University of
St Andrews

School of Computer Science

CS5001
OO Design and Implementation

Parts completed:

- Basic requirements
- Nearest transporter extension
- New Farmer and Consumer types added (Rotational crop farmer, and migration consumer)
- Interactive console user interface (UI) implemented

ID: 170018405

Table of Contents

Table of Figures

1.0 Program requirements

The purpose of this practical was to develop a farm simulation game using object oriented programming techniques. Four classes were provided as base resources for this game. Those were: AbstractGrid, AbstractItem, Game, and TimeStep. The former two classes provided the foundations for the game implementation. The AbstractGrid and AbstractItems abstract classes provided the structure of the game via abstract methods which needed to be implemented. The AbstractItem class provided the foundations for building any item that can be placed on the game grid. The three main items required to be implemented were Farmers, Transporters, and Consumers. The AbstractGrid class on the other hand provided the foundations for building a grid that can meet all the required functionality of the game grid.

2.0 Grid implementation

The majority of the methods required for the grid implementation were straight forward. However, given that most methods required input of coordinates, and those methods were public, it was necessary to check the coordinates used as inputs to prevent array index out of bounds exceptions. This was achieved via the use of the validCoordinate method which returns a Boolean indicating if any set of coordinates are within the grid dimensions. Using this method, it was possible to limit incorrect access of the grid array during the game runtime especially for the interactive console UI developed as part of the extensions (although this UI prevents users from entering invalid coordinates itself). Moreover, methods that took nutrition values as inputs needed to make sure that negative nutrition values were not passed as parameters as this could be a source of bugs if a user of the grid class didn't check for negative nutrition values.

Another place of discussion regarding the Grid class is the processItems method which is intended to be called upon every game turn or timeStep to process the state of all the items on the grid. The functionality of this method required that all farmers be processed first, then the transporters, then the consumers starting from the 0,0 location and iterating row by row until the end of the grid array. If this functionality were to be carried out using for-loops, then it is to be expected that three complete iterations over the whole grid are required to process the three item types sequentially. This is clearly highly inefficient especially given that a game grid could be very large. To improve the processing time of the game, it was determined that ArrayLists should be used to store the locations of any Farmer, Transporter, or Consumer objects added to the grid. The process items method could then just iterate over each ArrayList one after the other to process the states of the items. This eliminated the need to check if an item exists (at a location), if that item is an instance of a farmer/transporter/consumer three separate times during the game runtime.

This solution of course came with its own set of drawbacks. Firstly, whenever a new item was added to the grid, the item also needed to be added to the list, and if an item (not necessarily of the same type) with the same coordinates was already present in any of the lists, then that needed to be removed for a successful overwrite. This meant that the equalsTo method needed to be overloaded for all three item types to give access to the ArrayList contains method. Secondly, given that the required implementation dictates a very specific order in processing the items (left to right, from top to bottom), the transporters list needed to be sorted whenever an item was added or removed. Sorting was achieved via the Collections sort method. To make use of this method, the Transporters class implemented the comparable interface and overloaded the compareTo method.

The above drawbacks could be argued to lead to a slow implementation for games with small grids. However, it must be considered that the manipulation of the ArrayLists (including the

sorting process) only takes place during the addition of items to the grid. Once the game is actually initiated, these steps are not repeated.

Given that the registerItem method need not be called during run-time, it also made sense to update if farmers still have spaces or if transporters still had valid routes after each item addition rather than checking for these conditions within the respective item's process methods which would impact performance unnecessarily.

A final point of interest regarding the design of the Grid class was the implementation of the removeItem method. This method was implemented to allow a user to remove items mid game if playing the game using the console UI provided. The removeItem method does not need to re-sort the transporters list unlike the register items method, however, it still needed to re-evaluate if farmers have increased or decreased space to farm, and if transporters can transport items and if their routes should be change since a blockage could be removed or added. Given that the method just requires the coordinates of the item to be removed as parameters, a dummy item object of type AbstractItem is created to allow for the comparison of the coordinates between this object and those in the Arraylists. These dummy objects are instances of the ComparisonItem class.

3.0 AbstractItem inheritance

For the three main items types (Farmers/Transporters/Consumers), three abstract classes were implemented which inherited the AbstractItem class. These three abstract classes provided the basic shared functions and fields for their item types. For-example, the Farmer class provides fields to store the production capacity, interval between production cycles, and the space requirements to farm. All subclasses of the Farmer class share these needs, hence, why an abstract Farmer class was created.

Similarly, an abstract Transporter class which inherits the AbstractItem class was created. In addition to providing shared functionalities (like the Farmer and Consumer classes), this class also defines two abstract methods which any subclass of this class needs to implement. Those functions are setFarmer, and setConsumer. These functions are different for horizontal, vertical, and nearest transporters since the algorithms required to search for Farmer and Consumer objects will be different. Yet, these functions are required to update the routes when Transporter objects are inserted or removed from the grid. The power of an abstract class is hence utilised best in this example.

3.1 Notes on encapsulation

Farmer class

The production interval, vertical, and horizontal space requirements were set to private as setting any of these fields to negative integers will cause the farmer process and checkSpace methods to function incorrectly or throw errors. Setter methods are provided to prevent setting these values to negative numbers.

On the other hand, the production capacity and hasSpace fields were set to private for two different reasons. For the former, a negative production value will not be processed by the grid nutrition setter or recording values, so there was no need to restrict access to this filed. On the other hand, the hasSpace Boolean was set to protected because it is likely that a class could inherit the Farmer class that implements some sort of positive or negative effect on the farmer as the game progresses which could affect if a farmer can produce stock or not. For example, if a farmer uses fertilisers for a few turns, then the space requirements could be overturned, and this can be achieved manually by setting hasSpace to true. On the other hand, consider if a storm or natural disaster were to take place, setting hasSpace to false will prevent the farmer from producing stock. Note: the setVSapce and setHSpace methods may also be used in

conjunction with the checkSpace method to achieve the same thing. Moreover, a public method to change the interval is provided in case bad whether causes delays in production.

Transporter class

The transporter class encapsulates all its fields however setter methods are provided.

Consumer class

The consumer class only has a single field – consumption. This field is set to private as a negative consumption value will not result in increased stock as the grid methods check for this. Therefore, there is no need to restrict access to this field to any subclass.

4.0 Extensions and testing

In addition to the basic requirements of this assignment, three extensions were completed.

The first extension completed was the nearest transporter implementation. This implementation can be found in the NearestTransporter class as described in the assignment brief.

The second set of extensions completed involved the implementation of a new kind of farmer, and a new kind of consumer.

The new farmer type is called a rotational crop farmer. This farmer rotates crops based on the production interval of each crop. The first crop is corn, the second is radish, and the third is potato. The corn and radish crops have the same production capacities and intervals as the regular corn and radish farmer with the exception that a rotational crop farmer requires 2 spaces horizontally and vertically to produce the crops. The potato crop on the other hand provides 40 units of nutrition and takes 6 time steps to produce.

To test this new farmer class, a simple test class that creates a grid and places the farmer on it was created. The production units and intervals are then monitored to ensure they match up with the specifications. Furthermore, the toString method is tested by observing the name of the crop on the grid change. Further tests were then carried out to ensure the space requirements worked as intended. However, given this logic is carried out by the Farmer class, no change in expected behaviour was expected and none were observed.

The new consumer type created is the migrating buffalo consumer. This consumer can consumer upto 80 units of nutrition. However, unlike beavers and rabbits, this consumer migrates and returns after 10 time steps. In between this time, any amount of nutrition can be deposited at the migrating buffalo location, however, 10% of all nutrition will rot each turn. This punishes a player who transports a lot of nutrition to this location.

A basic test to test the reduction in nutrition per turn was created and can be found in the MigratingBuffaloTest class. To run this test, simply compile it and run it with the command:

- java MigratingBuffaloTest

The third extension completed was the implementation of a console UI to allow users to play the game interactively. This UI not only served as a basic implementation of a UI to play the game, but it also provides a new way to test all functionalities of the game.

For example, to test the farmer spacing, item registration, and item removal implementations, the following example steps can be used:

1. Launch the UI with a command to create a 3 by 3 grid
2. Insert a corn farmer at 1,1
3. Insert a corn farmer at 1,2

4. Proceed 5 game turns – no production should be observed
5. Remove farmer 1,1
6. Proceed 5 more turns – farmer 1,2 should now produce stock
7. Re-insert a corn farmer at 1,1
8. Proceed another 5 turns – the production should be halted once again.

To test the transporter implementation:

1. Launch the UI with a command to create a 3 by 3 grid
2. Insert a farmer of any type at 1,1
3. Insert horizontal transporter at 2,1
4. Insert a rabbit at 3,1 and at 4,1
5. Proceed 5 turns – the transporter should have transferred nutrition to the closes rabbit
6. Remove the transporter
7. Process another 5 turns – the production should have increased, but no stock is transferred from the farm
8. At this stage the horizontal transporter may be reinserted
9. The transporter should begin transporting upon the next turn
10. If the closes consumer is removed, the transporter should move the stock to the next consumer.

The code for this UI can be found in the ConsoleUI class. To run the UI, compile it and use the following command followed by two command line arguments for the grid height and width:

- java ConsoleUI 2 3

Note: the above command will launch the game with a 2 by 3 grid.

Figure 1 shows the main menu for the console user interface of the farm-simulator game. The user interface requests an integer to be typed into the command line to perform actions such as inserting or removing at item, showing the grid, and proceeding to the next turn.

```
*** This is a console UI for the Farming-sim game ***

1. Insert Farmer
2. Insert Transporter
3. Insert Consumer
4. Show grid
5. End turn
6. Remove Item
7. Exit Game

Enter choice [1-7]:
```

Figure 1 - Console user interface for interactive game

5.0 UML

The UML diagram for the complete implementation of the game can be found on the next page. A high resolution image can also be found within the directory of this report. The UML diagram follows the specifications presented in the lectures. In addition to those, constructors are marked with <<constructor>> tags.

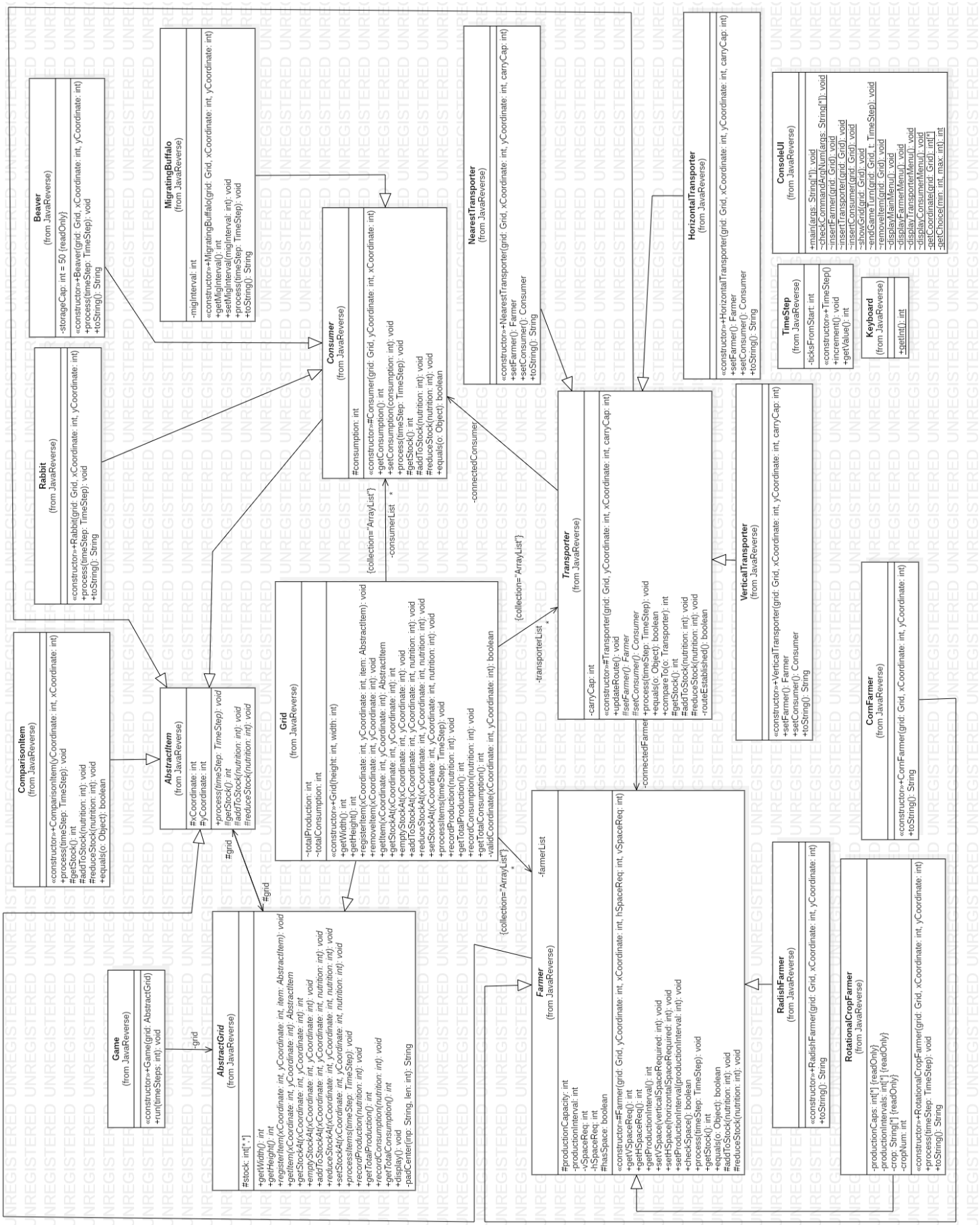

Figure 2 - UML diagram