Coursework I manual: chess video game

Typos/errors? Email markel.vigo@manchester.ac.uk and I'll get them fixed.

Table of Contents

Submission

The submission deadline is March 11 (Friday of week 5) at 6PM. This is a formative assessment exercise in order to learn the Java development practices, apply weekly theoretical principles in a real project and familiarise with the process of testing and submission, which will be useful for the summative coursework later on. Consequently, marks are not going to be awarded for this coursework but you will receive feedback over the duration of the project in different ways:

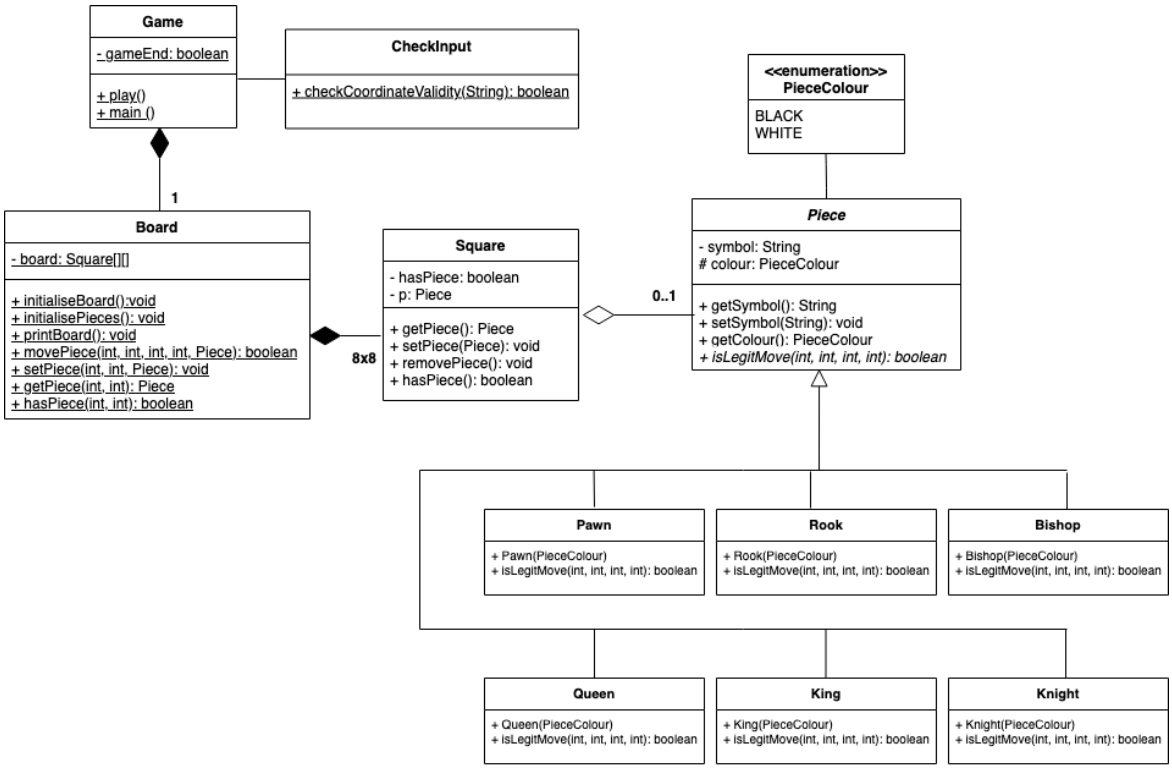
- Using the tests that will guide you through the development process.
- Asking questions in the forum or at labs.
- On the landing lecture in week 7, which will be focused on the video game.

---

1. Instructions

In this coursework we'll be asking you to implement a console-based chess game. The game **must** meet the provided specification, so you should take time to read all instructions carefully.

1. The structure of the following UML class diagram informs the design of the application. Your implementation must meet the structure provided in the UML diagram.



Check the UML reference sheet to help you understand the syntax of the above class diagram.

2. Most classes and methods are self-explanatory but to remove any ambiguity, the logic of the game, which is handled in the `Game` driver class, implements the following steps:

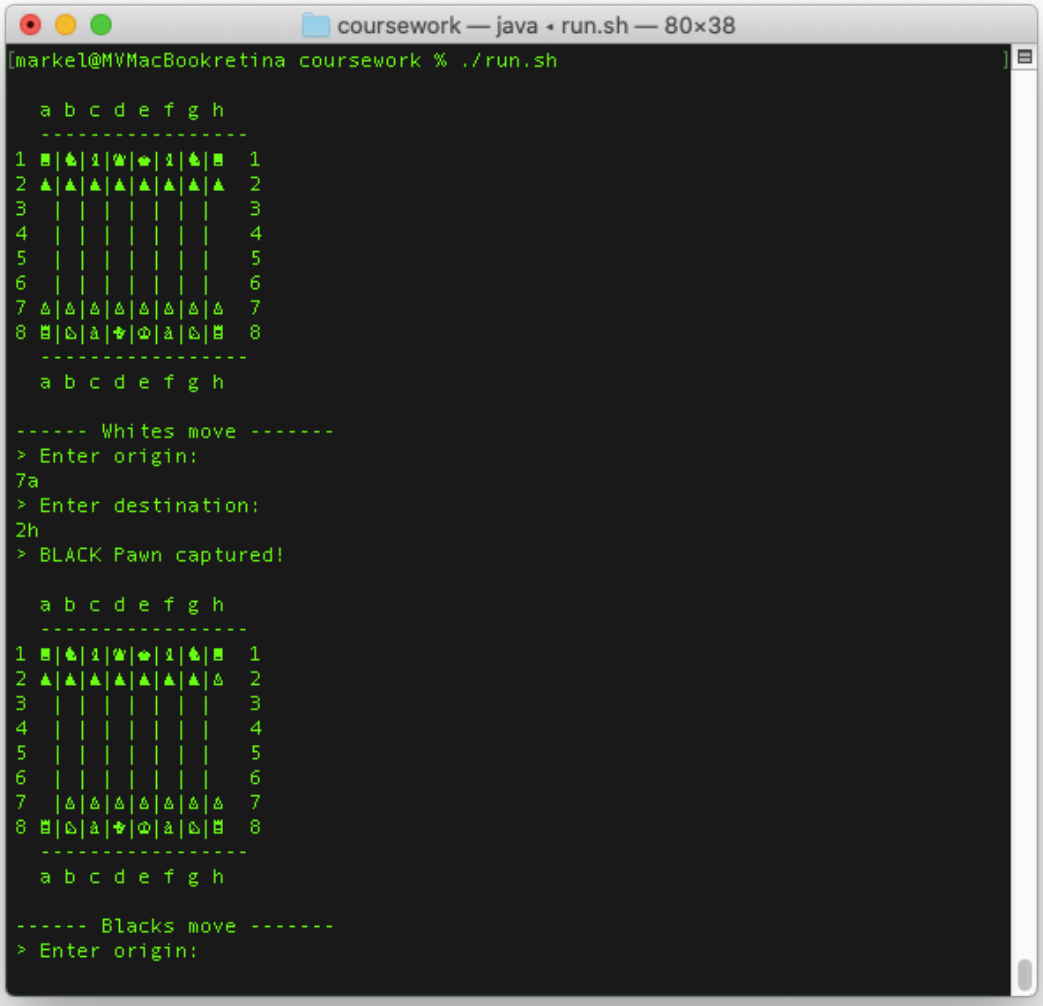
    1. Initialising the board, the pieces and printing it using the `Board` class in the `main` method. The logic of the game is handled in the `play()` method.
    2. Input handling and making sure that input values are valid (through the `CheckInput` class);
    3. Managing turns (ie which pieces are moved when);
    4. Making sure there is a piece on the origin coordinates;
    5. If the coordinates are valid as indicated by (2) and (4), making sure that the intended movement is legitimate (through the `isLegitMove`).
    6. Making movements and printing the board after every move (through `Board`);
    7. Managing the end of the game and who the winner is.

3. The game is partially developed. You will have to fill the gaps as defined in the weekly plan. At the outset you can compile and run the game but you will have to address two type of implementation problems: (a) the game is not following the specifications of the UML diagram (to be addressed from week 2 onwards); (b) Since the rules of the game are not implemented the game has the following non-valid behaviour:

    - Any input from players is valid so you will have to make sure that only the coordinates that follow the established format are accepted. Right now, if the input

is invalid Java generates a runtime error. This will be addressed from week 3 onwards.

- When the coordinates follow the expected format, all pieces can be moved in any direction, which is not valid. See for instance the move of a white pawn below. You can start addressing these issues from week 1.



2. Running and testing the game

Clone the repository from GitLab. Your URL should be

`https://gitlab.cs.man.ac.uk/comp16412_2021/chess-coursework_<username>`

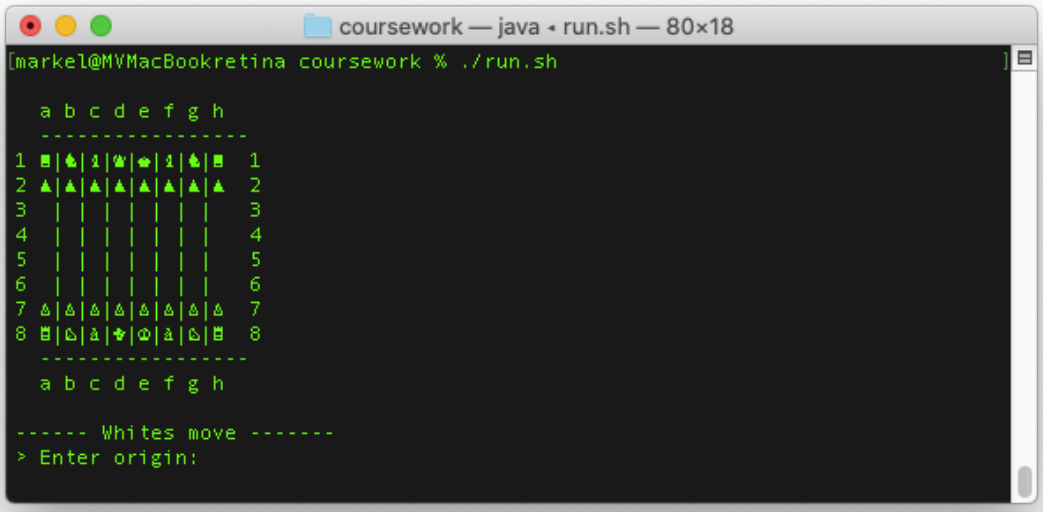
We have provided two scripts to run and test your chess game.

2.1 Running

You should be able to run the current codebase by typing

```
./run.sh
```

in the console or terminal. Do not edit or move the scripts, do not change the folder structure either as the scripts may not run. You should see the board and a prompt asking you to enter the origin coordinates of a piece.



2.2 Testing

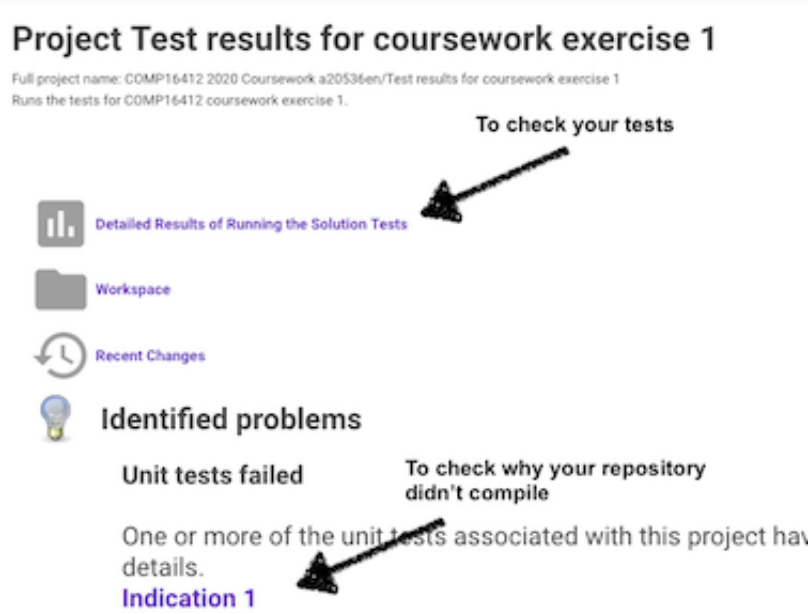
There are two ways of testing your project.

1. Locally in your PC. After cloning the repository in your machine you can run a script that compiles the tests and runs all of them: `./run-tests.sh` which shows a long list of tests with a green tick next to them if they pass and a red cross otherwise.

2. Remotely on a server. Every time you push your current repository to GiLab, you will be able to see the outcome of the tests in Jenkins, a continuous integration server at https://ci.cs.manchester.ac.uk/jenkins/. This continuous integration server runs the tests agains the remote repositories.

If the tests were run successfully, click on "Detailed Results of running the solution tests" where you will see the distribution of the test that passed and failed.

If your repository did not compile you can check the output by clicking in "Indication 1".

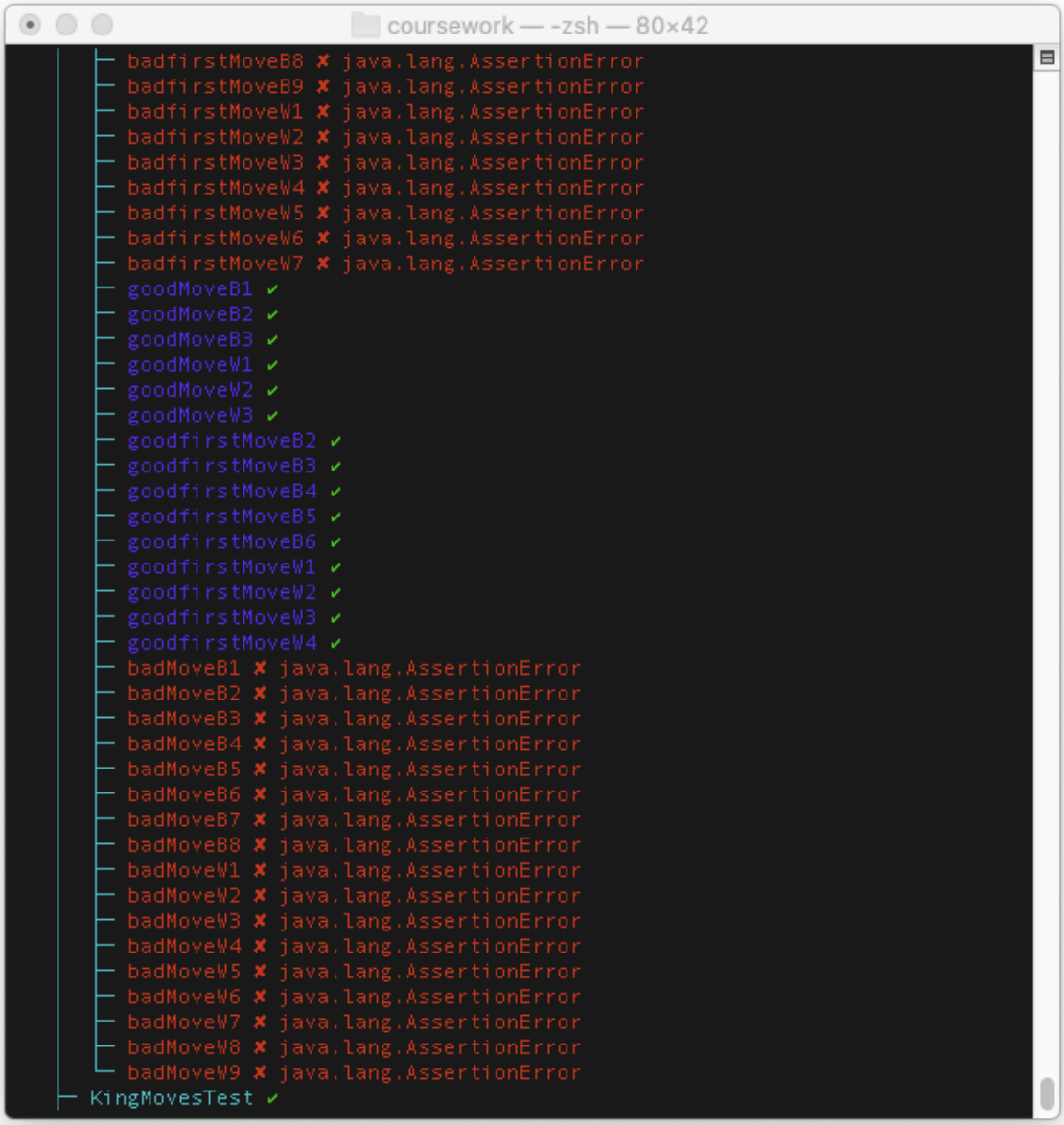


What is testing? Unit testing is a software engineering technique to assess whether your Java programs follow the expectations of the developer. Tests can typically be found in the the `test` folder.

Take for instance the `PawnMovesTest.java` test class and check the `badMoveB2` test.

```
1   @Test
2   public void badMoveB2( {
3       setUpforPawnB();
4       Pawn p = (Pawn) Board.getPiece(3,4);
5       assertFalse(p.isLegitMove(3,4,4,4));
6   }
```

Among many others, the above test fails according to the output of the script:



Why is this? Let's check the test.

- Line 3 creates a state of an hypothetical game where the board looks like.

- Line 4 gets a black pawn from the 3,4 coordinates which translated to the **printed**

board's coordinate system is 4e.

- Line 5 indicates that moving the black pawn from 4e to 5e should not be allowed so the `isLegitMove` should return `false`. In the current implementation, this move is allowed as `true` is returned by default. The test fails because it expects that `isLegitMove` returns `false` for this piece, board state and movement.

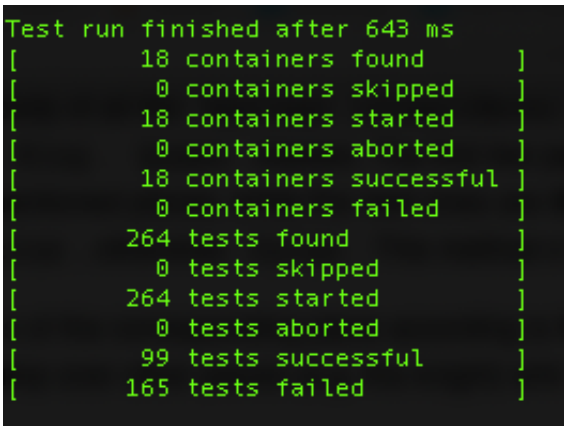
⚠️ Important: note that if you intend to work on an Integrated Development Environment (IDE) it may create extra folders beyond your control, which will prevent the scripts from running. In previous years some students submitted their coursework through their IDEs resulting in losing all the marks associated to automated testing. Make sure you are using a simple editor such as Atom, Sublime or similar

3. Test-driven development

264 tests are provided to guide your development under the `src > test`. folder.

⚠️ You must not edit the tests.

On the first run 99 tests should pass while 165 fail. You should be able to replicate this output:



There are two type of tests:

- Specification related tests make sure the specifications are followed as defined by the

UML diagram. It is mostly about the visibility and modifiers of classes, methods and attributes. These tests belong to those test classes that end as `...SpecsTest.java` such as `KingSpecsTest.java`.

- Tests about piece movements are the majority of tests. These tests belong to those test classes that end as `...MovesTest.java` such as `KingMovesTest.java`. You can start addressing these tests from week 1. All the moves tests are concerned with the `isLegitMove` methods of all the pieces with legal and non-legal moves as the one described in Section 2.2.

The next table summarises all the tests.

Test class	Total number	Pass	Fail	When to start
BishopMovesTest	27	19	8	w1
KingMovesTest	25	12	13	w1
KnightMovesTest	27	8	19	w1
PawnMovesTest	48	15	33	w1
QueenMovesTest	45	14	33	w1
RookMovesTest	30	9	21	w1
PieceTest	10	7	3	w2
SquareTest	7	0	7	w2
BishopSpecsTest	4	1	3	w2 and w4
KnightSpecsTest	4	1	3	w2 and w4
PawnSpecsTest	4	1	3	w2 and w4
QueenSpecsTest	4	1	3	w2 and w4
RookSpecsTest	4	1	3	w2 and w4
CheckInputTest	13	1	12	w3
BoardTest	8	8	0	NA
Total	**264**	**99**	**165**	

☝️ You may wonder why some move tests pass now considering that the body of `isLegitMove` is not implemented. Since `isLegitMove` returns `true` by default now so the videogame can compile and some tests are expected to return `true`, these

tests pass for the time being. This will change as soon as the body of the different `isLegitMove` methods is implemented. As you develop this method, try to run and pass the tests, your goal is to pass all the tests. For every week there will be a plan to develop the game by addressing the tests that are related to the learning outcomes of the week. The more passing tests you have, the closer you will be to completing the chess videogame.

4. The rules of the game

4.1 The basic rules

This chess game implements a subset of the rules of chess including:

- The initial setup as defined in the Rules of Chess wikipedia page.
- All the basic moves.
- The game ends when a king is captured or one player resigns.
- Note that the notation for the coordinates is different in our game. The original game is *a* to *h* on the X axis (left to right) and 8 to 1 on the Y axis (top to bottom), while in our case the latter should be 1 to 8 from the top to the bottom.

4.2 What the game should not implement

- *En passant* (pawn)
- Pawn promotion (pawn)
- Castling (king and rook)
- Check detection
- Draw detection
- Time control

4.3 Other considerations

- The game is supposed to be played by two players in the same terminal in the same computer.
- Squares are not coloured.
- Pieces are coloured using the black and white unicode chess symbols. See for instance: ♔ and ♜.
- Make sure to configure your terminal to use a monospaced font. This gives the same width to whitespace and pieces. Otherwise, the board and the pieces may look odd. Oddly, in a terminal with a black background the black pieces look white (don't get confused with this) – see examples in the above screenshots.

4.4 Commands and accepted format

- Movement command: each movement command must contain two characters that

convey a specific coordinate of the chess board. The first one is a number from `1` to `8` (corresponding to a `i` coordinate in a matrix), while the second is a character from `a` to `h` (corresponding to a `j` coordinate in a matrix). Note that while commands are in the `1` to `8` range, the indices of Java arrays start at `0`. So the commands are internally "translated" in order to be able to run the tests which use a `0` to `7` range.

- Ending the game: type `END`.