Practical 2 – OO Design and Implementation

Due Friday, Week 5 – weighting 35%

In this practical you are required to design and implement a simple game. Several details of the object-oriented design are provided in this practical sheet. You must produce a UML class diagram that corresponds to the design and produce an implementation that is consistent with the UML diagram. You may need to add 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 text editor of your choice, however, you must ensure that your source code is in a folder named **CS5001-p2-oop/src** and your classes are named according to the specification provided.

System Description

We are developing a game!

This is a single player strategy game that is played on a rectangular grid. Before the game starts, the player places items on the cells of the grid and chooses a number of time-steps to run the game. There are three kinds of items in the game: farmers, transporters, and consumers. Farmers produce a single kind of product, transporters carry the products from the farmer to the consumer and the consumers consume the produce. There are limits to the amount of production, number of products that can be carried, and number of products that can be consumed by the eaters. So, some products may be produced only to be wasted due to these limits.

Once the items are placed on the grid, the player will not interact with the game: the game runs automatically and produces a final score. The score is the ratio of products consumed over products produced.

If this brief description is confusing, don't worry, keep reading. The next section describes the kinds of farmers, transporters, and consumers in the basic game. The following section gives some example game simulations.

Item - Farmers

There are two kinds of farmers: corn farmers and radish farmers. Each farmer produces only one kind of product.

- Corn farmers require 2 spaces to their left and right and 1 space below and above where there are no other farmers. If this condition is not met, they do not produce anything.
- Corn farmers produce 5 corns every 4 time-steps. The game starts at time-step 1 and runs for a number of time-steps that is provided by the user.
- Each corn provides 5 units of nutrition.

- Radish farmers require less space around them, they produce radishes as long as there isn't an adjacent farmer next to them. So just 1 space in each four directions (excluding the diagonals) without farmers.
- Radish farmers produce 10 radishes every 3 time-steps.
- Each radish provides 1 unit of nutrition.

Item - Consumers

There are two kinds of consumers who are ready to eat both corns and radishes. They are rabbits and beavers. They both need the products to be delivered to them before they can eat.

- A rabbit can consume up to 8 units of nutrition at each time-step. If they are delivered any more than this, they cannot save the products for later consumption, so the excess is wasted.
- A beaver can consume up to 5 units of nutrition at each time-step. In addition, they can store up to 50 units for later consumption. Any more than 50 units will be wasted.

Item - Transporters

The last kind of item are transporters, which carry products from farmers to consumers. There are two kinds of transporters, horizontal and vertical.

- Horizontal transporters move products between a farmer and a consumer when they are placed on the same row and between the two.
- Vertical transporters are similar, but they need to be on the same column between a farmer and a consumer.
- If the horizontal and vertical transformers are not placed correctly (i.e. if they are not between a farmer and a consumer) they do not carry anything.

Examples

As an example consider this game grid with 5 rows, 3 columns. There are 5 items on this grid: 2 farmers, 1 consumer, and 2 transporters.

```
----------------------------------------------
|   Corn(0)     |       HT      |  Rabbit(0)    |
|               |               |               |
|               |               |      VT       |
|               |               |               |
|               |               |  Radish(0)    |
----------------------------------------------
```

A method called `AbstractGrid.display()` is provided which will display a game grid using this notation.
In this example the corn farmer on the top-left will start producing corn, as well as the radish farmer on the right-bottom. HT and VT stand for horizontal and vertical transporters respectively and they will carry the produce from the farmers to the rabbit. The numbers

within parenthesis indicate the number of nutritional units that are currently store at that grid cell. Initially these are all zero, of course.

The output for running this game for 5 steps is given in
https://studres.cs.st-andrews.ac.uk/CS5001/Practicals/p2-oop/examples/rabbit-5.out

There are a few other examples in the same directory on StudRes and a number of stacscheck test cases that range from very simple to more and more complex. Study these examples to get a better feeling of the game.

Gameplay

The game runs on a given initial grid configuration for a number of time-steps. At each time-step, all items on the grid are processed using the **AbstractItem.process()** method. This method is abstract and will be implemented by all item types.

At each time-step the process methods are called in the following order: firstly all farmers, secondly all transporters and finally all consumers are processed. The order of processing within each group is performed systematically starting from top-left, processing a row completely before moving to the next row. After each time-step the board is displayed together with some stats. This code is provided in the **Game** class.

Each item class must overload their **toString** methods to match the output given in stacscheck.

Basic Requirements

Draw a UML class diagram that corresponds to this practical specification. Implement the classes and methods that are in the UML class diagram.
Four classes are provided at the following location
https://studres.cs.st-andrews.ac.uk/CS5001/Practicals/p2-oop/Resources/src
You may want to start from the **Game** class, which expected demonstrates the use of several other classes. Do not edit the provided files.

See the following Java class with a sample main method
https://studres.cs.st-andrews.ac.uk/CS5001/Practicals/p2-oop/Tests/1_3/corn/rabbit/5/Test.java

Extensions

Possible extensions:

- Add a new kind of transporter called NearestTransporter. Instead of only working on a single row or column, this transporter should carry products between the nearest farmer to it and the nearest consumer to it. If there is ambiguity about which farmer or consumer to choose (i.e. two farmers or consumers are equidistant to the transporter) the transporter should not carry anything.
  We provide stacscheck tests for this enhancement. See the Automated Checking section below.

- Add new kinds of farmers and consumers to make the game more interesting!
- Add a graphical user interface to replace the provided command line interface.
- Make the game interactive in some way.
- Develop an algorithm that configures the game grid in a way that maximises the score and the total consumed nutrition.

These are only suggestions, you are also free to implement your own extensions. If you attempt any extension, clearly explain these in your report. If the extensions change the basic functionality, submit them in a separate "extension" folder inside your ZIP file.

Automated Checking & Marking

This submission will be tested by the School's automated checker stacscheck so make sure that your software can run the provided tests by running the following command on a lab machine:

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

The automated tests for this practical are organised in a way to allow subtests to be executed independently. For example, to run all basic tests (excluding extension 1) run the following command:

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

Or, to run an individual test you may run:

```
stacscheck /cs/studres/CS5001/Practicals/p2-oop/Tests/ basic/1_3/corn/rabbit/500
```

Passing or failing automated tests does not automatically affect your mark, but make sure stacscheck can run because it is an important tool and it helps the markers provide fairer and more detailed feedback. Your submission will be marked using 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):
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:www.st-andrews.ac.uk/students/rules/academicpractice/