Victoria University of Wellington
School of Engineering and Computer Science

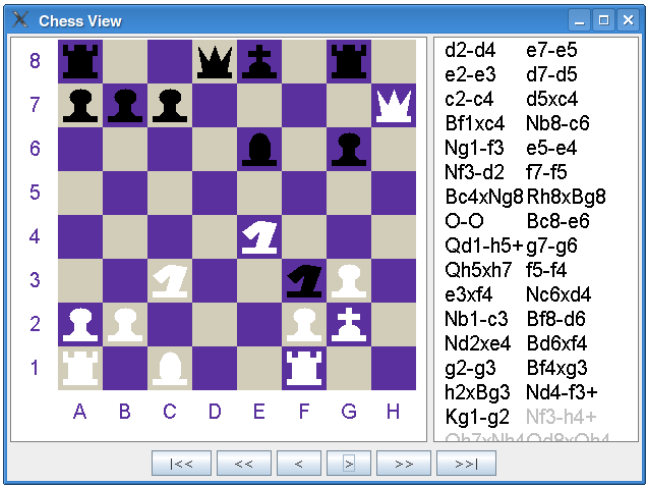
SWEN221: Software Development

Assignment 3

Due: Monday 30th March @ Mid-night

The Chess View System

The `ChessView` system is a simple Java application for viewing chess games written in *long algebraic chess notation*. The following shows a screenshot from `ChessView`:



`ChessView` has only one window, which allows the user to move forward and backward through a chess game. The moves of the game, written in long algebraic chess notation, are given in the rightmost pane of the window; the current state of the chess board is shown in the leftmost pane.

Long Algebraic Notation

Long algebraic chess notation is a way for writing down the moves taken during a chess game. The following illustrates the start of a game in this notation:

White	Black
`e2-e4`	`e7-e5`
`d2-d4`	`e5xd4`
`Nb1-c3`	`Qd8-f6`

The first move by White, `e2-e4`, indicates the pawn at position `e2` will advance to position `e4`. When indicating pawn movement, no piece specifier is given. However, when indicating the movement of

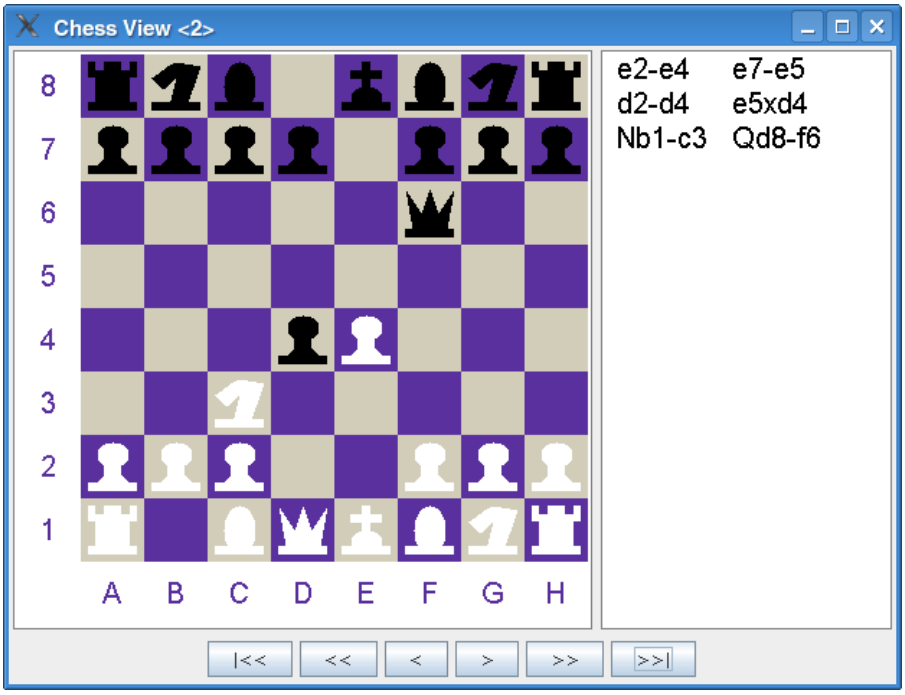


Figure 1: Illustrating the state of a game after 3 rounds.

other pieces a specifier is always given. The specifiers are: `N`=kNight; `B`=Bishop; `R`=Rook; `Q`=Queen; `K`=King. So, for example, White's last move above, `Nb1-c3`, indicates his Knight moves from `b1` to `c3`. Finally, a move where one piece takes another is indicated using a `x`, as in Black's move `e5xd4` — the (black) pawn at position `e5` takes the (white) pawn at `d4`. Figure 1 shows the state of the board after the moves in the above game have been made.

There are a number of other moves which can be made during a game of chess, including: *putting a king in check* (e.g `Qe1-h4+`), *castling* (e.g. `O-O`), *pawn promotion* (e.g. `b7-b8=N`) and *en passant* (e.g. `b4xa3ep`). If you are not familiar with the rules of Chess, the following links provide an excellent starting point:

```
http://en.wikipedia.org/wiki/Chess
http://en.wikipedia.org/wiki/Chess_notation
http://en.wikipedia.org/wiki/Algebraic_chess_notation
```

You can also find many other good resources on the Internet regarding the game of chess.

NOTE: *you do not need to be any good at playing chess in order to complete this assignment.*

Running Chess View

To get started, download the `chessview-buggy.jar` file from the lecture schedule on the course website. You will find several example games are provided as well. As usual, you can run the program from the command-line as follows:

```
java -jar chessview-buggy.jar game-001.txt
```

You should see the ChessView window open up and a chess board being displayed. You should also be able to navigate forward and backward through a game.

NOTE: *the version of chessview you have been given is broken (obviously). This means you will not be able to successfully navigate a full game yet. The ChessView window will show a move that it thinks is illegal in red, and will not let you move passed this point.*

Understanding Chess View

When you expand or import the chessview jar file, you should find the following java packages:

```
assignment3/chessview/
assignment3/chessview/pieces/
assignment3/chessview/moves/
assignment3/chessview/viewer/
assignment3/tests/part1/
assignment3/tests/part2/
assignment3/tests/part3/
```

Some notes on these packages:

- The `assignment3/chessview/` package contains the high-level game classes, including those for representing the chess board and the game itself.
- The `assignment3/chessview/pieces/` packages contains a class for each of the different chess pieces. These contain code related to the movement of the pieces.
- The `assignment3/chessview/moves/` packages contains a class for each of the different kind of move that can be made in the game. These contain code related to structuring a move, and ensuring it is valid.
- The `assignment3/chessview/viewer/` package contains code related to the ChessView interface. **NOTE:** you do not need to modify any code in this package.
- The three test packages `assignment3/tests/part1/`, `assignment3/tests/part2/` and also `assignment3/tests/part3/` contain JUnit tests for each of the three parts of the assignment (see below).

The class `assignment3/chessview/Board` is one of the main classes in the ChessView system. This is responsible for representing the state of the chess board, including the position of all of the pieces. This uses a 2-dimensional array to represent the chess board, where each cell in that array represents a location on the board. The board also provides methods for determining whether a particular diagonal, horizontal or vertical move will be unobstructed. Figure 2 provides a visualisation of a simple `Board` object.

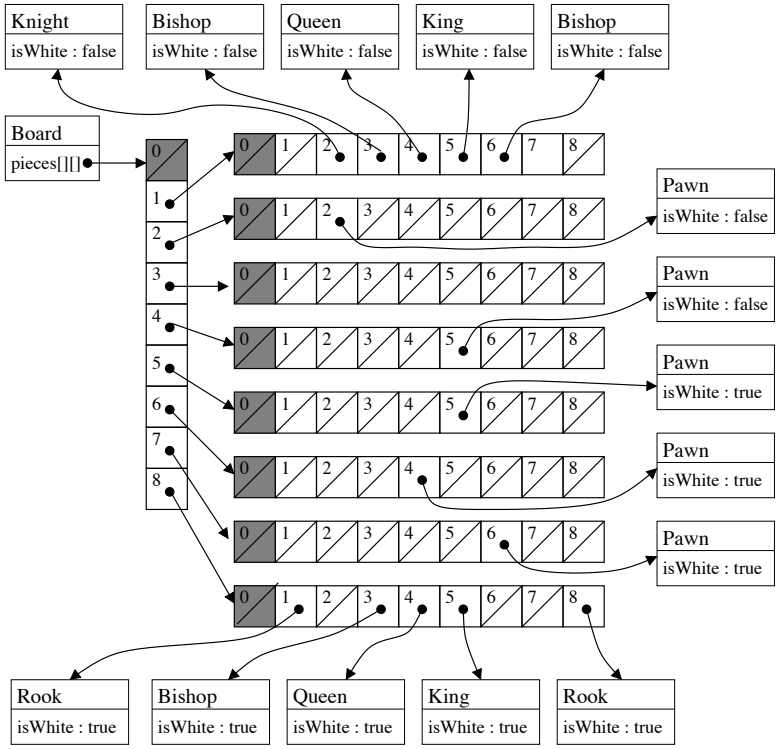


Figure 2: Illustrating an example `Board` object, which contains a 2-dimensional array of pieces. An empty square is indicated by a box with a slash, and this corresponds to the array cell holding `null`. The first element of the row array, and the first element of each column array are unused (marked in gray). This allows the row numbers to line up with those of the normal chess board. Column `a` of the chess board corresponds to index 1 in the column array, and so on for columns `b`, `c`, ..., etc.

Part 1 — Valid Moves

The first objective is to make sure that ChessView correctly recognises all of the simple moves that a piece can make, including taking of other pieces. At this stage more complex moves, such as castling, checking, en passant and pawn promotion, should be ignored. A suite of tests for this part is provided in package `assignment3/tests/part1`. You can run these tests individually, or all together by running the `Part1TestSuite`. **You are advised to write additional tests to ensure the system is working correctly.**

Part 2 — Invalid Moves

The second objective is to make sure that ChessView correctly recognises when a simple move is invalid. For example, Pawns may not move backwards; and, Bishops cannot move in a horizontal line. Again, more complex moves, such as castling, checking, en passant and pawn promotion, should be ignored. A suite of tests for this part is provided in package `assignment3/tests/part2`. You can run these tests individually, or all together by running the `Part2TestSuite`. **You are advised to write additional tests to ensure the system is working correctly.**

Part 3 — Complex Moves

The final objective is to ensure that the complex moves work properly in ChessView. The complex moves are *castling*, *checking*, *en passant* and *pawn promotion*. You will find that, to complete this part, you will have to make larger modifications to the code. A suite of tests for this part is provided in package `assignment3/tests/part3`. You can run these tests individually, or all together by running the `Part3TestSuite`. **You are advised to write additional tests to ensure the system is working correctly.**

Submission

Your source files should be submitted electronically via the *online submission system*, linked from the course homepage. The required files are:

```
assignment3/chessview/moves/MultiPieceMove.java
assignment3/chessview/moves/SinglePieceTake.java
assignment3/chessview/moves/Castling.java
assignment3/chessview/moves/NonCheck.java
assignment3/chessview/moves/Move.java
assignment3/chessview/moves/EnPassant.java
assignment3/chessview/moves/PawnPromotion.java
assignment3/chessview/moves/SinglePieceMove.java
assignment3/chessview/moves/Check.java
assignment3/chessview/pieces/PieceImpl.java
assignment3/chessview/pieces/Queen.java
assignment3/chessview/pieces/Piece.java
assignment3/chessview/pieces/King.java
assignment3/chessview/pieces/Knight.java
assignment3/chessview/pieces/Pawn.java
assignment3/chessview/pieces/Rook.java
assignment3/chessview/pieces/Bishop.java
assignment3/chessview/Round.java
assignment3/chessview/Board.java
assignment3/chessview/Main.java
assignment3/chessview/Position.java
assignment3/chessview/ChessGame.java
assignment3/chessview/viewer/BoardCanvas.java
assignment3/chessview/viewer/RoundCanvas.java
assignment3/chessview/viewer/BoardFrame.java
```

You must ensure your submission meets the following requirements (which are needed for the automatic marking script):

1. **Your submission is packaged into a jar file, including the source code**. See the export-to-jar tutorial linked from the course homepage for more on how to do this. *Note, the jar file does not need to be executable.*

2. **The names of all classes, methods and packages remain unchanged**. That is, you may add new classes and/or new methods and you may modify the body of existing methods.

However, you may not change the name of any existing class, method or package. *This is to ensure the automatic marking script can test your code.*

3. **All JUnit test files supplied for the assignment remain unchanged.** Specifically, you cannot alter the way in which your code is tested as the marking script relies on this. This does not prohibit you from adding new tests, as you can still create additional JUnit test files. *This is to ensure the automatic marking script can test your code.*

4. **You have removed any debugging code that produces output, or otherwise affects the computation.** *This ensures the output seen by the automatic marking script does not include spurious information.*

Note: Failure to meet these requirements could result in you getting zero marks for the assignment.

Assessment

This assignment will be marked as a letter grade (A+ ... E), based primarily on the following criteria:

- **Correctness of Part 1 (20%)** — does submission adhere to specification given for Part 1.
- **Correctness of Part 2 (30%)** — does submission adhere to specification given for Part 2.
- **Correctness of Part 3 (40%)** — does submission adhere to specification given for Part 3.
- **Style (10%)** — does the submitted code follow the style guide and have appropriate comments (inc. Javadoc)

As indicated above, part of the assessment for the coding assignments in SWEN221 involves a qualitative mark for style, given by a tutor. Whilst this is worth only a small percentage of your final grade, it is worth considering that good programmers have good style.

The qualitative marks for style are given for the following points:

- **Division of Concepts into Classes**. This refers to how *coherent* your classes are. That is, whether a given class is responsible for single specific task (coherent), or for many unrelated tasks (incoherent). In particular, big classes with lots of functionality should be avoided.
- **Division of Work into Methods**. This refers to how well a given task is split across methods. That is, whether a given task is broken down into many small methods (good) or implemented as one large method (bad). The approach of dividing a task into multiple small methods is commonly referred to as *divide-and-conquer*.
- **Use of Naming**. This refers to the choice of names for the classes, fields, methods and variables in your program. Firstly, naming should be consistent and follow the recommended Java Coding Standards (see `http://g.oswego.edu/dl/html/javaCodingStd.html`). Secondly, names of items should be descriptive and reflect their purpose in the program.
- **JavaDoc Comments**. This refers to the use of JavaDoc comments on classes, fields and methods. We certainly expect all `public` and `protected` items to be properly documented. For example, when documenting a method, an appropriate description should be given, as well as for its parameters and return value. Good style also dictates that `private` items are documented as well.

- **Other Comments**. This refers to the use of commenting within a given method. Generally speaking, comments should be used to explain what is happening, rather than simply repeating what is evident from the source code.
- **Consistency**. This refers to the consistent use of indentation and other conventions. Generally speaking, code must be properly indented and make consistent use of conventions for e.g. curly braces.

Finally, in addition to a mark, you should expect some written feedback highlighting the good and bad points of your solution.