CS5001 Object-Oriented Modelling, Design and Programming
Practical 2 – Stratego

School of Computer Science
University of St Andrews

Due Friday week 7, weighting 35%
MMS is the definitive source for deadlines and weightings.

In this practical, you will be provided with an object-oriented model, and your task will be to implement it in Java. The model is specified by two things: a UML class diagram showing the required classes, and a suite of tests that define how your program should behave. The details of implementation are up to you: after creating the classes in Java, you will need to fill in their methods yourself, and decide on any additional fields, methods, classes etc. that you might need in order to meet the requirements.

For this practical, you may develop your code in the IDE or text editor of your choice, but you must ensure all your source code is in a folder named `CS5001-p2-stratego/src/`.

Stratego

Stratego is a two-player strategic board game invented in the Netherlands in 1942. Players arrange their armies on the board at the beginning of the game, and take turns moving them. The pieces have different ranks which determine their abilities, but these ranks are hidden from the player's opponent, who does not know which piece is which!

Figure 1: A game of Stratego in progress

On a player's turn, they can either move or attack, and the winner is the first player to attack their opponent's flag.

Game Rules

The two players sit facing each other across a 10×10 board, with 40 pieces each. There are 12 different types of piece:

- 1 Marshal (rank 10)
- 1 General (rank 9)
- 2 Colonels (rank 8)
- 3 Majors (rank 7)
- 4 Captains (rank 6)
- 4 Lieutenants (rank 5)
- 4 Sergeants (rank 4)
- 5 Miners (rank 3)
- 8 Scouts (rank 2)
- 1 Spy
- 6 Bombs
- 1 Flag

They place these 40 pieces in the first 4 rows in front of them, in whatever order they wish. However, the pieces are designed so that only the player who owns them can identify which one is which.

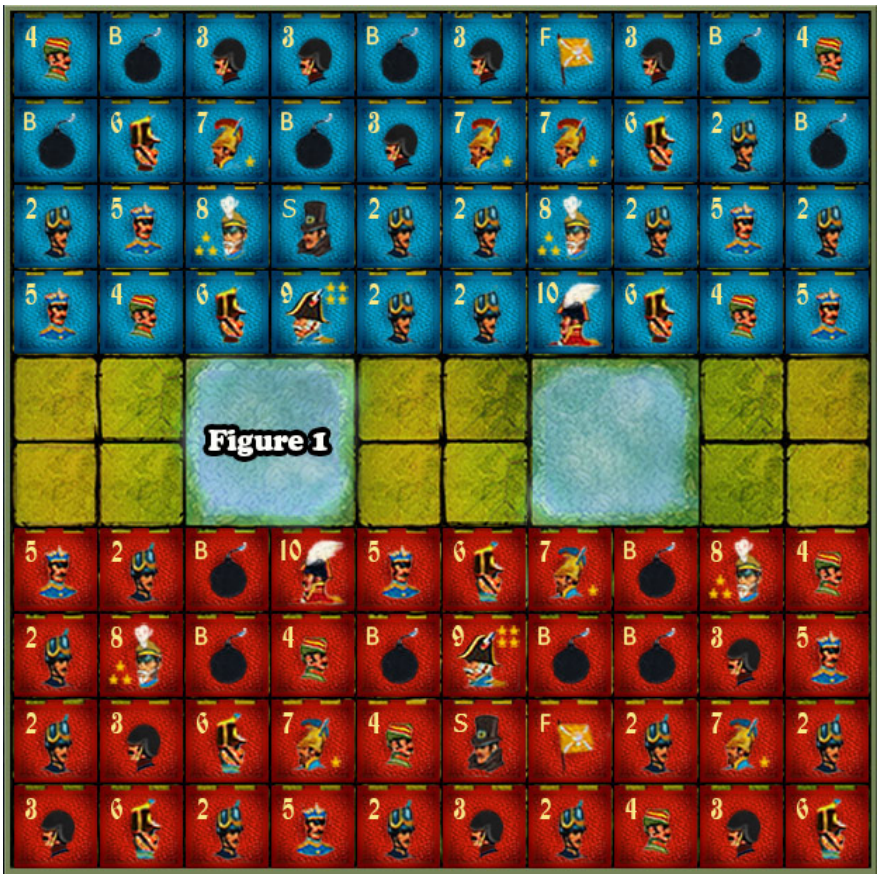

Figure 2: A possible setup at the beginning of the game

After setting up the pieces, players take turns. On each player's turn, they can either **move** or **attack**.

Moving

If a player chooses to *move*, they choose one of their pieces and move it one space forward, backward, or sideways. They may not move diagonally, and may not move onto a space already occupied by another piece. They may also not move onto or over any of the 8 squares occupied by water (see the figure above).

The six *Bomb* pieces and the one *Flag* piece can never be moved.

The six *Scout* pieces can move more than one square: they can move as many open spaces as they like in a single direction (forward, backward or sideways), but cannot jump over pieces. A Scout attacks like a normal piece.

Attacking

If a player chooses to *attack*, they choose one piece they own that is adjacent to a piece belonging to the opponent (diagonally adjacent does not count). They declare the piece they are attacking with, and the piece being attacked. The *ranks* of the two pieces are then revealed, and the piece with the lower rank is destroyed (removed from the board). If the attacking piece was victorious, it automatically moves into the square the defending piece occupied.

Example: My General is next to an opponent's piece. On my turn, I choose to attack the opponent's piece with my General. I reveal that my piece is the General (rank 9) and my opponent reveals that their piece is a Lieutenant (rank 5). Since rank 9 is higher than rank 5, my General wins. The opponent's Lieutenant is removed from the board, and my General moves into its place. It is now the opponent's turn.

If the two pieces have the same rank (e.g. they are both Captains) then they are both destroyed.

The six *Bomb* pieces and the one *Flag* piece can never attack.

Special Pieces

Some pieces have special rules attached to them when they are involved in combat:

- **Bomb**: If any piece attacks a Bomb, the Bomb is destroyed but the attacking piece is also destroyed (unless it is a Miner).
- **Miner**: If a Miner attacks a Bomb, the Bomb is destroyed and the Miner moves into the Bomb's space.
- **Spy**: If any piece attacks a Spy, the Spy is destroyed. However, if a Spy attacks a Marshal, then the Marshal is destroyed instead. If a Spy attacks any piece other than a Marshal or a Flag, then the Spy is destroyed.

Winning the Game

If you attack the opponent's Flag with any of your pieces, you capture the Flag and win the game.

These rules are all that you need to implement for this practical. For clarification, feel free to look at the rules at https://www.ultraboardgames.com/stratego/game-rules.php.

System Specification

You are required to implement the classes shown in the following UML class diagram, including all the public methods and attributes shown. Your program should not require any input from the user, nor print any output to the screen. Your code can be tested by running the JUnit tests that are included with this practical. This can be done on the lab machines using *stacscheck*, by navigating into your `CS5001-p2-stratego/` directory and running the following command:

```
stacscheck /cs/studres/CS5001/Coursework/p2-stratego/Tests
```

This will compile your classes and run all the provided tests on them. You can access *stacscheck* on the lab machines or via SSH, or if you use a Linux or Mac machine, you can install it on your own computer.

Important: The behaviour we want from the system is defined by these tests. If you're not absolutely sure what a class or method is supposed to do, you should look at the tests that correspond to it and examine the examples and assertions there. The relevant unit test files are in subdirectories of the `Tests/` directory, and they all have names of the form `<classname>Test.java`.

You can add to these by creating your own additional tests if you wish. You can run your own JUnit tests in an IDE, via the command line, or via *stacscheck*. If an aspect of the program's behaviour is not defined by the tests or the UML diagram, you should make your own choice about what the program should do, and document this in your code as a comment.

UML class diagram

The UML diagram included below shows the structure of the system you should build. You should include every class and every public method on the diagram, but you may implement more classes and methods if you wish, and the choice of private attributes and methods is up to you.

A full-size version of this diagram is also available on StudRes.

In this diagram, each yellow box is one class. The 'C' symbol in the top part refers to a class, while the 'A' symbol refers to an abstract class, and the 'E' symbol refers to an enum. Method names written in italics are *abstract methods*.

All pieces in the game belong to the class `Piece`. However, pieces move in different ways, so we have several subclasses that correspond to different piece types. Most pieces move and attack one square orthogonally; these are part of the class `StepMover`, and can be distinguished by their *rank*. However, scouts can move longer distances, so they are part of a different subclass. Finally, some pieces cannot move at all, and these are part of `ImmobilePiece`. Spies, miners, flags and bombs all have unique behaviour, so they have their own subclasses that inherit from the others mentioned. You should use these classes to override any behaviour that these piece types will do differently.

The whole `Game` has two `Players`, and 100 `Square`s (the 10×10 board shown above). If a square is in both a *water row* and a *water column*, it will be a water square.

There are also some methods marked with `{override}`: these are optional, but you can see them as hints as to methods you might want to override in those classes. Think about how those methods might need to behave differently.

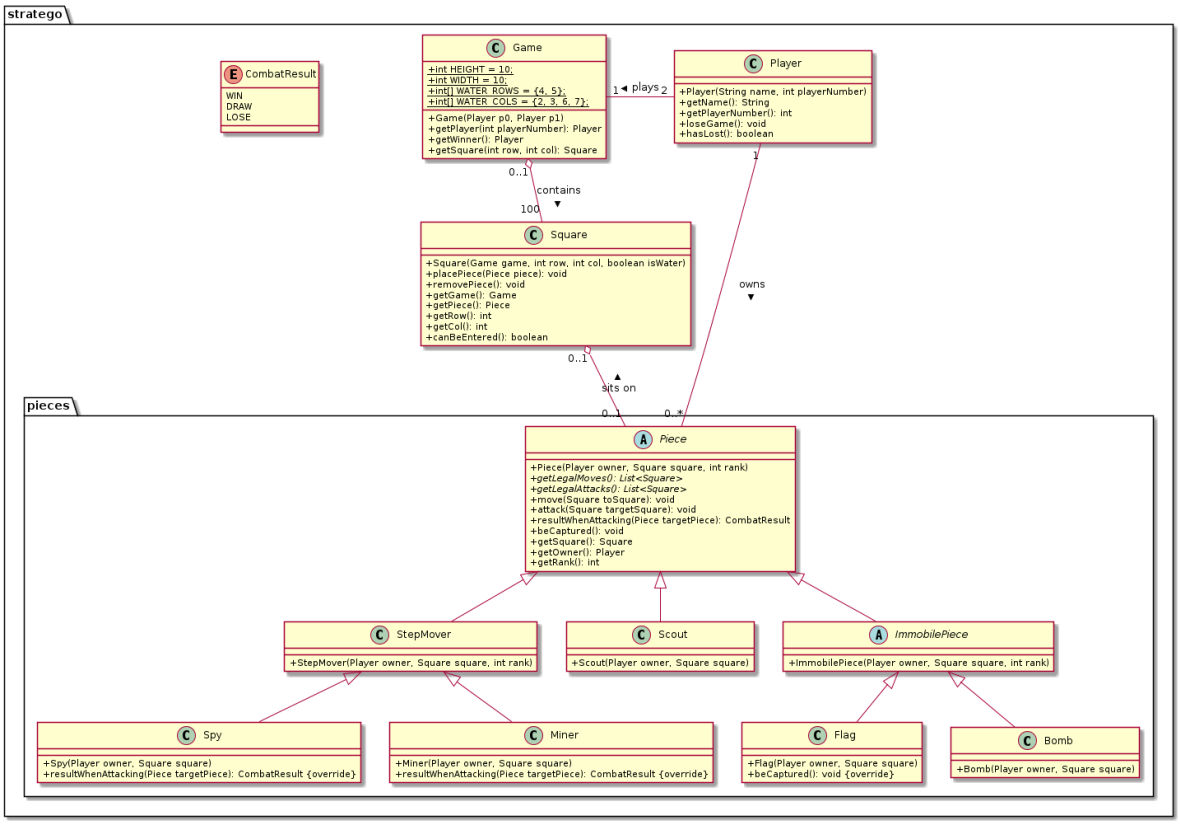

Figure 3: UML class diagram of the required system

Exception handling

An exception of the class `IllegalArgumentException` should be thrown if `placePiece` attempts to place a piece on a square that is already occupied. It may also be used in other situations where an operation cannot be carried out because of an illegal argument. A short message describing what went wrong should be included in the exception. Make sure to catch it and handle it where appropriate.

One or two other exceptions are required by the test suite, and you can find these by looking carefully at the tests!

Packages

Also shown on the UML diagram is the name of the two packages into which your classes should be placed: `stratego`, and its subpackage `pieces`. You should make sure to declare this package at the top of each file, with the line `package stratego;` or `package stratego.pieces;`. Make sure to organise your directory structure appropriately.

Behaviour

You might be surprised by some of the behaviour required by the tests. Maybe you would have designed things differently, or you think the tests are too restrictive. However, this style of programming is typical in *test-driven development*, and being able to write code according to someone else's design is important. So pay attention to the tests, and make sure you stick to what they require.

Certain events, such as a piece being promoted, or a player winning, are triggered by public methods. These events can be forced by calling the appropriate method – for example, `player.loseGame()` – but should also be triggered according to the rules of the game. For example, a player should lose when their flag is captured. You should also note that some methods should be allowed to work even if they break the rules of the game; these are made clear in the tests.

Deliverables

A report is not necessary for this project, but if you have strayed from the specification at all, or if you have made any design decisions that you wish to explain, you can include a short readme file in your `CS5001-p2-stratego/` folder; this should include any necessary instructions for compiling or running your program. Hand in an archive of your assignment folder, including your `src/` directory and any local test subdirectories, via MMS as usual.

Marking

A submission that does not satisfy all the tests above may still receive a good grade. A submission that implements all classes and methods as shown on the UML diagram, using object-oriented methods, but which does not have the required behaviour in all cases, could receive a grade of up to 15 if it is implemented cleanly and intelligently.

A submission which satisfies all the requirements described, including passing all the tests, can achieve any grade up to 20, with the highest grades awarded to implementations that show clarity of design and implementation, with additional tests that show an insight into the problem.

Any extension activities that show insight into object-oriented design and test-driven development may increase your grade, but a good implementation of the stated requirements should be your priority.

See the standard mark descriptors in the School Student Handbook:

https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html#Mark_Descriptors

Lateness

The standard penalty for late submission applies (Scheme B: 1 mark per 8-hour period, or part thereof) as shown at:

https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties

Good Academic Practice

The University policy on Good Academic Practice applies:

https://www.st-andrews.ac.uk/students/rules/academicpractice/