Practical 2 – OO Implementation

Due Friday, week 5 – weighting 35%

In this practical, you are required to implement an object-oriented model that you are provided with. Your implementation must be consistent with the UML class diagram you are provided, but you will need to decide upon any additional fields, methods, classes, associations, etc. that are needed to meet the requirements.

For this practical, you may develop your code in any IDE or editor of your choice, however, you must ensure you're your source code is in a folder named **CS5001-p2-oop/src** and your classes are named in accordance with the specification provided.

System Description

We are developing a Tower Defence game! 阻止

Tower defence is a genre of games where the goal is to defend the player's territory by obstructing enemy attackers. It is played on a map. Typically, the enemies appear from one side of the map and they advance towards the player's territory at every time step. The player is able to strategically place towers along the way which shoot at the enemies. Each time an enemy is shot they lose health, and eventually they die. If an enemy manages to reach the player's territory, the game ends.

The Map

走廊

The game map is a simple corridor. Enemies appear at position 1, they advance in the corridor to the next positions, until they reach the end of the corridor. If an enemy reaches the end of the corridor, the game is over. The length of the corridor should be a parameter.

Enemies

For the basic requirements, you need to implement two kinds of enemies. These are called *Rat* and *Elephant*. Rats are small and quick, Elephants are large and slow.

Rats advance by two positions at every game step, so they are very quick. Unless defended against, they will reach the end of the corridor, hence the player's territory very quickly. However, they are very small, and they start with a health of 1. Hitting them from a small tower just once would be enough to kill and stop them. 跟走0.5不一样

Elephants advance by one position every two game steps. They are very slow in comparison to rats, however killing them is much harder. They start with a health of 10.

Towers

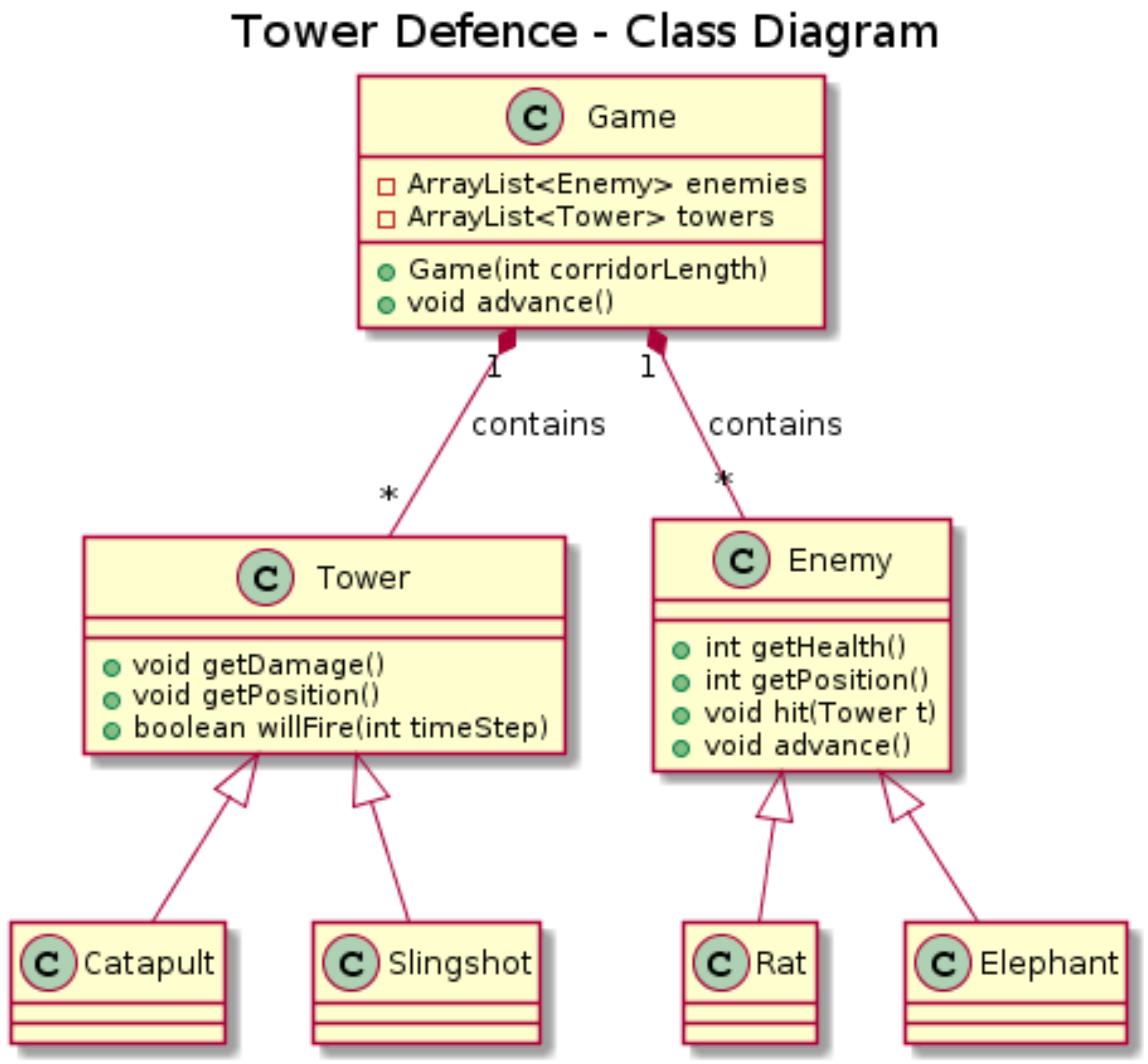
For the basic requirements, you need to implement two kinds of towers. These are called *Slingshot* and *Catapult*.

Slingshot is a small tower, if it manages to hit an enemy it can only damage them by 1 health point. But it is very quick to load again, so it can be used to hit enemies at every game step.

Catapult is a much bigger tower, and it can damage an enemy by 5 health points. However, it takes a while to load the Catapult, so it can only be used once every 3 game steps.

UML

You are provided with a UML class diagram for the basic requirements of this practical. You may add new methods if you need them.



- The Game class contains the length of the corridor, a list of enemies and a list of towers.
- The advance method in the Game class is used to advance the state of the game by one step. This includes:
  - Shooting from the towers. Towers cannot hit enemies that have passed them, so they will only hit an enemy whose position is less than or equal to the position of the tower.
  - Towers shoot as soon as they are loaded. This is indicated by the willFire method returning true. Remember, catapults need to wait a number of time steps before they can shoot again. The parameter to the willFire method is the current time step of the game.
- The getDamage method of the Tower class returns how much damage a given tower makes when they hit an enemy.

- The getPosition method of the Tower class returns the position of a tower. Towers may be created at any position, but they will only hit enemies whose position is less than or equal to theirs.
- The getHealth method of the Enemy class returns the current health level of an enemy. An enemy dies when their health level is 0 (or less).
- The getPosition method of the Enemy class returns the current position of an enemy.
- The advance method of the Enemy class is called at every game step by the advance method of the Game class. This method is responsible for updating the position of the enemy.
- The hit method of the Enemy class is called when a tower hits an enemy. The tower is passed in as a parameter in this case. The health of the enemy should be updated accordingly.
- The Catapult and Slingshot subclasses of Tower, and the Rat and Elephant subclasses of Enemy should override the necessary methods to specify the behaviour of these classes with respect to the constraints given in the system description above.

Basic Requirements

Implement the classes and methods that are in the UML class diagram. Extend the design to include a notion of cost for towers. Each tower costs a certain amount, and a player has a fixed budget. The player spends this amount to place a tower.
Make sure to override the **toString** method for all of your classes, this will help you inspect the state of the game during testing and debugging.

Enhancements

Possible enhancements:

- Extend your design such that the player earns some number of coins every time it manages to kill an enemy. The player can then use these coins to build more towers.
- Add new types of enemies (and towers) with different characteristics.
- Develop a simple terminal-based user interface which will allow a user to play this game. The interface should allow the user to configure the towers at the start of the game, and watch the game unfold with the given tower configuration. The enemies should be introduced automatically (randomly, or not).

Note: When implementing any enhancements, you must ensure that your program still adheres to the system description provided and that all unit tests still pass.

Automated Checking

You are provided with some basic unit tests to check that your code provides the required functionality. The tests can be found at `/cs/studres/CS5001/Practicals/p2-oop/Tests`. In order to run the automated checker on your program, open a terminal window connected to the Linux lab clients/servers and change directory to your **CS5001-p2-oop** directory and execute the following command.

```
stacscheck /cs/studres/CS5001/Practicals/p2-oop/Tests
```

If the automated checker doesn't run, or the build fails, or all tests fail, you may have mistyped a command or not have followed the instructions above.

You should open the provided unit test files and make sure you understand what the tests are doing and try to come up with some interesting tests of your own. You can run your own junit tests in an IDE, via the command line, or via the automated checker. You can create new tests in a local sub-directory in your assignment directory and run **stacscheck** with your own directory as the argument e.g.

```
stacscheck ~/CS5001-p2-oops/MyTests
```

You should also look at the documentation for the automated checker at:

https://studres.cs.st-andrews.ac.uk/Library/stacscheck

Deliverables: Software and Report

For this practical you should write a short report documenting your design and implementation decisions and the justification for those decisions. The structure of your classes and the methods in them are constrained by the UML class diagram that we have provided, you do not need to explain these. You may explain the implementation of these classes and methods, and the design decisions regarding the enhancements.

Include a section in your report which demonstrates the correctness of your solution. This section might include some sample scenarios and outputs produced by your program.

If you have attempted any extensions you should include a short section describing what extensions you have done, including any instructions for compiling, running and using your program.

Hand in a ZIP archive of your assignment folder, including the PDF file containing your report, your **src** directory, and any local test sub-directories, via MMS as usual.

Marking

A very good attempt at satisfying the basic requirements above in an object-oriented fashion can achieve a mark of 14 - 16. This means you should produce very good, re-usable code which makes proper use of inheritance, association, and encapsulation with very good method decomposition. To achieve a 17 or above, your code must in addition make a very good attempt at the enhancements.

See the standard mark descriptors in the School Student Handbook:

http://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):

http://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