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

School of Computer Science
University of St Andrews

Due Friday week 8, weighting 35%
MMS is the definitive source for deadlines and weightings.

In this practical, you will be provided with an object-oriented model for a Game Server, and your task will be to implement it in Java. The model for the Game 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. Your game should be playable using a client UI that is provided, which will be the same for everyone. The client will communicate with your server using HTTP and your Game Server should provide a HTTP server functionality that can handle these requests. This will also be required for the automated tests.

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. But the game must adhere to the game play specifications, use the protocol specified and be playable with the provided client.

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/src/` and your main method should be in a `GameServerMain` class in `GameServerMain.java`.

`GameServerMain` will take two command line arguments: port and seed. The first argument port should be an integer: the port the server listens on. The second argument seed should be a float: the seed passed to the game for the random number generator.

You may not use any third-party libraries in your code, except for the javax.json library for JSON parsing and generation.

Woodland Diplomacy (WD)

Woodland Diplomacy (*WD*) is a single-player game invented in 2023 by a group of people at the University of St Andrews with the help of ChatGPT. The game is played on a 20×20 board and the player's goal is to help a bunch of woodland animals survive the woodland environment and get to the other side of the woods. The challenge facing these animals through their perilous journey is that the woodland has been taken over by several mythical creatures that are hostile to the animals. The mythical creatures want the woodland all to themselves.

There are different types of mythical creature on the board with different attack values randomly

placed on the board. There are also different spells placed randomly on the board that can be picked up by the animals.

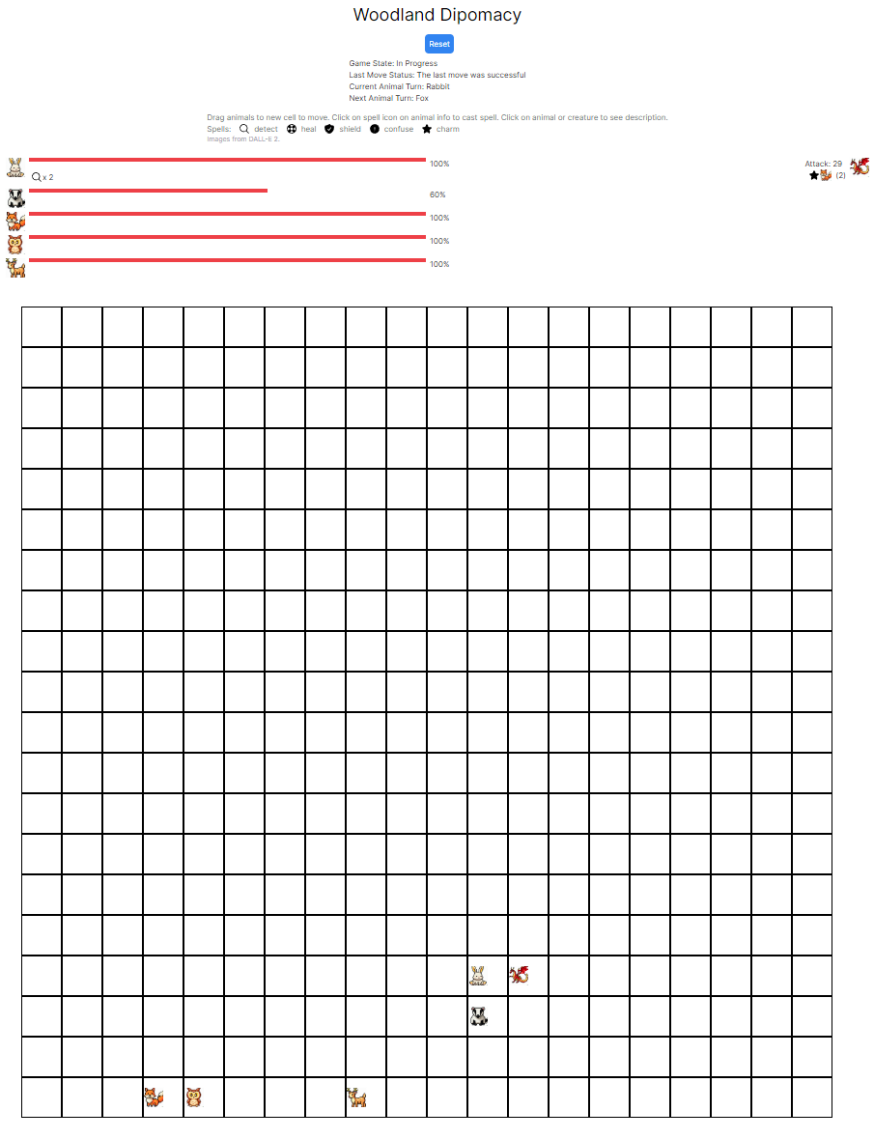

Figure 1: A game of Woodland Diplomacy in progress

The player takes turns to move each animal on the board. Different animals have different movement abilities on the board. If the player reaches a square with a mythical creature, they will get attacked. If the woodland animal reaches a square with a spell, they can pick it up and use it in the future. The woodland animals can potentially use spells to detect mythical creatures on adjacent squares. If the player moves all the animals successfully to the other side of the woodland, they win the game.

Game Rules

The five animals start at random grid cells on the bottom row of the 20x20 board. The five animals must all start on different cells. Each cell can only contain one animal or an animal with

a mythical creature. All animals can only move in straight lines on the board, though each can move different distances or different ways on the straight line.

The different woodland animals are:

- **Rabbit**: *The rabbit has fluffy ears and tail. The rabbit really likes to eat grass.* The rabbit can walk one square in any direction. The rabbit can also jump up to two squares in any direction. The rabbit cannot jump over any other animal. If it attempts to jump over a mythical creature, then it will land in the square of the mythical creature.
- **Fox**: *The fox has a bushy tail. The fox really enjoys looking at butterflies in the sunlight.* The fox can walk one square either horizontally or vertically. The fox can also jump up to three squares either horizontally or vertically, but cannot jump over any other animal. If it attempts to jump over a mythical creature, then it will land in the square of the mythical creature. The fox cannot move diagonally in any way.
- **Deer**: *The deer has antlers. The deer is recently divorced and is looking for a new partner.* The deer can walk one square in any direction. The deer can also jump up to three squares in any direction, and can also jump over any other animal or mythical creature.
- **Owl**: *The owl has wings. The owl has prescription contact lenses but cannot put them on.* The owl can walk one square in any direction. The owl can also fly in any direction for any number of squares. It can fly over any other animal but if it flies over a mythical creature, it will get attacked and will land in the square of the mythical creature. The owl cannot try to fly over a square with both an animal and a mythical creature; this will be considered an invalid move.
- **Badger**: *The badger has a black and white face. The badger is a often mistaken for a very small panda. The badger wears a t-shirt that says "I am not a panda" to combat this.* The badger can walk one square in any direction. The badger can also move exactly two squares (i.e. not one) in any direction by digging. The badger will dig under any other animal or mythical creature.

All animals can walk up to one square based on their ability. If an animal can jump, then when an animal jumps over a cell, it will be able to see what was on the cell it jumped over. If the jump failed because of a mythical creature, then it will land in the square of the mythical creature. If the jump failed because of another animal, then the jump will not take place at all. When an animal digs, it will not be able to see what was on the cell it dug under. Animals start with no spells and a life value of 100. If an animal gets attacked by a mythical creature, it will lose life points. If an animal's life points reaches 0, it will die and the game will end.

The mythical creatures are:

- **Under-appreciated Unicorn (UU)**: *The UU is a unicorn that is under-appreciated by the other mythical creatures because it is often mistaken for a horse with a horn.* The UU has an attack value of 14.
- **Complicated Centaur (CC)**: *The CC is a centaur that has mixed feeling about its love interest, a horse. The centaur is unsure whether they can love them fully.* The CC has an attack value of 36.
- **Deceptive Dragon (DD)**: *The DD is a dragon that practices social engineering. The dragon is very good at sending phishing emails pretending to be a prince.* The DD has an attack value of 29.
- **Precocious Phoenix (PP)**: *The PP is a phoenix that is very precocious. The phoenix understands the meaning of life and the universe.* The PP has an attack value of 42.
- **Sassy Sphinx (SS)**: *The SS is a sphinx that is very sassy. The sphinx is very good at giving sarcastic answers to questions.* The SS has an attack value of 21.

The spells are:

- **Detect**: *The detect spell allows the animal to detect the mythical creatures on the adjacent squares.*
- **Heal**: *The heal spell allows the animal to heal 10 life points.*
- **Shield**: *The shield spell allows the animal to block a mythical creature attack for that turn.*
- **Confuse**: *The confuse spell allows the animal to confuse a mythical creature on a square adjacent to the animal but not the square the animal is occupying. The mythical creature will not attack any animal for the next turn.*
- **Charm**: *The charm spell allows the animal to charm a mythical creature on a square adjacent to the animal but not the square the animal is occupying. The mythical creature will not attack the charming animal for the next three turns.*

Turn here means the turn of all the animals. Therefore, **Confuse** would last until the next turn of the animal that used the spell. **Charm** would last until the third turn of the animal that used the spell.

The animals are all placed at the bottom row of the grid. The mythical creatures and spells are placed randomly on the board, apart from the top and bottom rows. There can only be one animal, mythical creature or spell on each cell at the start. The contents of all cells without an animal is hidden at the start. Spells are never shown and can only be picked up by animals. The mythical creatures never move and the spells once picked up, will not reappear on the board. 10 random spells will be placed on the board at the start of the game.

After setting up the board, player take turns to move each animal.

Gameplay

On each animal's turn, the player can move or not move the animal based on its movement ability. The animal will pickup any spells on the destination square and the player will be informed of what was on the destination square, or if the movement was interrupted. Then the player can choose to use any spells that the animal has. Only up to one spell can be used. Then, the turn moves to the next animal. When the turn moves to the next animal, the current animal's turn will be concluded and any attack damage will be taken. If the animal does not move or use a spell, any attack damage taken will still apply when that animal's turn ends. You can only skip to the next animal's turn if you do not want the current animal to move or use a spell. You cannot skip more than one animal's turn at a time.

The animals take turns in the following order: Rabbit, Fox, Deer, Owl, Badger.

End Conditions

If any animal has no life points left, the game ends and the player loses. If all the animals reach the top row of the board, the game ends and the player wins.

System Specification

You are required to implement the classes shown in the following UML class diagram, including all the public methods and attributes shown. The classes should implement the game rules described above, and should be able to play a game of Woodland Diplomacy using the provided client. Your program must allow communication from the client using HTTP and should provide a HTTP server functionality that can handle these requests.

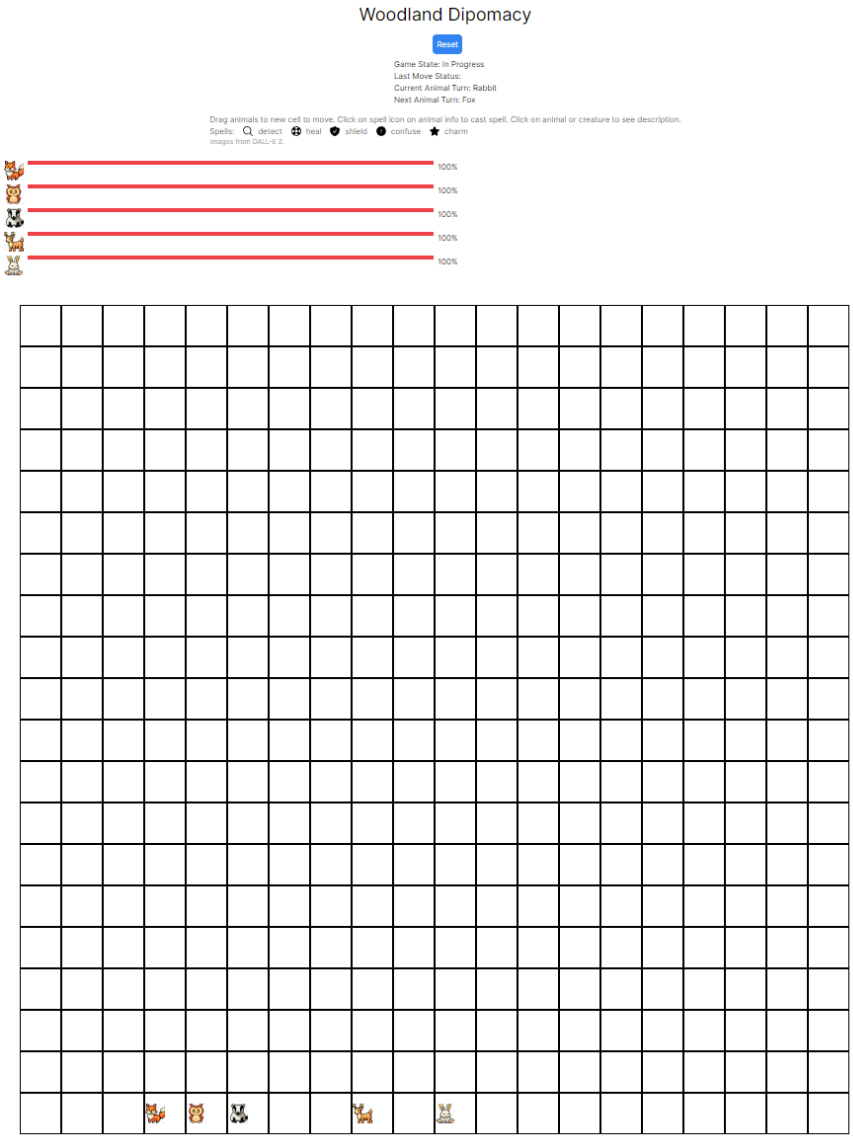

Figure 2: A possible setup at the beginning of the game

The client is at https://stacs5001.github.io/p2-client. The game should be playable using this client.

You program should also pass the provided automated tests. The tests will be released sometime after the specification, so if they are not there yet, please check back later. The tests can be run on the lab machines using *stacscheck*, by navigating into your `CS5001-p2/` directory and running the following command:

```
stacscheck /cs/studres/CS5001/Coursework/p2/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 test files are in subdirectories of the `Tests/` directory.

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.

Board Setup

In order to predictively test your program, the starting board has to be predictable. Therefore, you must use the following method to set up your initial board state.

You need to use the java.util.Random class to create a new Random object. You must initialise the Random generator with the seed passed in through the program arguments. You must then place the animals, then creatures, then spells in the following order:

Animals: Rabbit, Fox, Deer, Owl, Badger

Creatures: Under-Appreciated Unicorn, Complicated Centaur, Deceptive Dragon, Precocious Phoenix, Sassy Sphinx

For each animal, you should call the nextInt with an upper bound of the number of columns to get the column number. The row number should always be the bottom row. If the cell is already occupied, you should try a different column again until you find an empty cell.

For each creature, you should call the nextInt with an upper bound of the number of rows - 2 and add 1 afterwards to get the row number, excluding the first and last row. You should call nextInt again with an upper bound of the number of columns to get the column number. If the cell is already occupied, you should try a different column again until you find an empty cell.

For each spell, you should first call the nextInt with an upper bound of 5 to pick the spell from an array. The order of array you should pick the spell from is: Detect, Heal, Shield, Confuse, Charm. You should then call nextInt with an upper bound of the number of rows - 2 and add 1 afterwards to get the row number, excluding the first and last row. You should call nextInt again with an upper bound of the number of columns to get the column number. If the cell is already occupied, you should try a different column again until you find an empty cell.

You must call nextInt in this order in order for the board state to be predictable.

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. In particular, the HTTP server implementation is not included in the UML and is up to you to implement.

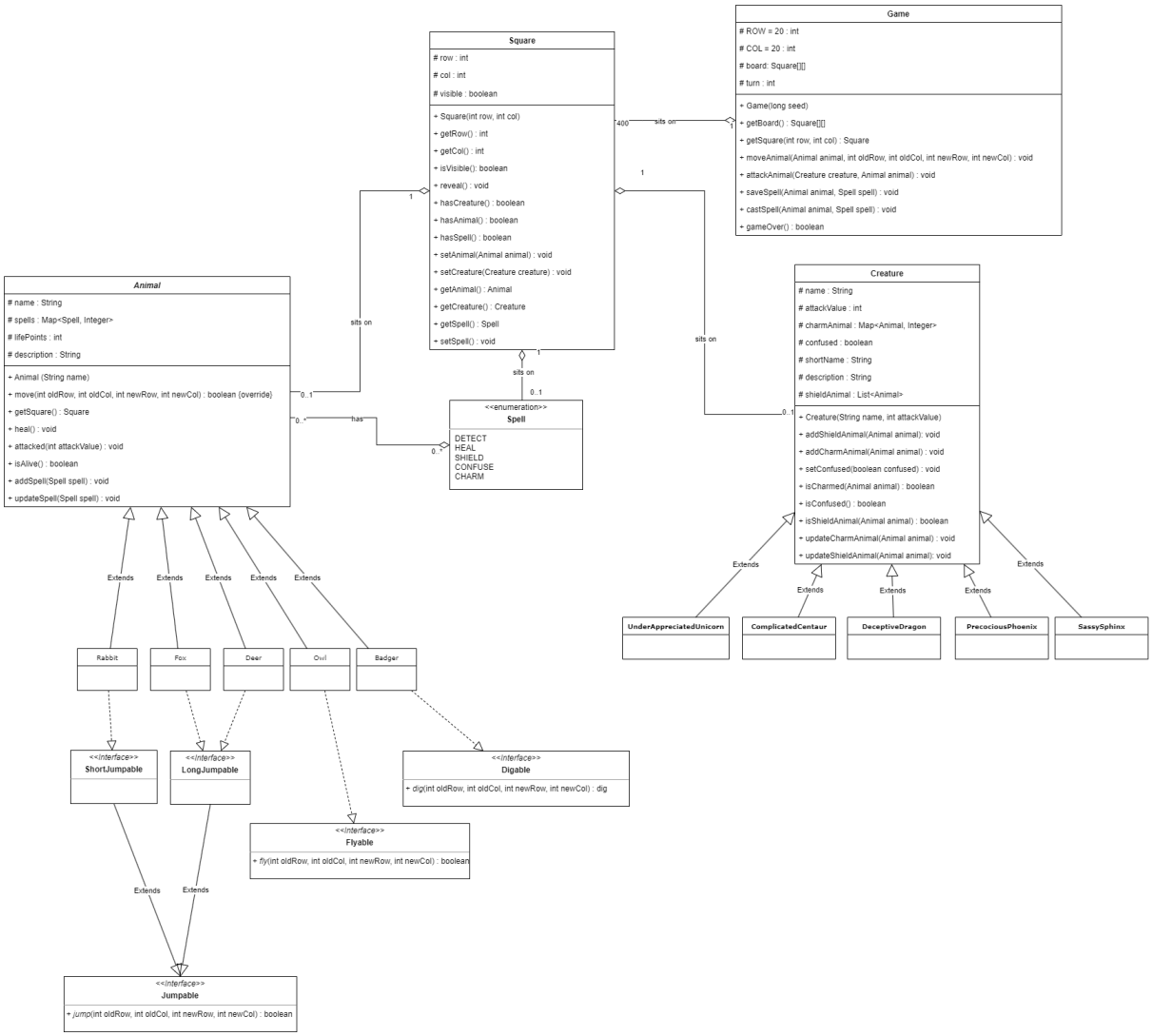

Figure 3: UML class diagram of the required system

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

In this diagram, each box is one class. Method names written in italics are *abstract methods*.

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.

Network Protocol

The server needs to accept HTTP requests from the client. The client is provided to you and should not be changed. The server should accept the following HTTP requests:

- `GET /` - This request is used to check if the server is running. The server should respond with a HTTP response with status code 200 and the body `{"status": "ok"}`.
- `GET /game` - This request is used to get the current state of the game. The server should respond with a HTTP response with status code 200 and the body containing the current state of the game in JSON game format. The JSON game format is described below.
- `POST /game` - This request is used to make a move or use a spell in the game. The body of the request should contain the action in JSON action format. The JSON action format is described below. The server should respond with a HTTP response with status code 200 and the body containing the current state of the game in JSON game format. The JSON game format is described below.
- `POST /reset` - This request is used to reset the game. The server should respond with a HTTP response with status code 200 and the body containing the new state of the game in JSON game format. The JSON game format is described below.

The HTTP server should also be able to accept OPTIONS requests for these endpoints. OPTIONS requests only send the response headers without any body or content.

The OPTIONS request must be implemented as it is required for the CORS preflight request that the client will send.

All responses (GET, POST or OPTIONS) must include the following CORS headers:

- `Access-Control-Allow-Origin: *`
- `Access-Control-Allow-Methods: *`
- `Access-Control-Allow-Headers: *`
- `Access-Control-Max-Age: 86400`

Remember to also include the `Content-Type: application/json` header in all responses that contain a body.

Also, note that to read the body of a POST request, you need to read the Content-Length header first to know how many bytes to read. Then only read that number of bytes from the socket.

JSON Format

The JSON game format is used to represent the state of the game. The JSON game format is as follows (a smaller board, a subset of animals and creatures are represented here for clarity, along with shortened descriptions):

```
{
 "board": [
  [
   [],
   [],
   []
  ],
```

```
[
 [],
 [
  {
   "name": "Fox",
   "type": "Animal",
   "description": "The fox has a bushy tail...",
   "life": 100,
   "spells": [
     {
         "name": "Detect",
         "description": "The detect spell allows the ...",
         "amount": 1
     }
    ]
  },
  {
   "name": "Deceptive Dragon",
   "type": "Creature",
   "shortName": "DD",
   "description": "The DD is a dragon that practices ...",
   "attack": 29,
   "confused": false,
   "charmed" : [
     {
         "animal": "Fox",
         "turnsLeft": 2
     }
    ]
  }
 ],
 []
],
[
 [
  {
  "name": "Rabbit",
  "type": "Animal",
  "description": "The rabbit has a fluffy ears...",
  "life": 100,
  "spells": [
    {
        "name": "Detect",
        "description": "The detect spell allows the ...",
        "amount": 2
    }
   ]
  }
 ],
```

```
    [],
    []
   ]
  ],
  "gameOver": false,
  "currentAnimalTurn": "Rabbit",
  "nextAnimalTurn": "Fox",
  "status": "The last move was successful.",
  "currentAnimalTurnType": "spell",
  "extendedStatus": ""
}
```

The format of the board is a nested array. The top level is the row, the next level is the column (getting the square), and last level is the list of square occupants.

If an action was performed, the status will contain a message describing the outcome of the action. If no action was performed, the status message will be blank. The board state after the action will be the new state after the action.

The status will also show the end result of the game when the game is over.

Table of status messages:

Outcome	Status Message
Move successful	The last move was successful.
Move interrupted	The last move was interrupted by a creature.
Move invalid	The last move was invalid.
Spell successful	The last spell was successful.
Spell invalid	The last spell was invalid.
Game win	You have won the game.
Game lost	You have lost the game.

If you want to provide more feedback to the user, you can use the extendedStatus field and the currentAnimalTurnType field. The currentAnimalTurnType field will be either "move" or "spell" depending on whether it is the animal's turn to move or use the spell next. In error cases, you can use the extendedStatus field to provide more information about the error. You can decide what to put in the extendedStatus field. Any message that is descriptive of the error is fine.

The currentAnimalTurnType and extendedStatus fields are optional and can be omitted.

The JSON action format is used to represent the action that the player wants to make.

The JSON action format is as follows for moves:

```
{
 "action": "move",
 "animal": "Rabbit",
 "toSquare": {
  "row": 2,
  "col": 0
 }
}
```

The row and column of the square is zero-indexed and goes from left to right and top to bottom. The top left square is row 0, column 0. The bottom right square is row 19, column 19.

The JSON action format is as follows for spells:

```
{
 "action": "spell",
 "animal": "Rabbit",
 "spell": "Detect"
}
```

Exception handling

An exception should be thrown if attempts to move an animal onto a square is invalid. 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. Some errors will need to be shown in the JSON output.

Packages

You should also place all your code under the packge: `woodland`. You should make sure to declare this package at the top of each file, with the line `package woodland;` for the top level classes. You should also create more subpackages to structure the code better. 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.

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/` 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.

Test and Debug

In order to test your server manually, you can use the provided client or use a program that sends HTTP requests, like a web browser or Postman.

In your own code, take care to flush sockets and close them when appropriate. Otherwise you may not see the responses made by the server on the client and you may have issues running your server program due to sockets being left open.

You might also consider performing more rigorous automated testing. If you decide to adopt this approach, it is probably a good idea to start by looking at the existing tests on StudRes and follow the same approach. You can create new tests in a local sub-directory in your assignment directory and pass the directory of your own tests to stacscheck when you run it from your assignment directory.

You can learn more about the automated checker at https://studres.cs.st-andrews.ac.uk/Library/stacscheck/.

Marking

Grades will be awarded according to the General Mark Descriptors in the Feedback section of the School Student Handbook. The following table shows some example descriptions for projects that would fall into each band.

Grade	Examples
1–3	Little or no working code, and little or no understanding shown.
4–6	Acceptable code for part of the problem, but with serious issues such as not compiling or running. Some understanding of the issues shown.
7	A running server that accepts connections from clients, and uses sockets to send and receive strings, that implements commands similar to those in the specification and goes some way towards allowing the game to connect and start a game. May have bugs, poor style, and many missing features.
8–10	A reasonably stable server that implements some basic gameplay and implements most of the UML game model. Possibly lacks some of the more complex features of the gameplay, has problems with error-handling and poor style. Some documentation.
11–13	Implements most of the game features in the specification, with reasonable error-handling and acceptable program design and code quality. Some object-oriented design, and few bugs. Reasonable documentation.
14–16	A submission that implements all classes and methods as shown on the UML diagram along with HTTP communication, using object-oriented methods, but which does not have the required behaviour in all cases, could receive a grade of up to 16 if it is implemented cleanly and intelligently, with a well-structured object-oriented design, and very little code repetition. Good documentation.
17–18	Implements all clases and methods in the specification correctly with correct behaviour, with full error-handling including sensible decisions for errors not mentioned in the specification. Clean code in an excellent object-oriented design, with almost no repetition. Excellent documentation.
19–20	Outstanding implementations of all features as above, and possibly some extra features not mentioned in the specification, with exceptional clarity of design and implementation.

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.

Lateness

The standard penalty for late submission applies (Scheme A: 1 mark per 24-hour period, or part thereof):

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/