Programming Practice (PRP), Coursework Exercise 3, Mark Scheme (Full Version with Notes)

1 Overview

1. The main focus of this assignment was to test manipulation of **control flow** (conditional statements and loops).
2. The maximum mark that a student can achieve if their code does not compile is 12 / 30 (40%).
3. Students who provide their solutions procedurally should receive a maximum mark of 12 / 30 (40%). Procedural solutions do not use objects at all, solving everything in the main method; create objects but do not use them; or, in the context of this assignment, create objects only to store information *without* calling any methods on those objects to solve the task.
4. Students who do not model the soldier types they were asked to receive a maximum mark of 15 / 30 (50%).
5. There is no need to deduct marks for extra code, but take this into account when awarding final marks for efficiency.
6. For each library class used, it is essential that you establish whether a student understands what that class is helping them to achieve.

2 Mark Scheme

For 0 - 12 marks: 0% - 40%, a passing grade

1. Correctly decomposing the problem into a set of classes relevant to the problem (**2 marks total**).

    - The only requirement, at this low level, is to have *at least* two classes: one that stores information about a soldier (**1 mark**), one that holds the main method (**1 mark**).

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 (**8 marks total**).

    - Required fields:
        - **x location** (**1 mark**), of type **double** (**1 mark**)
        - **y location** (**1 mark**), of type **double** (**1 mark**)
        - **speed** (**1 mark**), of type **double** (**1 mark**)
        - **range** (**1 mark**), of type **double** (may be in subclass) (**1 mark**)
    - If students choose to use a library class like `Point2D` to represent the x location and the y location, ask them to explain how they would have represented the position of a soldier *without* using this class. They should answer saying that they would have used two fields of double type, or something similar. If you are satisfied with their explanation, then award the 4 marks available for the x location and the y location.

3. Correctly encapsulating all information (**2 marks total**).

    - Fields are either **private** or **protected**. If they are public, they are both final and static. (**1 mark**).
    - All information is set to private fields using methods (could be via a constructor, but this is not essential for these marks). If fields are protected, they *must* only be set from inside the class hierarchy, not from outside (i.e. students should not write <`Object`>.<`Protected Field`> anywhere in their code. (**1 mark**).

    Effectively, students should pass if they have been able to apply what they learnt from the superhero assignment to this assignment.

For 12 - 15 marks 40% - 50%

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) (**1 mark total**).

- A constructor that sets the x and the y position, either by passed variables, or by assigning random variables to the x and y fields (may be a call to a separate method) (**1 mark**).

2. Positioning the soldiers on the battlefield at random locations (**1 mark total**).

   - Generating random **double** values between 0 and 100. Inclusive upper value 100 is not required (so a random number between 0 and 99.99... is fine). Example: `Math.random() * 100` (**1 mark**).
   - Random values may be assigned in the constructor, or passed to the constructor when the soldier objects are created.

3. Providing a suitable String representation of a `Soldier` (**1 mark total**).

   - A `toString()` method inside Soldier returning the name of the soldier. May be via a pre-set name field, or using something like `getClass().getSimpleName()` (**1 mark**).

For 15 - 20 marks 50% - 66%

All of the above, and:

1. Implementing the ability for a `Soldier` to calculate the distance from himself to another `Soldier` (**1 mark total**).

   - A method that calculates the distance between two points (information extracted from two soldiers), and uses the standard distance formula to do this. Should be of the form `Math.sqrt( Math.pow ( ( x2 - x1 ) , 2 ) + Math.pow ( ( y2 - y1 ) , 2 ) )`. This isn't necessarily inside the Soldier class, for this mark (**1 mark**).
   - If a student has chosen to use an inbuilt class to calculate the distance between two points (like `Point2D`), ask them to describe how they could have used the `Math` class to make the calculation for them instead. Detailed knowledge of the distance formula is not required, but a reasonable statement about the functionality the `Math` class provides (square root, power etc.) is enough for this mark.

2. Implementing the ability for a `Soldier` to calculate how long it would take for him to reach another `Soldier` (**2 marks total**).

- Dividing the calculation of distance by the Soldier's speed in order to calculate how long it would take the soldier to reach their target (**1 mark**).
- A separation between the distance method and this 'time' method is desirable, but not essential.
- Making the distance calculation and the time calculation *inside* the soldier class. Calling the distance calculation from elsewhere (e.g. a utility method) from inside the soldier class is fine (**1 mark**).

3. Computing who will win the battle (excluding the special `Archer` case described below) (**2 marks total**).

    - A section of code that compares the three times associated with each soldier and determines which is the lowest. This may be done through conditionals, or through another creative way. Efficiency *is not* a concern at this point (**2 marks**).

For 20 - 24 marks 66% - 80%

All of the above, and:

1. Handling the special case in which the `Archer` is closest to (in time), but still out of range of, their target (**1 mark total**).

    - An additional check for range that ensures the Archer is only chosen as the winning soldier if he is in range (**1 mark**).
    - There are lots of different ways to achieve this but students should effectively check that the distance between the Ranged Unit and the Spear Unit is less than the Ranged Unit's range.

2. Moving each soldier to their appropriate locations after the battle is over, according to a calculation of the winner (**2 marks total**).

    - Moving the soldiers based upon a calculation of the winner (**1 mark**).
    - *Correctly* moving the soldiers based upon the following rules from the specification (**1 mark**):
        - If the Ranged Unit wins, he stays still and the Spear Unit is moved off the field to (-1, -1).
        - If the Spear Unit wins he moves to the Mounted Unit's position, and the Mounted Unit *stays on* the field.

- If the Mounted Unit wins he moves to the Ranged Unit's position, and the Ranged Unit is moved off the field to (-1, -1).

- This functionality may be spread throughout different methods.
- The idea here is for a student to recognise not only when behaviour is *common* to an object like a Soldier, but also when behaviour is unique to different specialisations of that object. In other words, some behaviour cannot be collected into a superclass (e.g. not all Soldiers are moved off the battlefield when they lose).

3. Running three battles, and identifying three winners (**1 mark total**).

   - A loop that ensures all the code to position soldiers randomly, calculate the winning soldier and move the soldiers after a battle occurs three times. Defining three objects to achieve, within which some battle functionality is contained, is fine (**1 mark**).

For 24 marks - 30 marks 80% - 100%

All of the above, and:

1. Maximising efficiency through abstraction (i.e. collecting the common features of all `Soldiers`) (**1 mark total**).

   - The creation of three subclasses to represent the individual soldiers, placing the range field inside the Ranged Soldier subclass (**1 mark**).
   - Inheritance must achieve something. To gain this mark it is not sufficient to simply create the three subclasses. Students must also place range inside the Ranged Soldier class. It might also be nice to set the speed inside each subclass, but setting speed external to the class is also fine (although I think the former is preferable).

2. Computing the winner of the battle in a manner that minimises the use of conditional statements (**1 mark total**).

   - There are no superfluous checks (i.e. repeated conditions) when computing who the winner of the battle should be (**1 mark**).
   - This entry on the mark scheme should also inspire students to move towards more reusable code, which is rewarded later on.

- There is a difference here between *minimising* the use of conditional statements and having *minimal* conditional statements. Be careful here not to penalise students who use extra conditional statements in an effort to provide a reusable solution (e.g. to perform a sort of an array of soldiers, or to find the lowest time in a loop), and only penalise those who *repeat* conditional statements.
- This should be your opportunity to reward efficient solutions.

3. Consistent variable name schemes and capitalisation, and consistent tabbing schemes, in order to promote reusability (**1 mark total**).

   - Code looks good visually in that, for each class, it is clear which code sits within the class itself, which code sits within each method, and which code sits within conditional statements and loops. Variable names must represent the data held within that variable (**1 mark**).

4. Appropriate commenting that explains your code. Javadoc is good practice, but not required (**1 mark total**).

   - There are a sufficient number of comments in the program that explain how the code operates (**1 mark**).

5. Writing code in a manner that promotes reusability and extensibility (**2 marks total**).

   - This original mark scheme entry was more specific. I have generalised it to reward a number of different ways in which students may consider making their code reusable. Reward up to 2 marks for any two of the following:
     - The original mark scheme entry: 'Returning objects from methods where appropriate, to improve the extensibility of the program' (**1 mark**). This could be something like a 'TargetTime' object that represents some information about the battle.
     - Calculating the winner of the fight in an inventive way, that is likely to allow for *multiple* soldiers to be compared in the future (e.g. using a loop to order the soldiers in an array, using a loop to get the lowest time taken etc.) (**1 mark**).
     - Having a clearly structured class, perhaps called Battle or Battlefield, that contains a set of methods pertinent to a battle: calculate winner, position soldiers, end battle etc.
     - Using a location class to store information about a soldier's position (**1 mark**).
     - Using a utility class to store a reusable set of static methods (**1 mark**).

– Providing multiple constructors to create Soldiers in different ways (e.g. using a supplied x and y location, or by positioning the soldiers randomly) (**1 mark**).
– Storing the dimensions of the battlefield as variables that can be changed in order to have different size battlefields (**1 mark**).
– This list is not exhaustive.