { state = Stopped }  
    public String gameDesc() { return rules.gameDesc(); }  
    public List<Piece> getPlayersPieces() { return pieces; }  
    public Game.State getState() { return state; }  
    public Piece getTurn() { return turn; }  
    public Piece getWinner() { return winner; }  
    public void makeMove(Player player) { ... }
```

Consultar el estado y ejecutar operaciones sobre el modelo ..

Main: Conectar los Componentes

```
public static void main(String[] args) {  
    GameRules rules = new ConnectNRules(5);  
    Game game = new Game(rules);
```

Crear reglas y juego

Crear las fichas

Crear los jugadores

```
List<Piece> pieces = new ArrayList<>();  
pieces.add(new Piece("X"));  
pieces.add(new Piece("O"));
```

```
List<Player> players = new ArrayList<Player>();  
players.add( new ConnectNConsolePlayer(new Scanner(System.in)));  
players.add( new ConnectNRandomPlayer());
```

```
ConsoleCtrl ctrl = new ConsoleCtrl(game, pieces, players, ...);  
ctrl.start();
```

```
}
```

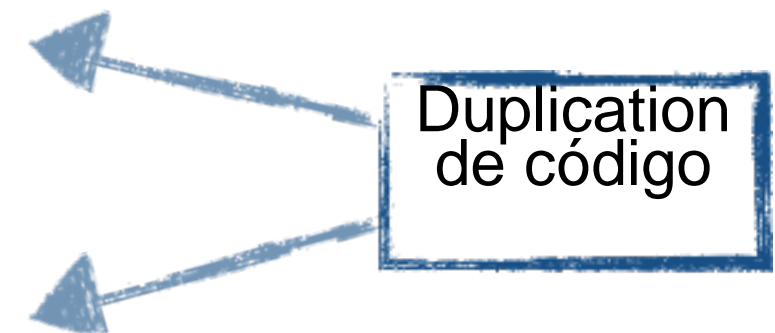
Crear y empezar el controlador

Añadir más Juegos ...

```
public static void main(String[] args) {
    GameRules rules = null;
    List<Piece> pieces = new ArrayList<Piece>();
    List<Player> players = new ArrayList<Player>();

    if (args[0].equals("cn")) {
        rules = new ConnectNRules(5);
        pieces.add(new Piece("X"));
        pieces.add(new Piece("O"));
        players.add(new ConnectNConsolePlayer(
            new Scanner(System.in)));
        players.add(new ConnectNRandomPlayer());
    } else if (args[0].equals("ttt")) {
        rules = new TicTacToeRules();
        pieces.add(new Piece("B"));
        pieces.add(new Piece("W"));
        players.add(new TTTConsolePlayer(new Scanner(System.in)));
        players.add(new TTTRandomPlayer());
    }

    Game game = new Game(rules);
    ConsoleCtrl ctrl = new ConsoleCtrl(game, pieces, players);
    ctrl.start();
}
```



Requiere una modificación profunda del código ...

Factoría: Abstracción de Creación

```
public static void main(String[] args) {  
    GameFactory factory = null;  
  
    if (args[0].equals("cn")) {  
        factory = new ConnectNFactory(5);  
    } else if (args[0].equals("tft")) {  
        factory = new TicTacToeFactory();  
    } else {  
        throw new GameError("Uknown game");  
    }  
}
```

La diferencia entre los juegos a este nivel es sólo en cómo creamos los componentes. La factoría es una abstracción de este concepto ...

No construimos un componente, lo pedimos a la factoría

```
GameRules rules = factory.gameRules();  
List<Piece> pieces = factory.createDefaultPieces();  
List<Player> players = new ArrayList<Player>();  
players.add( factory.createConsolePlayer() );  
players.add( factory.createRandomPlayer() );  
Game game = new Game(rules);  
ConsoleCtrl ctrl = new ConsoleCtrl(game, pieces, players, ...);  
ctrl.start();  
}
```

Esta parte no cambia, añadir más juegos no requiere modificación profunda.

La Interfaz Factory

```
public interface GameFactory {  
  
    public abstract GameRules gameRules();  
  
    public abstract Player createConsolePlayer();  
  
    public abstract Player createRandomPlayer();  
  
    public abstract Player createAIPlayer(AIAlgorithm alg);  
  
    public abstract List<Piece> createDefaultPieces();  
  
    public void createConsoleView(Observable<GameObserver> game, Controller ctrl);  
  
    public void createSwingView( ... );  
}
```



Modificar La Vista ...

Modificación profunda!! Lo se escribe no se toca!

Que hay que modificar si quememos cambiar la salida del programa?

```
public class ConsoleCtrl {  
    ...  
    public void start() {  
        game.start(pieces);  
  
        while (game.getState() == State.InPlay) {  
            System.out.println(game);  
            game.makeMove(players.get(game.getTurn()), ...  
        }  
    }  
}
```

Tenemos que modificar esta línea y al mejor toString de Game o otras clases ...



Abstracción de la Vista

```
public static void main(String[] args) {  
    ...  
    GameRules rules = factory.gameRules();  
    List<Piece> pieces = factory.createDefaultPieces();  
    List<Player> players = new ArrayList<Player>();  
    players.add( factory.createConsolePlayer() );  
    players.add( factory.createRandomPlayer() );  
    Game game = new Game(rules);  
    ConsoleCtrl ctrl = new ConsoleCtrlMVC(game, pieces, players, ...);  
  
    GenericConsoleView view = new GenericConsoleView(...);  
    game.addObserver(view);  
  
    ctrl.start();  
}
```

Todas las instrucciones de salida que están relacionadas con el model se deben borrar del controlador.

Crear una vista ...

Registrar la vista como observador en el juego, para recibir notificaciones cuando ocurra algo “interesante”. La vista debe reaccionar escribiendo lo que quiera en, por ejemplo, la consola.

La Interfaz GameObserver

```
public interface GameObserver {
```

```
    void onGameStart(Board board, String gameDesc, List<Piece> pieces, Piece turn);
```

```
    void onGameOver(Board board, Game.State state, Piece winner);
```

```
    void onMoveStart(Board board, Piece turn);
```

```
    void onMoveEnd(Board board, Piece turn, boolean success);
```

```
    void onChangeTurn(Board board, Piece turn);
```

```
    void onError(String msg);
```

```
}
```

Cuando empieza la partida

Cuando termina la partida

Antes y después de ejecutar un movimiento

Cuando cambia el turno

Cuando algo va mal ...

Lo que hemos visto

- ◆ Piece
- ◆ Board
- ◆ GameRules
- ◆ GameMove
- ◆ Player
- ◆ Controller
- ◆ Model
- ◆ View
- ◆ GameFactory