Programming Practice (PRP), Coursework Exercise 3 (33%, 30 marks)

Please read the document marked 'Continuous Assessment Guidelines' carefully, before attempting any piece of coursework.

This assignment counts for 33% of your mark for PRP continuous assessment, and is the third of four. If you have not yet completed the first two assignments for PRP, you are advised to do so before proceeding with this one.

The release week for this assignment starts 31st October, at 23:55, and ends 7th November, at 23:55. All submissions must occur before the end of the release week.

If you have any questions about the structure of this assessment, please email martin.chapman@kcl.ac.uk.

1 Problem

This assignment is based on the following problem:

There are three `Soldier`s on a `Battlefield`, each from opposing sides. An `Arbalist`, a `Spearman` and a `Clibanarii`.

A `Battlefield` is a square surface described by points $(\texttt{x}, \texttt{y})$ with $0 \leq \texttt{x} \leq 100$ and $0 \leq \texttt{y} \leq 100$. When a battle begins, the three soldiers are positioned on the battlefield at random points. Note that soldiers do *not* occupy *only* integer locations.

The `Clibanarii` rides on his horse at a `speed` of 5 units of space per unit of time. The `Spearman` runs at a `speed` of 4.3 units of space per unit of time. The `Arbalist` does not

move, but his arrows travel at a `speed` of 12.2 units of space per unit of time and have a `range` that covers 10.6 units of space around the `Arbalist`.

The goal of each soldier is to wound *one* other soldier. The `Arbalist` wants to wound the `Spearman` by shooting an arrow, from distance, through the `Spearman`'s light armour (but does not target the `Clibanarii` due to his heavy armour). The `Clibanarii`, mounted on his horse, wants to ride over to the `Arbalist` and wound him (but does not target the `Spearman`, as he would lose). The `Spearman` wants to run over to a `Clibanarii` and wound him high up on his horse with his long spear (but does not target the `Arbalist`, as he would lose).

Before moving, each `Soldier` has the ability to compute the `distance` to another soldier, and how much `time` it would take either them, or in the case of the `Arbalist` an arrow, to reach the other soldier. For simplicity, a soldier does not take into account that their target may also be moving.

At the start of a battle, once the soldiers are positioned on the battlefield, we want to determine which soldier would reach their target first. The soldier who would reach their target first will wound their target first, and will therefore be considered the winner of the battle. We want to determine who this winner will be.

If the `Arbalist` is closer to the `Spearman` than the `Clibanarii` is to either the `Arbalist` or the `Spearman`, but the `Spearman` is further than 10.6 units of space away from the `Arbalist` (i.e. the arrow would reach the `Spearman` first, but the `Spearman` is out of range), then the next fastest soldier wins (that is the `Spearman` if they are closer to the `Clibanarii`, and the `Clibanarii` if they are closer to the `Arbalist`). This scenario is shown in Figure 1, which also serves as a conceptual overview of a `Battlefield`.

After we have determined who the winner of a battle will be, we then assume that the battle takes place. After the battle, we want to *update* the position of the relevant soldiers on the battlefield based upon the winner. If the `Arbalist` wins the battle, then the wounded `Spearman` disappears from the battlefield (i.e. the position of the `Spearman` is set to some predefined value that is not on the field, such as (-1.0, -1.0)), and the `Arbalist` remains where he is. If the `Clibanarii` wins, then the `Clibanarii` instantly moves to the `Arbalist`'s position, and the `Arbalist` disappears. If the `Spearman` wins, then the `Spearman` instantly moves to the `Clibanarii`'s location but the `Clibanarii` does *not* disappear from the battlefield. In all cases the third soldier is unaffected.

We want the ability to run three battles, and to understand who the winner of the battle is after each iteration.

Figure 1: An `Arbalist`, a `Spearman` and a `Clibanarii`. The `Arbalist` and the `Spearman` are the two closest soldiers, so had the `Arbalist` been in range, he would have hit his target first. However, the `Arbalist` is not in range, so, actually, the `Spearman` wounds the `Clibanarii`, because the `Spearman` is closer to the `Clibanarii` than the `Clibanarii` is to the `Arbalist`. Therefore, the `Spearman` wins.

2 Requirements

Write a set of classes, the use of which enables you to print who would win a battle to the terminal (i.e. which soldier would reach their target first), given the random placement of each soldier. After the winner is established, your program should move the winning and losing soldiers as instructed. This should all occur three times.

3 Mark Scheme

Marks for this assignment will be awarded as follows:

For 0 - 12 marks:

1. Correctly decomposing the problem into a set of classes relevant to the problem.
2. Using these classes to both store and provide access to all information relevant to the problem. This information should be of an appropriate type.
3. Correctly encapsulating all information.

For 12 - 15 marks All of the above, and:

1. Taking the appropriate steps to ensure that all information that is required by each class is always present (e.g. every `Soldier` has a position).
2. Positioning the soldiers on the battlefield at random locations.
3. Providing a suitable String representation of a `Soldier`.

For 15 - 20 marks All of the above, and:

1. Implementing the ability for a `Soldier` to calculate the distance from himself to another `Soldier`.

2. Implementing the ability for a `Soldier` to calculate how long it would take for him to reach another `Soldier`.
3. Computing who will win the battle (excluding the special `Arbalist` case described below).

For 20 - 24 marks All of the above, and:

1. Handling the special case in which the `Arbalist` is closest to, but still out of range of, their target.
2. Moving each soldier to their appropriate locations after the battle is over, according to a calculation of the winner.
3. Running three battles, and identifying three winners.

For 24+ marks All of the above, and:

1. Maximising efficiency through abstraction (i.e. collecting the common features of all `Soldiers`).
2. Computing the winner of the battle in a manner that minimises the use of conditional statements.
3. Returning objects from methods where appropriate, to improve the extensibility of the program.
4. Consistent variable name schemes and capitalisation, and consistent tabbing schemes, in order to promote reusability.
5. Appropriate commenting that explains your code. Javadoc is good practice, but not required.

In addition, note that:

1. The mark scheme given allows for a passing mark to be achieved even if the entirety of the specification is not met.

2. However, code that does not compile will receive a maximum mark of 40%. As with all previous coursework, you must test whether your code compiles on one of the lab computers *outside* of any IDE (i.e. by compiling it from the command line), even if you intend to demonstrate your code to your examiner on your own laptop.
3. The mark scheme for 24 marks and above is not exhaustive. It is at the discretion of us as examiners to reward those students who demonstrate an understanding of reusability, encapsulation, abstraction and decomposition, and who produce their solutions in the most efficient manner.
4. Exception handling and error management are not required for this assignment.
5. **As in previous assignments, your final grade is based upon both the quality of your code and your ability to describe your code to your examiner.**

4 Tips

Tips and useful information for this assignment are as follows:

1. Do not be overwhelmed by the wording of the problem statement. Not all information needs to be captured in your program, only the information that is relevant to the requirements.
2. Keep things simple. One of the model solutions to this assignment is less than 100 lines long (excluding braces, spaces and comments).
3. Do not rush to a computer. This entire problem can be solved using a pen and paper. Try doing this yourself before you start to program.
4. Start of with the easy parts of the problem first. Build what you can and what you are familiar with, and then focus on how these components can be combined and developed in order to solve the harder parts of the problem.
5. Consulting the lab exercises is the *best* way to gain insight into how this problem can be solved.

Once you have completed this assignment, you must place all the code you have produced into a folder, name this folder 'Exercise3', compress it (to one of a .zip, .rar or .tar.gz file, no other formats) and submit it to KEATS. Please note that you should only submit plain text files with a .java extension for assessment (so no proprietary formats such as PDF or Rich Text).