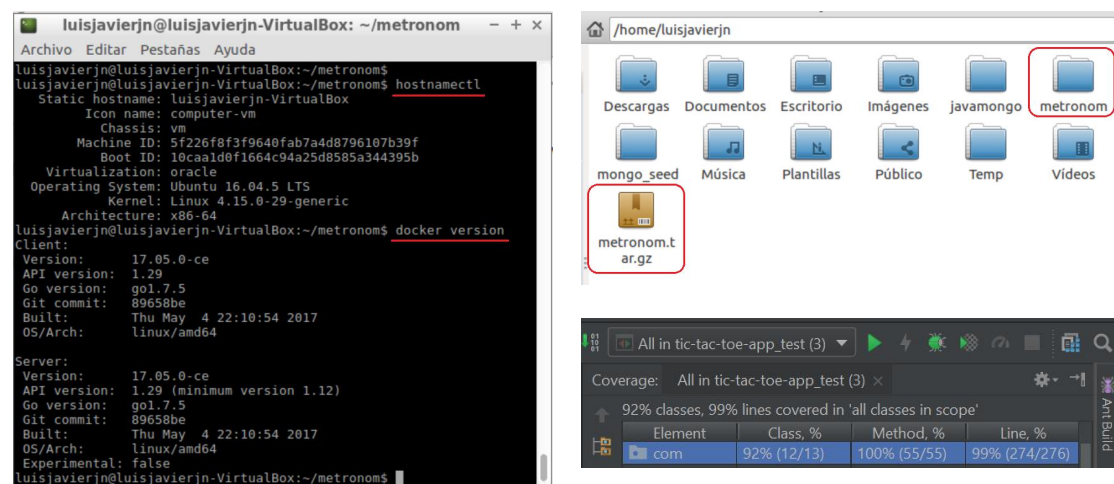


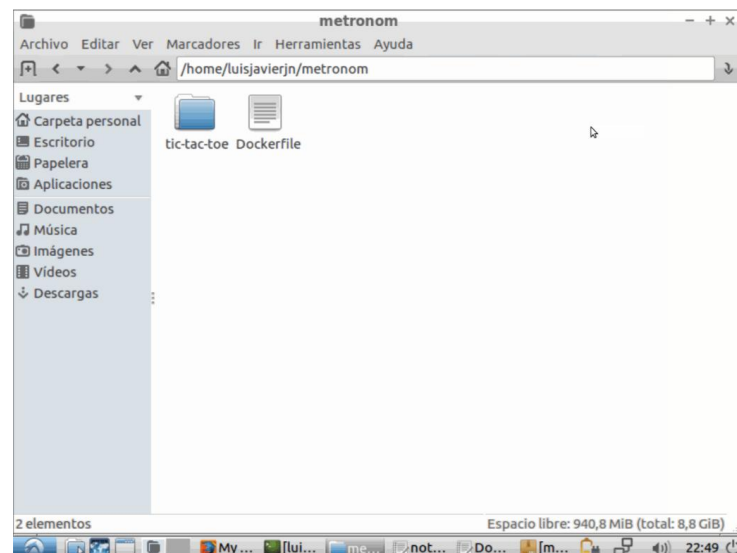
How to run the project

First at all, the source code was tested in Ubuntu, and along with this documentation you're going to find a compressed file called metronom.tar.gz (and metronom.zip, it's the same) where you are going to get the project folder called metronom as well. Besides, Ubuntu has docker installed and the lines coverage of the project out of junit test is 99%. Finally, the IDE used to develop was IntelliJ IDEA 2018.1 (Community Edition).

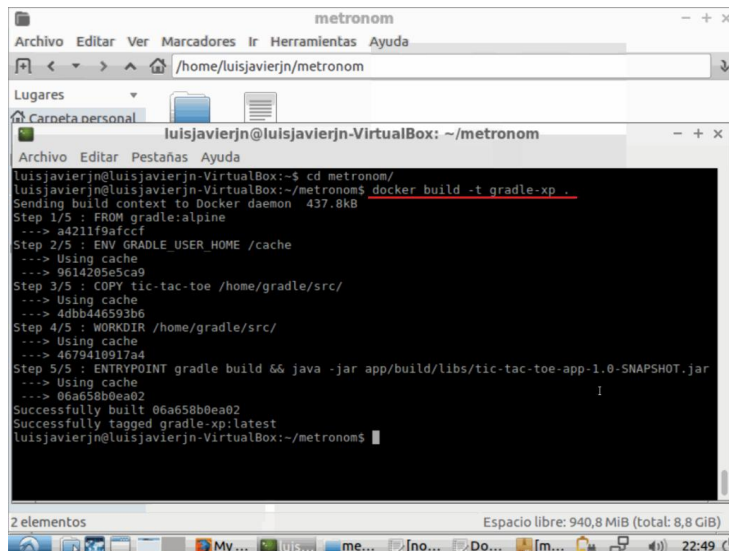


Happy Path

To see the whole steps together there is a file called happy-path.gif where you can see the entire procedure to get the application running right away. Nevertheless, in the following images you can see the same steps to run the game:



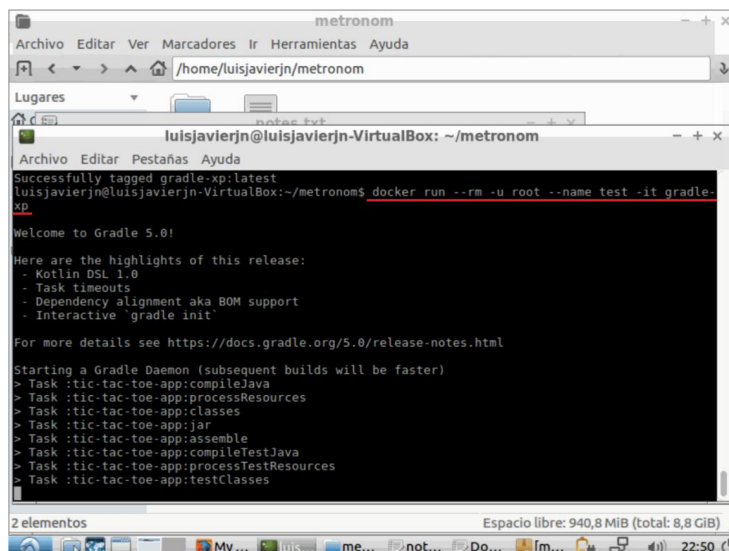
Once we have decompress the metronom.tar.gz file we're going to have a metronom directory, we get in and see that we have a tic-tac-toe folder and a Dockerfile file



```
luisjavierjn@luisjavierjn-VirtualBox: ~/metronom
luisjavierjn@luisjavierjn-VirtualBox:~$ cd metronom/
luisjavierjn@luisjavierjn-VirtualBox:~/metronom$ docker build -t gradle-xp .
Sending build context to Docker daemon 437.8kB
Step 1/5 : FROM gradle:alpine
--> a4211f9a1cf
Step 2/5 : ENV GRADLE_USER_HOME /cache
--> Using cache
--> 9614205e5ca9
Step 3/5 : COPY tic-tac-toe /home/gradle/src/
--> Using cache
--> 4abb446593b6
Step 4/5 : WORKDIR /home/gradle/src/
--> Using cache
--> 4679410917a4
Step 5/5 : ENTRYPOINT gradle build && java -jar app/build/libs/tic-tac-toe-app-1.0-SNAPSHOT.jar
--> Using cache
--> 06a658b0ea02
Successfully built 06a658b0ea02
Successfully tagged gradle-xp:latest
luisjavierjn@luisjavierjn-VirtualBox:~/metronom$
```

Then in the console log we type the following docker command: `docker build -t gradle-xp .`

Obviously we can change `gradle-xp` for whatever other name that we want for the image



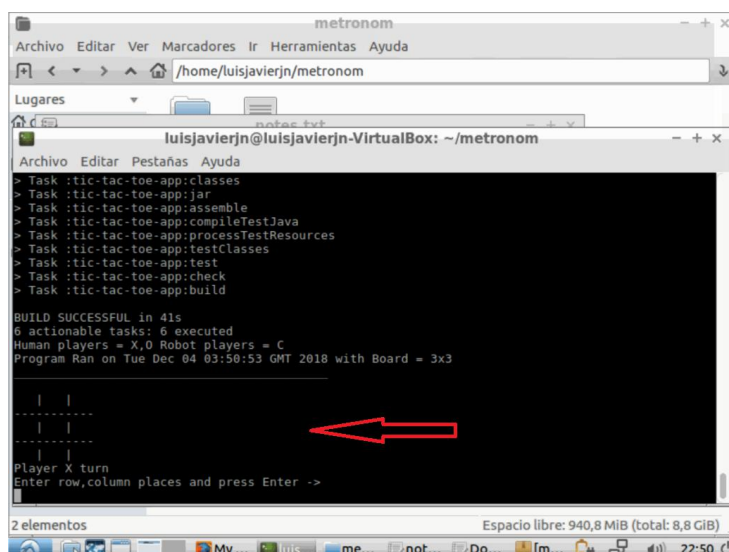
```
luisjavierjn@luisjavierjn-VirtualBox:~/metronom$ docker run --rm -u root --name test -it gradle-xp
Welcome to Gradle 5.0!

Here are the highlights of this release:
- Kotlin DSL 1.0
- Task timeouts
- Dependency alignment aka BOM support
- Interactive 'gradle init'

For more details see https://docs.gradle.org/5.0/release-notes.html

Starting a Gradle Daemon (subsequent builds will be faster)
> Task :tic-tac-toe-app:compileJava
> Task :tic-tac-toe-app:processResources
> Task :tic-tac-toe-app:classes
> Task :tic-tac-toe-app:jar
> Task :tic-tac-toe-app:assemble
> Task :tic-tac-toe-app:compileTestJava
> Task :tic-tac-toe-app:processTestResources
> Task :tic-tac-toe-app:testClasses
```

Then we run an interactive container called `test` out of `gradle-xp`



```
> Task :tic-tac-toe-app:classes
> Task :tic-tac-toe-app:jar
> Task :tic-tac-toe-app:assemble
> Task :tic-tac-toe-app:compileTestJava
> Task :tic-tac-toe-app:processTestResources
> Task :tic-tac-toe-app:testClasses
> Task :tic-tac-toe-app:test
> Task :tic-tac-toe-app:check
> Task :tic-tac-toe-app:build

BUILD SUCCESSFUL in 41s
6 actionable tasks: 6 executed
Human players = X,0 Robot players = C
Program Ran on Tue Dec 04 03:50:53 GMT 2018 with Board = 3x3

  | | 
--|--
  | | 
--|--
  | | 
  | | 
Player X turn
Enter row,column places and press Enter ->
```

Once all test passed, the `ENTRYPOINT` makes the game show up, and we see that it is the X turn

```
metronom
Archivo Editar Ver Marcadores Ir Herramientas Ayuda
/home/luisjavierjn/metronom

luisjavierjn@luisjavierjn-VirtualBox: ~/metronom
BUILD SUCCESSFUL in 41s
6 actionable tasks: 6 executed
Human players = X,0 Robot players = C
Program Ran on Tue Dec 04 03:50:53 GMT 2018 with Board = 3x3

| | |
| | |
| | |
-----
| | |
| | |
| | |
-----
Player X turn
Enter row,column places and press Enter ->
1,1

X | |
| | |
| | |
-----
| | |
| | |
| | |
-----
Player 0 turn
Enter row,column places and press Enter ->
```

After the X player makes his move to the row 1 and column 1 it is the O turn

```
metronom
Archivo Editar Ver Marcadores Ir Herramientas Ayuda
/home/luisjavierjn/metronom

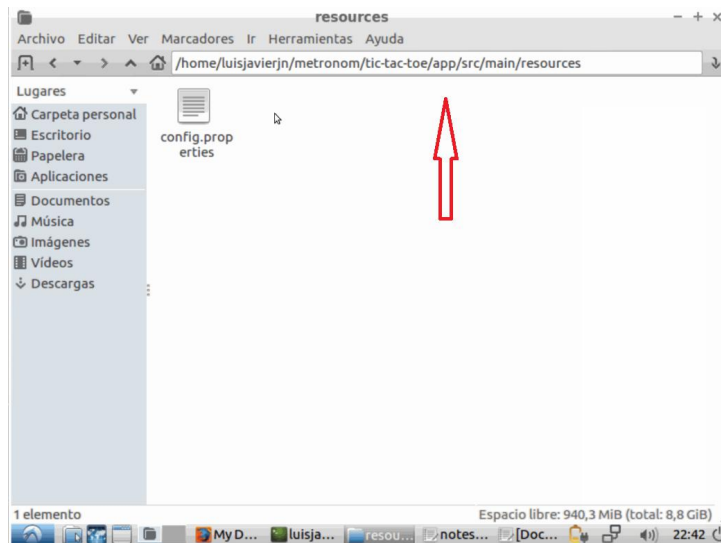
luisjavierjn@luisjavierjn-VirtualBox: ~/metronom
X | |
Now is coming the computer move...
-----
X | C | O
| | O
-----
X | | C
Player X turn
Enter row,column places and press Enter ->
2,1

X | C | O
X | | O
-----
X | | C
Player X wins!!!!!!
Press 0 and Enter to exit or just Enter to reset
0
Game Over
luisjavierjn@luisjavierjn-VirtualBox:~/metronom$
```

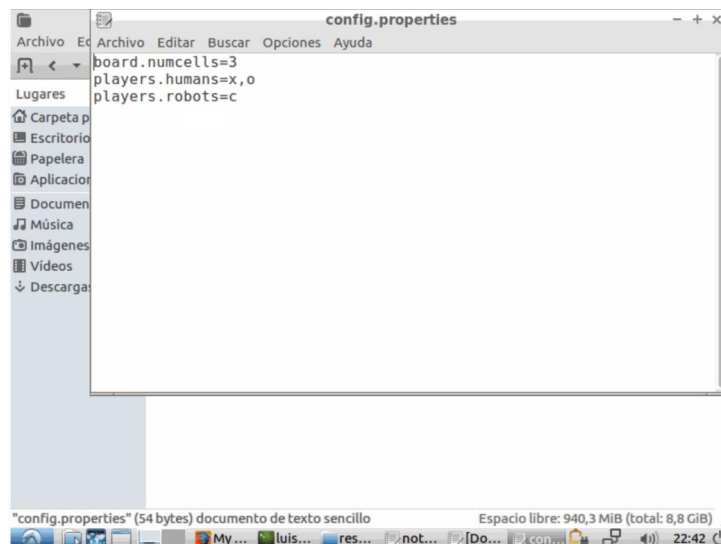
In the config.properties, by default the game is set up for a 3x3 board, 2 human players and 1 robot. So, once O end his move it is the C turn, that's to say, the Computer turn, and this is something that happen automatically showing us the message "Now is coming the computer move", after that, the game get back to the X player. We also can see in the screenshot that the program detects that someone won and it ask to the user if he/she wants to exit or reset the game

Now let's see how we can run the game with different parameters so we can change the behaviour and shape of the game. In the following images you are going to see how to configure the game for a board 4x4 and 2 robots so they play one against the other. There is also a file called board-4x4-two-robots.gif where you can see the whole procedure to do that so. Nevertheless, in the following images you can see the same steps to run the game this way:

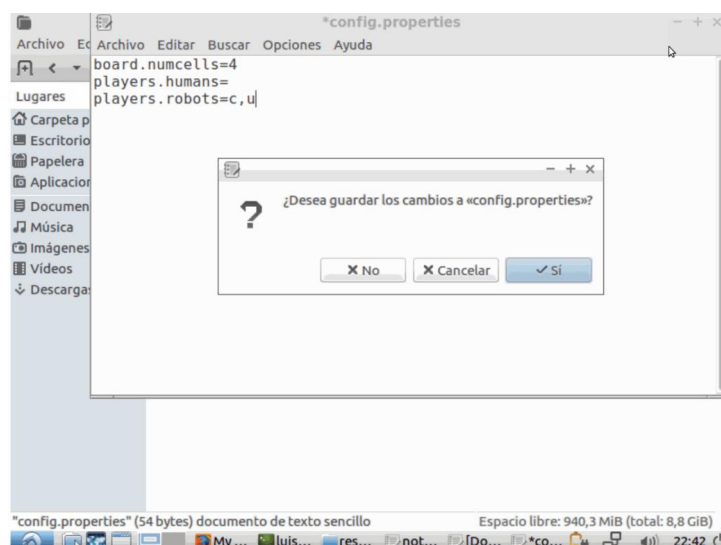
Changing Parameters



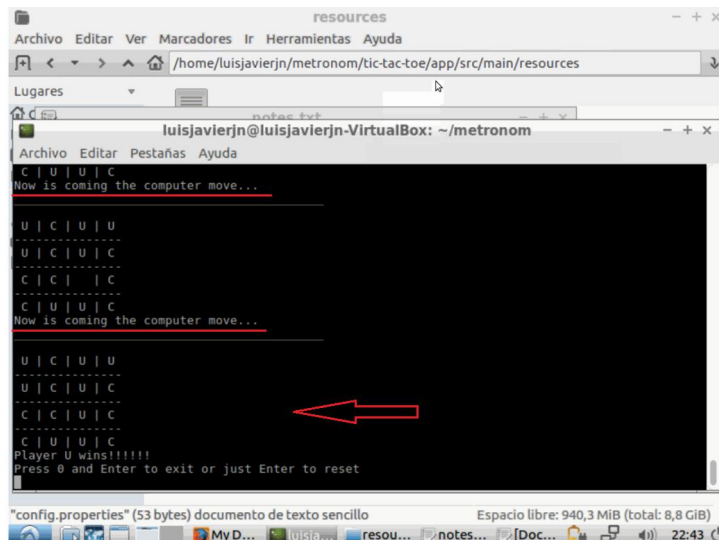
If we want to change the shape of the board and the number of players we go to the file `config.properties` located in `tic-tac-toe/app/src/main/resources` and we proceed to edit it



We open the file and change the default values which are a 3x3 board, 2 human players and 1 robot. We can have as many players as we want but at least one, and we can have from a 3x3 board to a 10x10 board



In this case we are going to change the default values to a 4x4 board, no humans and 2 robots. So we're going to see the computer playing against itself. We cannot repeat letters for the players even if they are from different type

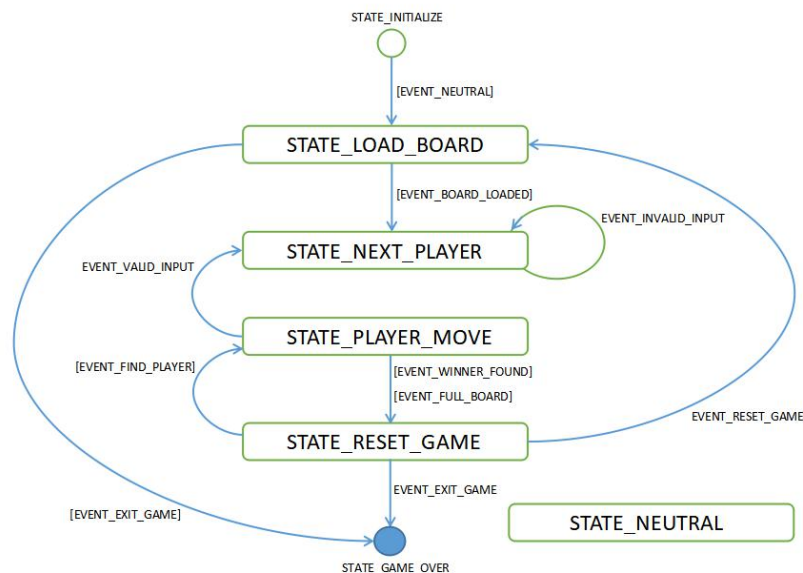


And the match goes randomly until it becomes a tie or some of the robots win, at that point the program ask for human interaction to check whether it reset the game or finish it

Design of Tic-Tac-Toe

Flow of the Game

The design of the program was based on a State Machine to control the flow of the game. I worked on simulation processes of Oil Reservoirs a few years ago, I also made some javascript games and out of it I consider that that is the best way to go with any other game because what we want is to make the user interact with the program at very specific points during the run. Besides, an state machine diagram is straightforward to translate to code. You can see the state machine diagram for tic-tac-toe (ttt from now on) in the following image:



From this drawing we can take the states and events to create a transference table to see how we can get from an initial state to a final one. I created an enum for the state constants and another enum for the event constants.

IN AND OUT STATES	STATE_INITIALIZE	STATE_LOAD_BOARD	STATE_NEXT_PLAYER	STATE_PLAYER_MOVE	STATE_RESET_GAME	STATE_GAME_OVER
STATE_INITIALIZE		EVENT_NEUTRAL				
STATE_LOAD_BOARD			EVENT_BOARD_LOADED			EVENT_EXIT_GAME
STATE_NEXT_PLAYER				EVENT_VALID_INPUT		
STATE_PLAYER_MOVE					EVENT_WINNER_FOUND EVENT_FULL_BOARD	
STATE_RESET_GAME		EVENT_RESET_GAME				EVENT_EXIT_GAME
STATE_GAME_OVER						

Now, we can model this table as an array, a hashtable, but I decided to model it as a switch sentence because it seems to me more readable. For each state there is a function to process that piece of information, so for each transference of state we need to execute a function related to the final state we get, which we can take through the “execute” function.

```
private void execute(States state)
{
    if (state != STATE_NEUTRAL) {
        previousState = currentState;
        currentState = state;
    }

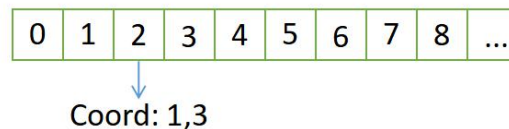
    switch (state) {
        case STATE_LOAD_BOARD:
            sLoadBoard();
            break;
        case STATE_NEXT_PLAYER:
            sNextPlayer();
            break;
        case STATE_PLAYER_MOVE:
            sPlayerMove();
            break;
        case STATE_RESET_GAME:
            sResetGame();
            break;
        case STATE_GAME_OVER:
            sGameOver();
    }
}
```

The game deal with two types of events, automatics and manuals, the automatic events are dispatched by the program itself to change from one state to another, while the manual ones are dispatched by the user once he/she puts information into the program through the console when it require it so. In the diagram the automatics events are drawn inside brackets while the manual one are not.

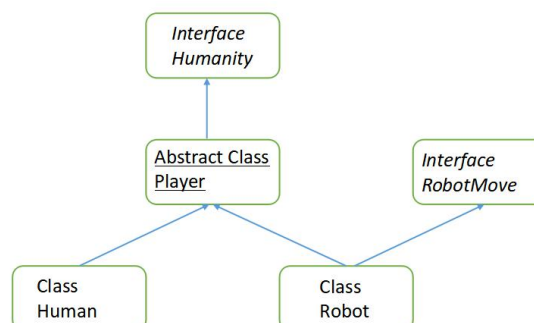
Regarding to the events handling, I use a queue for managing the in and out of each event in a iterative loop. Every state throw the events inserting them into the queue so when we check for the queue emptiness on each iteration, that event is thrown to the state machine to generate a new state. The other way around for doing this is making the state machine to call itself in recursive loop, but this approach has the disadvantage of overflowing the heap so it could cause a stack overflow error after some usage.

Board and Players

The board is a class to register the move of each players and it knows when this movements boils down to a tie, a winner, or if it needs another move because the game still goes on. It has a map of the positions to play, modeled by an array of integers. Also, the board make transformations between “row, column” coordinates and linear array positions which goes from 0 to “total cells”, to do that, it use another class called Coord as a data transfer object.



On the other hand, the players are divided in humans and robots, but they both are still players, so it was kind of obvious to inherit human and robot classes from the class player to keep common properties together. Also, for each move on the board the state machine ask to each player if it is human, but that question can be answered just for the specific implementation of each one, so I created an interface Humanity for the abstract class Player in order to make the children to answer the question, and in the case of robots I created an interface RobotMove, so they have to answer how they are going to calculate their moves because we could have several types of method for the robot to do that.



Configuration and Guidelines

There are three files that cross over the program, one of them is a file for the constants used by some classes, other one is for reading the properties file and another one is a custom exception to handle some scenarios very tied to what the program do.

About S.O.L.I.D. principles

Single Responsibility Principle is something that I always try to accomplish, in this case for instance I have a board, players, an state machine and each of them is a single piece with a very specific responsibility and you can see that at the package level as well with models, machina, exceptions and config, it is the best way to get it clear.

The Open/Closed Principle is something included in this design. The state machine ask if the player is human to know if it has to ask for the coordinates through the console or if it sends a message to the robot to make it calculate automatically the coordinates to play. By default the robot calculate a random number to take a position on the board, however, the idea was to add another player called AI to implement the minimax algorithm to play ttt, that would be a terrific future improvement and it would be very easy, I would create another class AI which would extends from Player and implements RobotMove and make it calculate its move through the minimax algorithm. This way, I haven't touch any other model or the robot implementation that already exist, so they are closed but I have extended the functionality of the game. Either way, this class structure allows the game to have a bunch of player of any kind humans or robots, so you could just set up robot players and see how the play against each other without human interaction

The Liskov Substitution Principle states that objects of a superclass shall be replaceable with objects of its subclasses without breaking the application, and thanks to heritage actually I am creating humans and robots players and all of them are treated as if they were the superclass Player. In this case at least, the children are not restricting the superclass methods because Player is an abstract class with the mayor implementation of its features.

The Interface Segregation Principle states that clients should not be forced to depend upon interfaces that they do not use, and in this case I have created two interfaces so I don't force to humans player to have to implement the method calculateMove if that is something just for robots, so I have one interface for asking if the player is human an another one to make the robots implement the calculateMove method.

The general idea of Dependency Inversion Principle is that High-level modules, which provide complex logic, should be easily reusable and unaffected by changes in low-level modules, which provide utility features. To achieve that, we need to introduce an abstraction that decouples the high-level and low-level modules from each other. For this program the high-level module is the state machine and the low-level modules are the models like board and players, in this case to accomplish a complete abstraction between them I would have to create an Interface for the board, because if I want to create not a square board but another type of it, like a rectangular one, I need to refactor state machine to handle the interface instead of an specific board instance, the work is already done with players.

Clean Code

Around this topic there's a lot of tips and practices to consider. Nevertheless, I will describe the ones that I always keep in mind and that you can see reflected into the code:

- All variables, methods and classes use CamelCase style. The only difference: a first letter of class should be capital
- Constants use all capital letters divided with underscore
- Not special characters and numbers in variables name

- Don't use too much words in the variable/method name. The exception to the rule is the name of the methods for the junit tests which should be very descriptive
- Verbs for function names and nouns for classes and attributes
- Variables represent the value it was created for
- All variables in the class are declared at the top of the class
- If variable is used only in one method then it is declared as a local variable of this method
- One method should be responsible only for one action
- Not code duplication in the project

Guidelines Checklist

- Software design (Clean Code, SOLID) is more important than a highly developed AI. **Explained above**
- You may use external libraries only for testing or building purposes e.g. JUnit, Gradle, Maven, Rspec, Rake, GulpJS, Mocha, etc. **junit, assertj, mockito, gradle**
- If you are coding in JavaScript - you can use node.js as your runtime environment, the same way you can use JRE, if you are coding in Java, etc. You can use the provided standard APIs within these platforms. **Java8**
- You cannot use simple libraries like Apache StringUtils, or complex frameworks like Spring MVC, Spring Boot, Django, React, Angular, etc. **None of them is used**
- Please provide an explanation how to run your code. **Explained above**
- Please explain your design decisions and assumptions. **Explained above**
- Don't include executables* in your submission. **Source code is compiled through Dockerfile**
- Please write your solution in a way, that you would feel comfortable handing this over to a colleague and deploy it into production. **tar.gz or zip file**
- We especially look at design aspects (e.g. OOP) and if the code is well tested and understandable. **The project has 99% in line coverage**