

Tecnología de la Programación 15-16

Paulina Barthelemy

Práctica 4

Práctica 4

Práctica 4.

Fecha entrega viernes 11 de marzo

Implementar el juego Ataxx

- **Juego de tablero para dos jugadores.**
- **Partiendo de un código donde están implementados los juegos ConnectN, TicTacToe (tres en raya) y Advanced TicTacToe (tres en raya avanzado).**
- **Estructura Modelo-Vista-Controlador (MCV)**

Implementarlo en modo consola

Práctica 4

El patrón modelo-vista-controlador

Es un patrón de arquitectura de las aplicaciones de software. Se utiliza mucho en el desarrollo de aplicaciones visuales y aplicaciones web.

Divide la aplicación en tres componentes:

- **El modelo:** contiene los datos y funcionalidades del dominio de la aplicación.
- **La vista:** es la representación gráfica de la aplicación. Puede ser gráfica o basada en texto.
- **El controlador:** interpreta las acciones del usuario y las traduce en operaciones sobre el modelo. Esta estructura permite que cada componente pueda evolucionar por separado.

En una aplicación existe un solo modelo, pero puede haber una o varias vistas y uno o varios controladores.

Práctica 4

MVC: El modelo

El modelo contiene los datos y funcionalidades (operaciones) que se pueden realizar en la aplicación. Puede recibir consultas sobre su estado.

Puede recibir solicitudes para cambiar su estado (realizar operaciones sobre su estado). El modelo debe ser independiente de la vista y del controlador: no debe ver las clases de los otros dos componentes.

- Esto garantiza que se puede cambiar la vista y el controlador sin afectar al modelo.
- Aun así, debe notificar a las vistas las modificaciones producidas en el modelo. Para ello, se utilizará otro patrón de diseño: el patrón **Observer**. El controlador debe tener acceso a las operaciones sobre el modelo.

Práctica 4

MVC: Las vistas

Son representaciones visuales del modelo. Pueden estar formadas por componentes gráficos, pero no necesariamente.

Las vistas pueden ser una representación parcial del modelo: destacar algunos aspectos y ocultar otros.

Las vistas no deben ver las clases del modelo: es el modelo el que notifica a las vistas los cambios que se producen en su estado.

Las vistas tienen acceso al controlador. Así pueden indicar acciones del usuario que modifiquen el modelo. Es posible tener varias vistas simultáneamente para un modelo. De hecho, las vistas se pueden crear posteriormente sin necesidad de modificar el modelo.

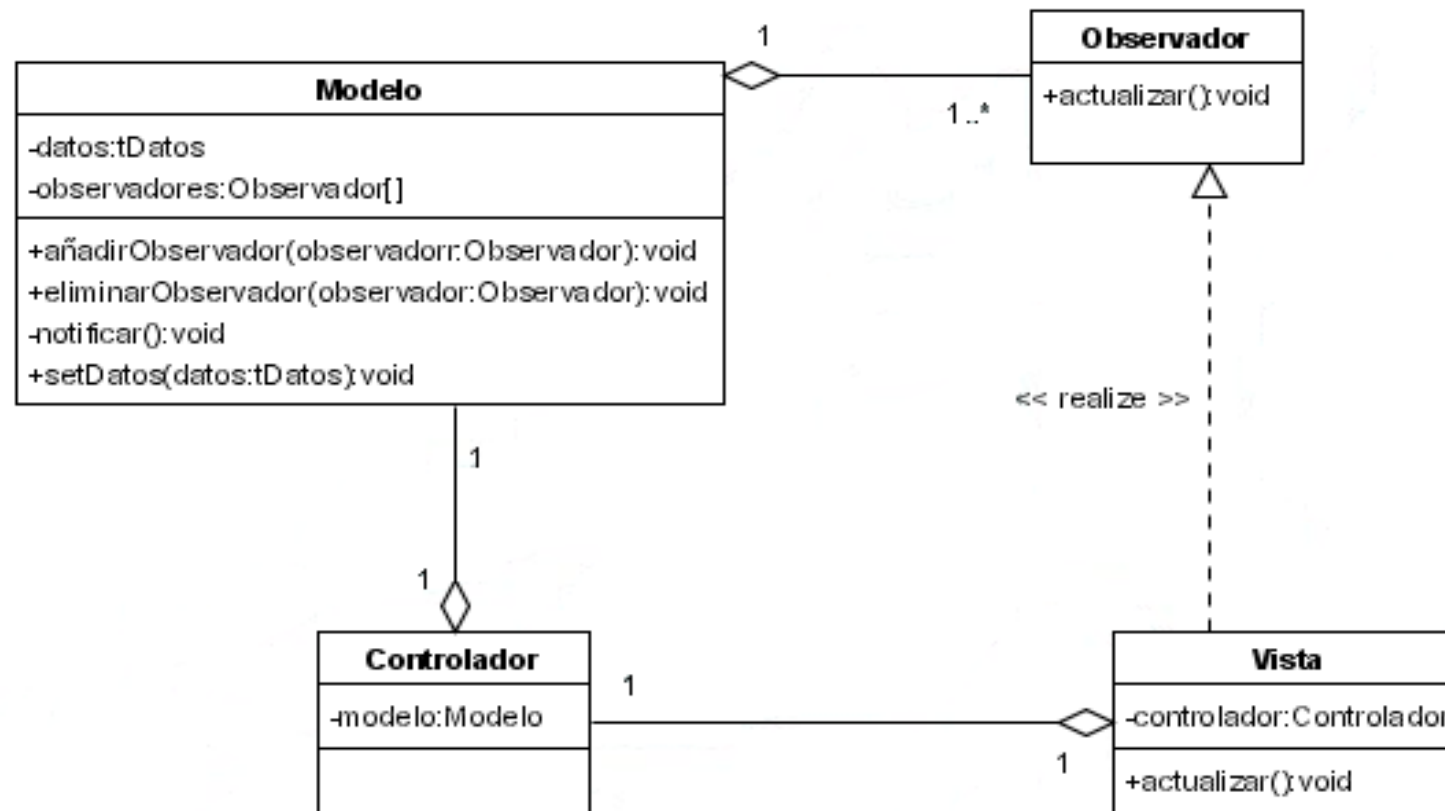
Práctica 4

MVC: El controlador














El controlador determina las modificaciones del modelo de acuerdo a interacciones con la vista. El controlador recibe peticiones de la vista y responde a ellas modificando el modelo.

El controlador ve el modelo y modifica su estado invocando a los métodos necesarios. La relación del controlador con las vistas puede ser diferente según la variante de MVC que se utilice:

Práctica 4



Práctica 4

- ▶  Practica4v2
 - ▶  src
 - ▶  es.ucm.fdi.tp.basecode
 - ▶  es.ucm.fdi.tp.basecode.attd
 - ▶  es.ucm.fdi.tp.basecode.bgame
 - ▶  es.ucm.fdi.tp.basecode.bgame.control
 - ▶  es.ucm.fdi.tp.basecode.bgame.model
 - ▶  es.ucm.fdi.tp.basecode.bgame.views
 - ▶  es.ucm.fdi.tp.basecode.connectN
 - ▶  es.ucm.fdi.tp.basecode.ttt
 - ▶  JRE System Library [JavaSE-1.7]
 - ▶  Referenced Libraries
 - ▶  lib

Práctica 4

Control - Controlador

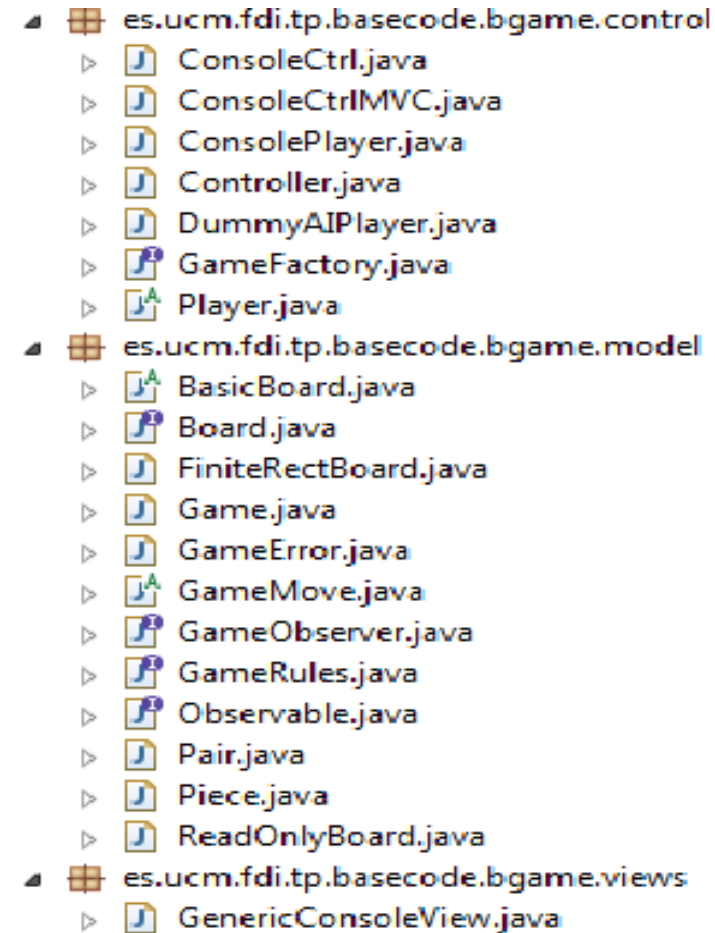
- GameFactory (Interfaz)
- Player (Clase abstracta)

Model - Modelo

- GameRules (Interfaz)
- GameMove (Clase abstracta)
- GameObserver (Interfaz)

Views - Vistas

- GenericConsoleView



Práctica 4

Patrón Factoría

- Define la interfaz de creación de un cierto tipo de objeto, permitiendo que las subclases decidan que clase concreta necesitan instancias.

Muchas veces ocurre que una clase no puede anticipar el tipo de objetos que debe crear, ya que la jerarquía de clases que tiene requiere que deba delegar la responsabilidad a una subclase.

Este patrón debe ser utilizado cuando:

- ✓ Una clase no puede anticipar el tipo de objeto que debe crear y quiere que sus subclases especifiquen dichos objetos..
- Podemos utilizar este patrón cuando definamos una clase a partir de la que se crearán objetos pero sin saber de qué tipo son, siendo otras subclases las encargadas de decidirlo.



Práctica 4

Clases de la práctica

Práctica 4

bgame.model: contiene todas las clases relacionadas con el modelo de

- Las piezas (Piece).
- El tablero (Board).
- Los movimientos del juego (GameMove).
- Las reglas del juego (GameRules).
- La partida (Game).
- Diversos interfaces y clases abstractas.

Práctica 4

La clase Piece

- Esta clase representa una ficha de juego (un contador) que los jugadores pueden utilizar para colocar en el tablero de juego.
- También se puede utilizar para indicar obstáculos, etc. en el tablero.
- Cada ficha tiene un identificador (string) que no debe incluir espacios en blanco.
- Las fichas con el mismo identificador se consideran idénticas.
- Los juegos normalmente necesitan que los identificadores de las fichas sean diferentes.
- Dos fichas con el mismo identificador son iguales y tienen el mismo código hash.

Práctica 4

La interfaz Board

Este interface representa un tablero de un juego.

Permite la siguiente funcionalidad:

- Colocar o eliminar una ficha del tablero, en el que null se utiliza normalmente para representar una posición vacía.
- Preguntar por el valor de una casilla del tablero, el tamaño del tablero, etc.
- Mantener un contador de cada uno de los tipos de chas. Esto normalmente depende del juego, la implementación concreta solo debe proporcionar una forma de asociar un numero (el contador de fichas) con una ficha, modificarlo, etc.
- Hacer una copia del tablero, que es útil para simular lo que ocurriría si se realizara determinado movimiento, pero sin modificar el estado actual del juego.

Práctica 4

La clase abstracta BasicBoard

```
public abstract class BasicBoard implements Board {  
  
    .....  
    @Override  
    public void setPieceCount(Piece p, Integer n) {  
        if (n == null)  
            pieceCount.remove(p);  
        else  
            pieceCount.put(p, n);  
    }  
    @Override  
    public Integer getPieceCount(Piece p) {  
        return pieceCount.get(p);  
    }  
}
```

Práctica 4

La clase abstracta BasicBoard

```
import java. util . HashMap;
```

```
//Tablero abstracto que implementa un mecanismo contador de fichas del interfaz Board usando un HashMap*/
```

```
public abstract class BasicBoard implements Board {
```

```
    private HashMap<Piece, Integer> pieceCount;
```

```
        public BasicBoard() {
```

```
            this . pieceCount = new HashMap<Piece, Integer>();
```

```
        }
```

```
protected void copyTo(BasicBoard board) {
```

```
    board.pieceCount = new HashMap<Piece, Integer>(pieceCount);
```

```
}
```

```
}
```


Práctica 4

La clase FiniteRectBoard

// Implementacion de un tablero rectangular de dimensión finita.

```
public class FiniteRectBoard extends BasicBoard {  
    private Piece[][] board;  
    private int occupied;  
    private int numOfCells;  
    private int cols ; private int rows;  
    public FiniteRectBoard() {}  
    public FiniteRectBoard(int rows, int cols ) {  
        if (rows <= 0 || cols <= 0)  
            throw new GameError("Invalid finite rectangular board size (" + rows + "," + cols + "));  
        this . rows = rows; this . cols = cols;  
        board = new Piece[rows][cols ];    occupied = 0;  
        numOfCells = rows  * cols;  
    }
```

Práctica 4

La clase abstracta GameMove

```
import java.util.List;
```

```
/**Clase abstracta para representar un movimiento de un juego.  
*/
```

```
public abstract class GameMove implements java.io.Serializable {
```

```
/** Ficha (es decir, jugador) al que pertenece esta acción. Se asigna solamente cuando se crea una instancia  
* de esta clase y solo se accede mediante {getPiece()}. */
```

```
private Piece piece ;
```

```
/**Constructor para utilizar con el patrón command de ConsolePlayer
```

```
public GameMove() {}
```

```
/** Constructor abstracto que puede ser llamado por las subclases (y solo por ellas)
```

```
* Devuelve la pieza (el jugador) al que le toca mover en el juego*/
```

```
protected GameMove(Piece piece) {
```

```
this . piece = piece;
```

```
}
```

Práctica 4

La clase ConnectNMove

```
/**  
* Clase para representar un movimiento de los juegos conecta-n.  
*/  
public class ConnectNMove extends GameMove f  
    protected int row;  
    protected int col ;  
    /**  
        Este constructor se usa para obtener una instancia de ConnectNMove para generar movimientos del  
        juego y mostrarlos con String(String)}  
    */  
    public ConnectNMove() {}  
    public ConnectNMove(int row, int col, Piece p) {  
        super(p);  
        this . row = row;  
        this . col = col;  
    }
```

Práctica 4

La interfaz GameRules

Representa las reglas del juego. Tiene las siguientes operaciones

- Proporciona una descripción textual del juego .
- Crea un tablero inicial
- Selecciona un jugador inicial
- Proporciona información acerca del mínimo y el máximo número de jugadores y las reglas permitidas
- Actualiza el estado actual del juego (ganador, yablas...)
- Selecciona el siguiente jugador.
- Genera una lista de movimientos válidos
- Evalúa las jugadas para ganar para jugadores automáticos

Práctica 4

La clase ConnectNRules

Reglas para el juego oConnectN.

- El juego se juega en un tablero $N \times N$
- El número de jugadores es entre 2 y 4
- Los jugadores mueves siguiendo un turno colocando una pieza en una celda vacía
- El ganador es el que forme una línea (horizontal, vertical o diagonal) con N piezas

Práctica 4

La clase TicTacToeRules

```
import es. ucm.fdi. tp. basecode.connectN.ConnectNRules;
public class TicTacToeRules extends ConnectNRules {
    public TicTacToeRules() {
        super(3);
    }
    @Override
    public String gameDesc() {
        return "Tic-Tac-Toe ";
    }
    @Override
    public int maxPlayers() {
        return 2;
    }
}
```

Práctica 4

La clase Game

Clase que representa una partida de juego. Implementa el interface Observable de MVC para notificar a los observadores los cambios en su estado.

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

```
    // Lista de observadores.
```

```
    private ArrayList <GameObserver> observers;
```

```
    //Tablero usado para este juego.
```

```
    private Board board;
```

```
    private Board roBoard;
```

```
    private List <Piece> pieces;
```

```
    private List <Piece> roPieces;
```

```
    private GameRules rules;
```

```
    private Piece turn;
```

```
    private State state ;
```

```
    private Piece winner;
```

Práctica 4

Factoria

Controlador

```
public interface GameFactory {  
    public abstract GameRules gameRules();  
    public abstract Player createConsolePlayer();  
    public abstract Player createRandomPlayer();  
    public abstract Player createAIPlayer();    /// para AI  
    public abstract List<Piece> createDefaultPieces();  
    public void createConsoleView(Observable<GameObserver> g, Controller c);  
    public void createSwingView(Observable<GameObserver> g, Controller c, Piece viewPiece); //para swing  
}
```


Controlador

public interface GameFactory {

```
    public abstract GameRules gameRules();
    public abstract Player createConsolePlayer();
    public abstract Player createRandomPlayer();
    public abstract Player createAIPlayer();    /// para AI
    public abstract List<Piece> createDefaultPieces();
    public void createConsoleView(Observable<GameObserver> g,
                                Controller c);
    public void createSwingView(Observable<GameObserver> g,
                               Controller c, Piece viewPiece); //para swing
```

```
}
```

Modelo

public class AdvancedTTTFactory extends TicTacToeFactory {

```
    @Override
    public GameRules gameRules() {
        return new AdvancedTTTRules();
    }
    @Override
    public Player createConsolePlayer() {
        ArrayList<GameMove> possibleMoves = new ArrayList<GameMove>();
        possibleMoves.add(new AdvancedTTTMove());
        return new ConsolePlayer(new Scanner(System.in), possibleMoves);
    }
    @Override
    public Player createRandomPlayer() {
        return new AdvancedTTTRandomPlayer();
    }
}
```

public class ConnectNFactory implements GameFactory {

```
    @Override
    public GameRules gameRules() {
        return new ConnectNRules(dim);
    }
    @Override
    public Player createConsolePlayer() {
        ArrayList<GameMove> possibleMoves = new ArrayList<GameMove>();
        possibleMoves.add(new ConnectNMove());
        return new ConsolePlayer(new Scanner(System.in), possibleMoves);
    }
    @Override
    public Player createRandomPlayer() {
        return new ConnectNRandomPlayer();
    }
    @Override
    public List<Piece> createDefaultPieces() {
        List<Piece> pieces = new ArrayList<Piece>();
        pieces.add(new Piece("X"));
        pieces.add(new Piece("O"));
        return pieces;
    }
    @Override
    public void createConsoleView(Observable<GameObserver> g, Controller c) {
        new GenericConsoleView(g, c);
    }
    public class TicTacToeFactory extends ConnectNFactory {
        @Override
        public GameRules gameRules() {
            return new TicTacToeRules();
        }
    }
}
```

Práctica 4

main

```
private static GameFactory f;  
private static void parseGameOption(CommandLine line) throws ParseException {  
    String gameVal = line.getOptionValue("g", DEFAULT_GAME.getId());  
    if (gameVal.equals(GameInfo.TicTacToe.getId())) {  
        f = new TicTacToeFactory();  
    } else if (gameVal.equals(GameInfo.AdvancedTicTacToe.getId())) {  
        f = new AdvancedTTTFactory();  
    } else if (gameVal.equals(GameInfo.CONNECTN.getId())) {  
        if (dimRows != null && dimCols != null && dimRows == dimCols) {  
            f = new ConnectNFactory(dimRows);  
        } else {  
            f = new ConnectNFactory();  
        }  
    } else {  
        throw new ParseException("Unknown game " + gameVal + "");  
    }  
public static void startGameNoMVC() {  
    Game g = new Game(f.gameRules());
```

Práctica 4

Observadores

Modelo *public interface **Observable**<T> {*
 public void addObserver(T o);
 public void removeObserver(T o);
}

*public class **Game** implements **Observable**<**GameObserver**> {*
 private ArrayList<GameObserver> observers;

Vista

*public class **GenericConsoleView** implements **GameObserver** {*
 *public **GenericConsoleView**(**Observable**<**GameObserver**> g, **Controller** c) {*
 *g.addObserver(**this**);*
 }

Modelo

```
public interface Observable<T> {
    public void addObserver(T o);
    public void removeObserver(T o);
}

public class Game implements Observable<GameObserver> {
    private ArrayList<GameObserver> observers;
    protected void notifyGameStart() {
        for (GameObserver o : observers) {
            o.onGameStart(roBoard, rules.gameDesc(), roPieces, turn);
        }
    }
    protected void notifyDraw() {...}
    protected void notifyWon() {...}
    protected void notifyStartMove() {...}
    protected void notifyEndMove(boolean success) {...}
    protected void notifyChangeTurn() {...}
    protected void notifyError(RuntimeException e) {...}
    @Override
    public void addObserver(GameObserver o) {
        observers.add(o);
        if (state != State.Starting) {
            notifyGameStart();
        }
    }
    @Override
    public void removeObserver(GameObserver o) {
        observers.remove(o);
    }
}
```

```
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);
}
```

Vista

```
public class GenericConsoleView implements GameObserver {
    public GenericConsoleView(Observable<GameObserver> g,
        Controller c) {

        g.addObserver(this);
    }
    @Override
    public void onGameStart(Board board, String gameDesc, List<Piece> pieces,
        Piece turn) {...}
    @Override
    public void onGameOver(Board board, State state, Piece winner) {...}
    @Override
    public void onMoveStart(Board board, Piece turn) {...}
    @Override
    public void onMoveEnd(Board board, Piece turn, boolean success) {...}
    @Override
    public void onChangeTurn(Board board, Piece turn) {...}
    @Override
    public void onError(String msg) {...}
}
```

Controlador

```
public class Controller {  
    protected Game game;  
    protected List<Piece> pieces;  
    public Controller(Game game, List<Piece> pieces) {  
        this.game = game;  
        if (pieces != null)  
            this.pieces = new ArrayList<Piece>(pieces);  
        else  
            this.pieces = new ArrayList<Piece>();  
    }
```

Vista

```
public class GenericConsoleView implements GameObserver {  
    public GenericConsoleView(Observable<GameObserver> g,  
        Controller c)
```

Main

```
public static void startGame() {  
    Game g = new Game(f.gameRules());  
    Controller c = null;  
  
    switch (view) {  
        case CONSOLE:  
            c = new ConsoleCtrlMVC(g, pieces, players);  
            f.createConsoleView(g, c);  
            break;  
        case WINDOW:  
            c = new Controller(g, pieces);  
            if (multiviews) {  
                for (Piece p : pieces) {  
                    f.createSwingView(g, c, p);  
                }  
            } else {  
                f.createSwingView(g, c, null);  
            }  
            break;  
    }  
  
    public static void main(String[] args) {  
        parseArgs(args);  
        startGame();  
    }  
  
}
```