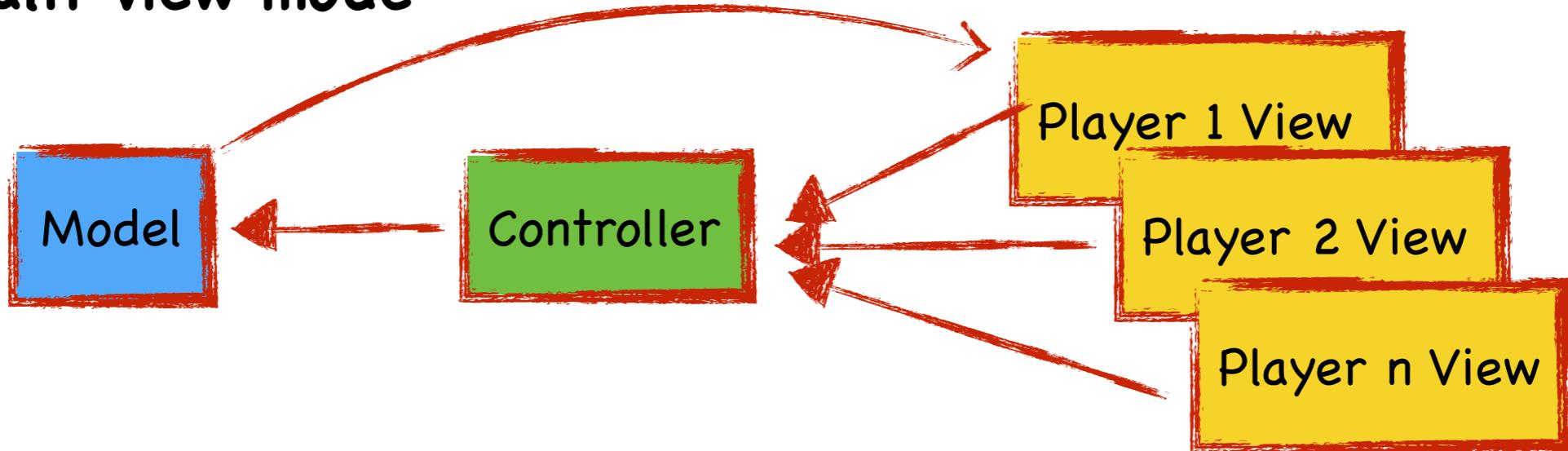


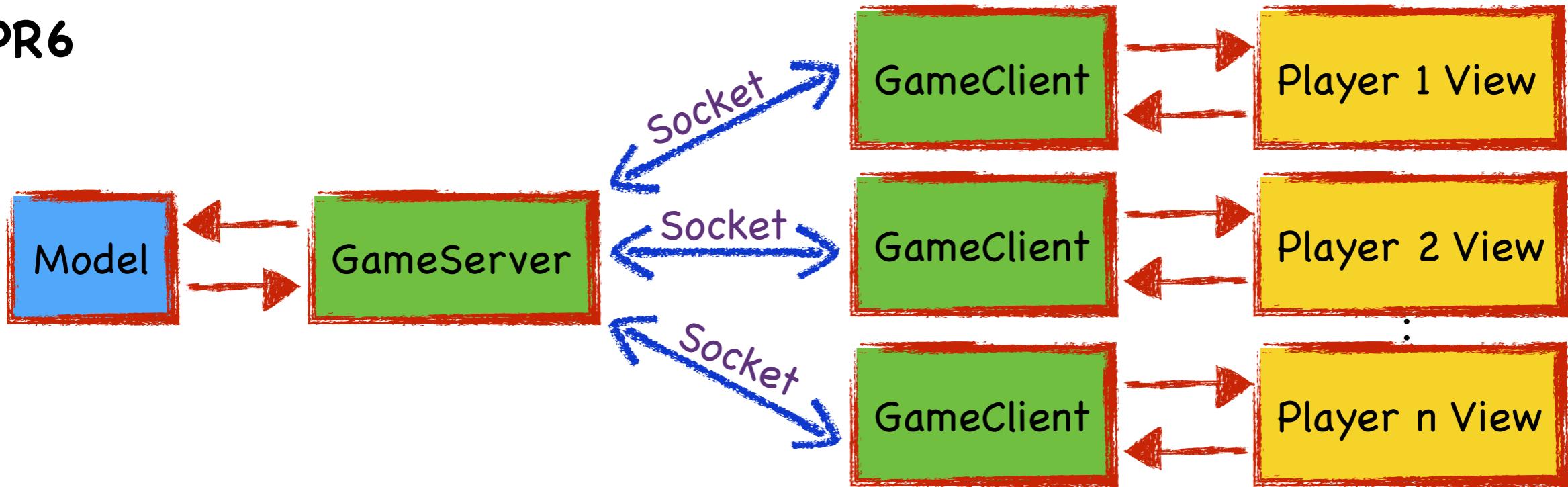
Práctica 6

Descripción General

PR5 in multi-view mode

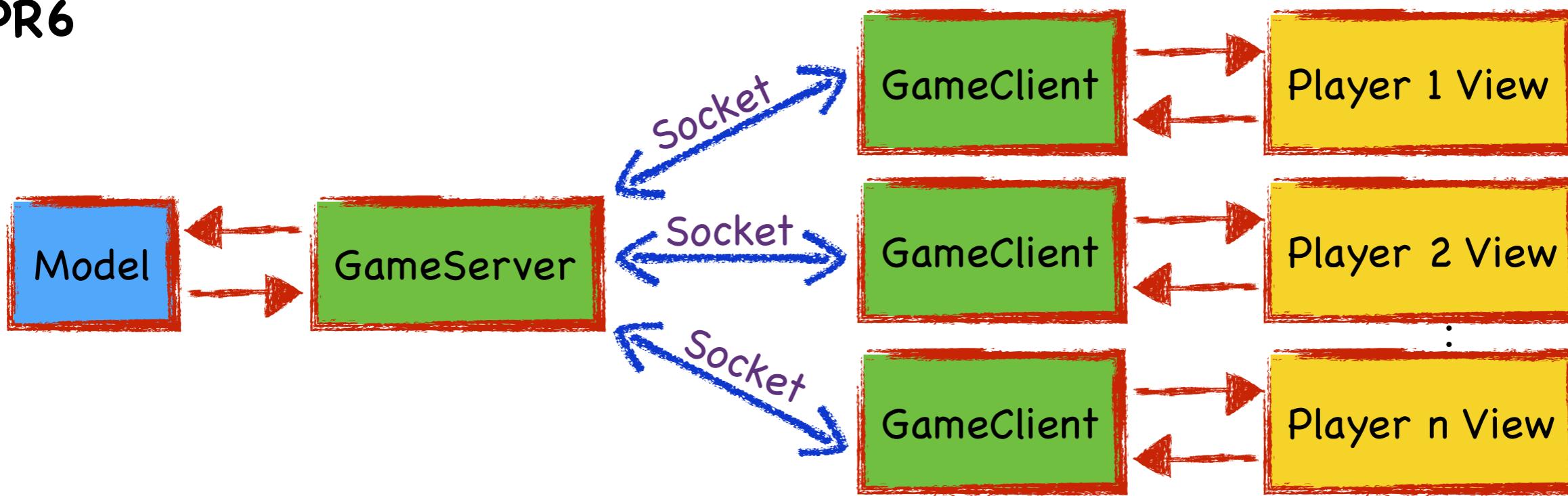


PR6



Descripción General

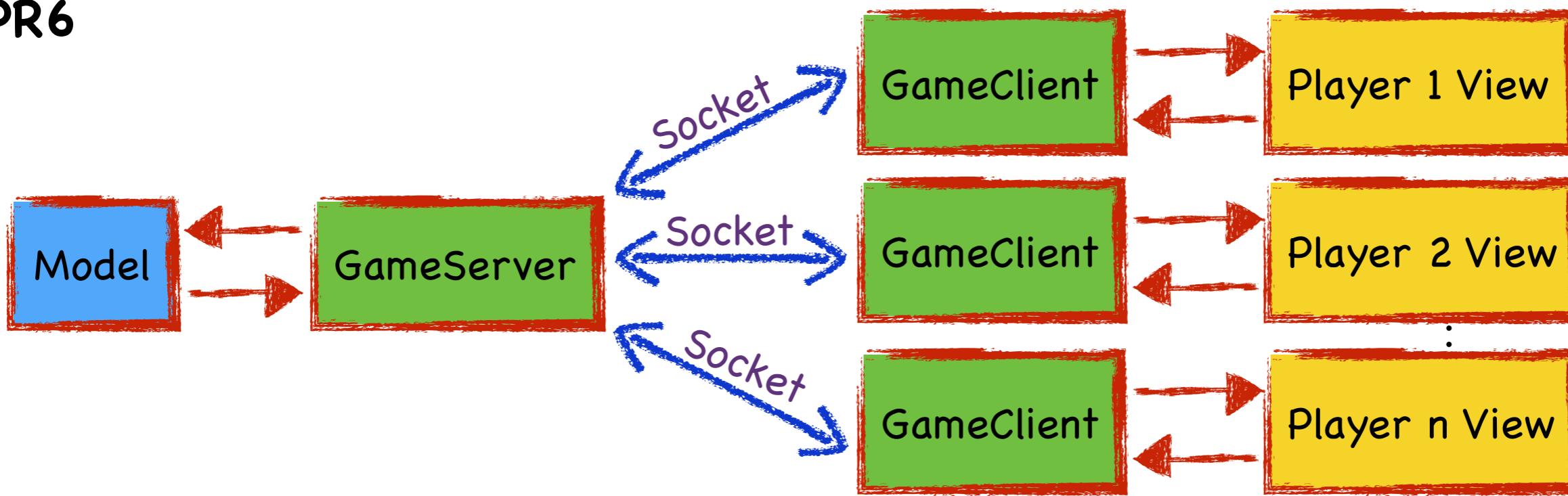
PR6



- ◆ GameServer es un Controller (tiene makeMove, etc.) y también es un GameObserver (tiene onGameStart, onMoveStart, etc.)
- ◆ GameClient es un Controller (tiene makeMove, etc.) y también es un Observable de tipo GameObserver (tiene addObserver, etc.)
- ◆ La vistas ven a GameClient tanto Controller como Observable (Model)
- ◆ Cuando la vista invoca makeMove de su GameClient, el GameClient reenvía la petición a GameServer para ejecutar su makeMove.
- ◆ Cuando GameServer recibe una notificación del modelo, la reenvía a todos los GameClients que la reenvían a sus observadores (las vistas)

Descripción General

PR6



- ◆ Ejecutamos **GameServer** con un juego, p.ej., Ataxx, y una lista de **Piece(s)** (usamos la opción **-p** o la lista de fichas por defecto). Los clientes no saben a qué juego van a jugar.
- ◆ Cuando ejecutamos un **GameClient**, tiene que mandar el String **"Connect"** a **GameServer** para decir que quiere entrar en el juego. Recibe de vuelta un **GameFactory** y un **Piece** que se usan para construir la vista.
- ◆ Una vez hay suficiente clientes conectados, **GameServer** inicia el juego
- ◆ Cuando acaba el juego, **GameServer** desconecta a todos los clientes y espera a otros clientes para contestar a jugar un nuevo juego, etc.

La Clase Main ...

```
public static void main(String[] args) {  
    parseArgs(args);  
    switch (applicationMode) {  
        case NORMAL:  
            startGame();  
            break;  
        case CLIENT:  
            startClient();  
            break;  
        case SERVER:  
            startServer();  
            break;  
    }  
}
```

El mismo main se usa para el cliente, el servidor y el modo de la PR5, depende de los parámetros de línea de comandos. Se tiene que añadir soporte para analizar las nuevas opciones de línea de comandos

```
private static void startServer() {  
    GameServer c = new GameServer(gameFactory, pieces, serverPort);  
    c.start();  
}
```

```
private static void startClient() {  
    try {  
        GameClient c = new GameClient(serverHost, serverPort);  
        gameFactory = c.getGameFactory();  
        gameFactory.createSwingView(c, c, c.getPlayerPiece(), ...);  
        c.start();  
    } catch (Exception e) {  
        System.err.println(...);  
    }  
}
```

Dependiendo del valor --application-mode, invocamos a startGame para usar el modo normal (de la PR5), a startClient para usar el modo cliente, o startServer para usar el modo servidor .

El valor de la opción --server-port

Los valores de --server-host y --server-port

The view sees 'c' as
a controller and as a
model

Usamos la ficha asignada al cliente
(por el servidor) para crear la vista

Usamos el GameFactory
enviado por el servidor
para crear la vista. El
cliente no sabe a qué
juego va a jugar.

La Clase Connection

```
public class Connection {
```

```
...
```

```
public Connection(Socket s) throws ... {
```

```
    this.s = s;
```

```
    this.out = new ObjectOutputStream( s.getOutputStream() );
```

```
    this.in = new ObjectInputStream( s.getInputStream() );
```

```
}
```

```
public void sendObject(Object r) throws ... {
```

```
    out.writeObject(r);
```

```
    out.flush();
```

```
    out.reset();
```

```
}
```

```
public Object getObject() throws ... {
```

```
    return in.readObject();
```

```
}
```

```
public void stop() throws ... {
```

```
    s.close();
```

```
}
```

Para simplificar el uso de sockets, definimos una clase Connection que nos permite enviar y recibir objetos usando un socket dado ...

Almacenar los streams de entrada y salida en atributos.

Enviar un objeto

Recibir un objeto

Cerrar el socket

GameServer

La Constructora de GameServer

```
public class GameServer extends Controller implements GameObserver {  
    ...  
    private int port;                                El puerto usado por el servidor  
    private int numPlayers;                            El número de jugadores necesario para iniciar el juego  
    private int numOfConnectedPlayers;                El número de jugadores conectados. Se usa para saber cuándo hay que iniciar el juego  
    private GameFactory gameFactory;                  El GameFactory que se usa para crear GameRules  
    private List<Connection> clients;                 Lista de clientes conectados  
    volatile private ServerSocket server;              Una referencia al servidor  
    volatile private boolean stopped;                 Indica si el servidor ha sido "apagado"  
    volatile private boolean gameOver;                Indica si el juego ha terminado  
    ...  
    public GameServer(GameFactory gameFactory, List<Piece> pieces, int port) {  
        super(new Game(gameFactory.gameRules()), pieces);  
        // initialise the fields with corresponding values  
        ...  
        game.addObserver(this);  
    }  
    ...  
    ... Registrar el servicio como observador en el juego para recibir notificaciones  
}
```

El código es una clase Java llamada GameServer que extiende la clase Controller y implementa la interfaz GameObserver. La clase tiene los siguientes atributos:

- port: El puerto usado por el servidor.
- numPlayers: El número de jugadores necesario para iniciar el juego.
- numOfConnectedPlayers: El número de jugadores conectados. Se usa para saber cuándo hay que iniciar el juego.
- gameFactory: El GameFactory que se usa para crear GameRules.
- clients: Lista de clientes conectados.
- server: Una referencia al servidor.
- stopped: Indica si el servidor ha sido "apagado".
- gameOver: Indica si el juego ha terminado.

La clase tiene un constructor que toma un GameFactory, una lista de Piezas y un puerto como argumentos. Llama al constructor de Game y registra el servicio como observador del juego.

Los Métodos de Controller

```
@Override
```

```
public synchronized void makeMove(Player player) {  
    try { super.makeMove(player); } catch (GameError e) { }  
}
```

```
@Override
```

```
public synchronized void stop() {  
    try { super.stop(player); } catch (GameError e) { }  
}
```

```
@Override
```

```
public synchronized void restart() {  
    try { super.restart(player); } catch (GameError e) { }  
}
```

```
@Override
```

```
public void start() {  
    controlGUI();  
    startServer();  
}
```

Estos métodos ya están definidos en la super clase Controller. Los sobreescrivimos aquí para poder capturar excepciones — podemos perder la conexión si se lanza un excepción ...

Al ejecutar el controlador GameServer, construir de la GUI de control para poder mostrar mensajes y parar el servidor, etc., y luego iniciar el servidor.

Terminología ...

- ◆ “parar el juego” significa
 1. invocar stop() si el juego está en estado InPlay
 2. cambiar ‘gameOver’ a true
 3. desconectar todos los clientes – invocar c.stop()
para cada cliente ‘c’ (instancia de Connection)
- ◆ “parar el servidor” significa
 1. cambiar ‘stopped’ to true
 2. para el juego (como arriba)
 3. apagar el servidor, es decir llamar a server.close()

La GUI de Control

```
private void controlGUI() {  
    try {  
        SwingUtilities.invokeAndWait(new Runnable() {  
            @Override  
            public void run() { constructGUI(); }  
        });  
    } catch (InvocationTargetException | InterruptedException e) {  
        throw new GameError("Something went wrong when constructing the GUI");  
    }  
}
```

```
private void constructGUI() {  
    JFrame window = new JFrame("Game Server");  
    // create text area for printing messages  
    infoArea = ...  
    // quit button  
    JButton quitButton = new JButton("Stop Sever");  
    ...  
    window.setPreferredSize( ... );  
    window.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);  
    window.pack();  
    window.setVisible(true);  
}
```

```
private void log(String msg) {  
    // show the message in infoArea, use invokeLater!!  
}
```

Usar invokeAndWait en lugar de invokeLater, así al salir de controlGUI sabemos que es seguro usar el método 'log' para añadir mensajes, etc.

Cuando se hace clic en este botón, tenemos que parar el servidor y salir de la aplicación.

Usar este método desde todas partes del GameServer para añadir mensajes a infoArea

El Bucle Principal del Servidor

```
private void startServer() {  
    server = new ServerSocket(port);  
    stopped = false;
```

Iniciar el servicio (necesitas capturar excepciones si es necesario)

Indicar que el juego no ha terminado, ya que estamos a punto de empezar

```
while (!stopped) {  
    try {  
        // 1. accept a connection into a socket s  
        // 2. log a corresponding message  
        // 3. call handleRequest(s) to handle the request  
    } catch (IOException e) {  
        if (!stopped) {  
            log("error while waiting for a connection: " + e.getMessage());  
        }  
    }  
}
```

El bucle del servidor: esperar a que un cliente conecta, y pasar el socket correspondiente a handleRequest para responder a la petición.

Si la ejecución entra en esta sección de catch, puede ser porque el servidor ha sido parado, en ese caso 'stopped' sería 'true' y salimos del bucle en la siguiente iteración. Si 'stopped' es 'false' entonces ha ocurrido alguna error, así que mostramos un mensaje adecuado ...

Manejar Peticiones de Clientes

```
private void handleRequest(Socket s) {  
    try {  
        Connection c = new Connection(s);  
  
        Object clientRequest = c.getObject();  
        if ( !(clientRequest instanceof String) &&  
            !((String) clientRequest).equalsIgnoreCase("Connect") ) {  
            c.sendObject(new GameError("Invalid Request"));  
            c.stop();  
            return;  
        }  
        // 1. ...  
        ...  
        // 2.  
        numOfConnectedPlayers++;  
        clients.add(c);  
        // 3. ...  
        ...  
        // 4. ...  
        ...  
        // 5.  
        startClientListener(c);  
    } catch (IOException | ClassNotFoundException _e) {}  
}
```

Envolver el socket con Connection, para que sea más fácil de usarlo

El primer mensaje del cliente tiene que ser el String "Connect"

Si el número de clientes conectados ya ha alcanzado el máximo, respondemos con un GameError adecuado

Incrementar el número de clientes conectados y añadir 'c' a la lista de clientes

Enviar el String "OK" al cliente, seguido por el GameFactory y el Piece a usar. asignamos al i-esimo cliente la i-esima ficha (de la lista 'pieces')

Si hay un numero suficiente de clientes, iniciar el juego (la primera vez usando start, después usando restart)

Invocar a startClientListener para iniciar una hebra para recibir comandos del cliente.

Recibir Comandos del Cliente

```
private void startClientListener(Connection c) {  
    gameOver = false; ← Poner gameOver a 'false' para indicar  
    Thread t = ... ← que el juego no ha terminado.  
    ...  
    t.start();  
    ...  
}  
  
while (!stopped && !gameOver) {  
    try {  
        Command cmd;  
        // 1. read a Command  
        // 2. execute the command ← Recibir un Object del cliente y  
        ← hacer casting a Command ← Ejecutar cmd.exec pasándole  
        ← el Controller GameServer.this,  
        ← exec llamará a makeMove,  
        ← stop, etc.  
    } catch (ClassNotFoundException | IOException e) {  
        if (!stopped && !gameOver) {  
            // stop the game (not the server)  
        }  
    }  
}  
  
Es importante que  
la hebra termina  
cuando el juego  
ha terminado o el  
servidor ha sido  
parado.  
  
Cuando termina el juego o el servidor ha sido parado, se  
cierran las conexiones con los clientes, así que si estamos  
esperando a leer un comando del cliente si lanza un  
excepción, en ese caso no hacemos nada. Si entramos en  
el catch por otro razón tenemos que parar el juego  
porque ha ocurrido un error con algún cliente. En ambos  
casos vamos a salir del bucle (y la hebra termina)
```

Enviar Notificaciones a Clientes

```
void onGameStart(Board board, String gameDesc, List<Piece> pieces, Piece turn) {  
    forwardNotification(new GameStartResponse(board, gameDesc, pieces, turn));  
}
```

```
void forwardNotification(Response r) {  
    // call c.sendObject(r) for each client connection 'c'  
}
```

```
public interface Response extends java.io.Serializable {  
    public void run(GameObserver o);  
}
```

```
public class GameStartResponse implements Response {  
    ...  
    public GameStartResponse(Board board, ...) {  
        this.board = board;  
        this.gameDesc = gameDesc;  
        this.pieces = pieces;  
        this.turn = turn;  
    }  
  
    @Override  
    public void run(GameObserver o) {  
        o.onGameStart(board, gameDesc, pieces, turn);  
    }  
}
```

Cuando el GameServer recibe una notificación del modelo, crea un objeto 'Response' que representa la notificación y lo envía a todos los clientes.

Cuando el GameClient recibe un 'Response' r, lo que va a hacer es ejecutar r.run(o) para todo observador 'o', en particular la vista sería un observador así que 'run' llamará al su método correspondiente. Lo vamos a ver más adelante.

Enviar Notificaciones a Clientes

Implementar 'Response' adecuado para cada metodo de GameObserver

```
void onGameOver(Board board, State state, Piece winner) {  
    forwardNotification(new GameOverResponse( ... ));  
    // stop the game  
}  
  
void onMoveStart(Board board, Piece turn) {  
    forwardNotification(new MoveStartResponse( ... ));  
}  
  
void onMoveEnd(Board board, Piece turn, boolean success) {  
    forwardNotification(new MoveEndResponse( ... ));  
}  
  
public void onChangeTurn(Board board, Piece turn) {  
    forwardNotification(new ChangeTurnResponse( ... ));  
}  
  
public void onError(String msg) {  
    forwardNotification(new ErrorResponse( ... ));  
}
```



Aparte de reenviar la notificación, paramos el juego en el caso de la notificación onGameOver

GameClient

The Constructor of GameClient

Es un Controller para que la vista use makeMove, etc. Es un Observable para que la vista se registre para recibir notificaciones del modelo – dichas notificaciones vienen del servidor. La vista usa GameClient como “modelo” y como “controlador”, no sabe que las notificaciones vienen del servidor o que las llamadas a makeMove van a servidor.

```
public class GameClient extends Controller implements Observable<GameObserver> {
```

```
    private String host;
```

```
    private int port;
```

```
    private List<GameObserver> observers;
```

```
    private Piece localPiece;
```

```
    private GameFactory gameFactory;
```

```
    private Connection connectToServer;
```

```
    private boolean gameOver;
```

```
...
```

```
    public GameClient(String host, int port) throws Exception {
```

```
        super(null, null);
```

```
        // initialise the fields
```

```
        connect();
```

```
}
```

```
    public GameFactory getGameFactoty() { ... }
```

```
    public Piece getPlayerPiece() { ... }
```

```
    public void addObserver(GameObserver o) { ... }
```

```
    public void removeObserver(GameObserver o) { ... }
```

```
...
```

Las notificaciones que manda el servidor se reenvían a todos los observadores

Piece y GameFactory que tiene que usar el cliente (viene del servidor)

La connexion al servidor

Indica si el juego ha terminado

Inicializar los atributos y llama a connect para establecer la conexión con el servidor

Consultar los valores de localPiece y gameFactory

La vistas usan estos methods para registrarse a recibir notificaciones

Llamadas Remotas a GameServer

```
@Override  
public void makeMove(Player p) {  
    forwardCommand(new PlayCommand(p));  
}
```

```
@Override  
public void stop() {  
    forwardCommand(new QuitCommand());  
}
```

```
@Override  
public void restart() {  
    forwardCommand(new RestartCommand());  
}
```

```
private void forwardCommand(Command cmd) {  
    // if the game is over do nothing, otherwise  
    // send the object cmd to the server  
}
```

Cuando la vista llama a los método makeMove del GameClient (makeMove, stop, etc.), queremos hace una llamada remota al método correspondiente del GameServer.

Implementamos dichas llamadas remotas usando clases que implementan la interfaz Command – ver el paquete:
`basecode.bgame.control.commands`

Cuando el GameServer recibe un objeto cmd de tipo Command, invocará a su execute pasándole sí mismo, es decir `cmd.execute(GameSerever.this)`. El método execute a su vez llamará al método correspondiente de `GameSerever.this` (p.ej., `makeMove`).

Conectar al Servidor

```
private void connect() throws Exception {  
    connectionToServer = new Connection(new Socket(host, port));  
    // 1. ...  
    ...  
    Object response = ...  
    if (response instanceof Exception) {  
        throw (Exception) response;  
    }  
  
    try {  
        gameFactory = ...  
        localPiece = ...  
    } catch (Exception e) {  
        throw new GameError("Unknown server response: "+e.getMessage());  
    }  
}
```

Crear una conexión con el servidor

Enviar el String "Connect" para expresar su interés en jugar

Leer el primer objeto en la respuesta del servidor. Si es una instancia de Exception entonces la lanzamos porque eso significa que el servidor ha rechazado la petición.

Si no es instancia de Exception, sería el String "OK" seguido por GameFactory y Piece que el cliente tiene que usar ...

El Bucle Principal de GameClient

```
public void start() {  
    this.observers.add( ... );  
    gameOver = false;  
    while (!gameOver) {  
        try {  
            Response res = ... // read a response  
            for (GameObserver o : observers) {  
                // execute the response on the observer o  
            }  
        } catch (ClassNotFoundException | IOException e) {  
        }  
    }  
    ...  
}
```

Crear una instancia Anónima de GameObserver y regístrala como observador en GameCient. Esta instancia tiene que cambiar gameOver a 'true' y cerrar la conexión con el servidor en su método onGameOver — nada más. Así podemos salir del bucle principal cuando termina el juego.

Mientras que el juego no ha terminado, leer un "Response" enviado por el servidor y ejecuta res.run(o) para cada observador para pasarle la notification correspondiente.

¡Importante!

Modificar la pr5 para que use `invokeAndWait` en lugar de `invokeLater` en `create SwingView`. Así no aseguramos que la vista ya se ha registrado como observador en `GameClient` antes de llamar a `c.start()`. Si no, puede ser que la vista no reciba la notificación `onGameStart`.

```
public void create SwingView(final Observable<GameObserver> g, ...) {  
    try {  
        SwingUtilities.invokeAndWait(new Runnable() {  
            @Override  
            public void run() {  
                new ConnectNSwingView(g, c, viewPiece, random, auto);  
            }  
        });  
    } catch (InvocationTargetException | InterruptedException e) {  
        throw new GameError("...");  
    }  
}
```

```
GameClient c = new GameClient(serverHost, serverPort);  
gameFactory = c.getGameFactoty();  
gameFactory.create SwingView(c, c, c.getPlayerPiece(), ...);  
c.start();
```

Si en tu implementación hay otros componentes de la vista que registran como observadores del modelo (p.ej., `BoardComponent`), no lo hagas con `invokeLater` o `invokeAndWait` sino llamar directamente a `addObserver`

Ejecutar la Aplicación

- ◆ Puedes probar la aplicación ejecutando el servidor y los clientes en el mismo ordenador (recuerda que el valor por defecto de la opción `--server-host` es “`localhost`”)
- ◆ Recuerda que tienes que ejecutar `Main.java` varias veces: una vez para el servidor y otras para cada cliente.
- ◆ Si usas WiFi dentro la universidad, puede ser que no sea posible ejecutar la aplicación en varios ordenadores porque casi todos los puertos están bloqueados.

Un Main Auxiliar

Para simplificar la forma de ejecutar la aplicación, en lugar de cambiar los parámetros de línea de comando cada vez, podemos crear clases auxiliares para llamar a Main.main de la siguiente manera :

```
public class Test1 {  
    public static void main(String args) {  
        String[] args = { "-am", "client", "-aialg", "minmax", "-md", "5" };  
        Main.main(args);  
    }  
}
```

```
public class Test2 {  
    public static void main(String args) {  
        String[] args = { "-am", "server", "-g", "ataxx", "-p", "X,O" };  
        Main.main(args);  
    }  
}
```

Crear un “Runnable JAR”

- ◆ Se puede crear un archivo jar de la aplicación y ejecutarlo en varios ordenadores desde un terminal usando “`java -jar pr6.jar -am servidor ...`” (la versión de Java tiene que ser compatible).
- ◆ Para crear un “Runnable JAR” en Eclipse
 1. Abrir “Export” -> “Runnable Jar File”
 2. Seleccionar el Main desde el “launch configuration”
 3. Indicar el nombre que quieras dar al archivo jar
 4. Seleccionar “package required libraries into JAR...”
 5. Hacer clic sobre el botón Finish