CS5001 P2 OOP Report

This project was to make a tile-based game that would follow a set of rules to produce, transport and consume stock in Java.

Design

The design of this game was dictated by four Java classes that were provided by the school. Two the classes are abstract, requiring implementation. The other two would be used to run the game and control the length of the game. The first abstract class (AbstractGrid.java) would require a simple implementation (Grid.java). The Grid class controls where the items for the game are and how much stock they have. AbstractItem.java, the second abstract class, would require implementations for each of the item types (farmer, transporter and consumer). The initial idea was to attempt to create Farmer.java, Trasnporter.java and Consumer.java (where each of the subtypes would be controlled by a class level enum. However, after implementation issues (discussed in the following section), the project design changed to its current form (reflected by the UML diagram in Figure 1). This involves having 6 implementation of AbstractItem.java which represent the 6 types and subtypes.

Whilst this design is very monolithic and not the easiest to maintain (due to a lot of very similar methods across the 6 item classes) I could not see an elegant solution around it. This is because the contructors that were expected from the stacscheck test files required there to be a HorizontalTranporter class and a VerticalTransporter class.

Implementation

This project was implemented using a Test Driven Development methodology. For each class I would create I would firstly write JUnit4 tests that were set up to run after each build of the project. The JUnit tests were written before the class methods were implemented. This allowed me to consistently check for ArrayOutOfBounds, NullPointerExceptions and negative number handling during development. For the sake of argument, the assumption was made that any negative number passed into the system by the user or the system itself would be in error, and therefore, should be mutated to a positive integer and then recursively passed into the method that the negative number was trying to utilise.

The first attempt of implementation was to create a single class for each item type (as discussed in the design section above). This did not work due to the expected constructors for each item type. After re-engineering the existing code into a format that met the expectations of the stacscheck tests.

Post re-engineering, the next implementation challenge was to ensure that all item class rules regarding stock were being adhered to at all times. This challenge arose because of the transporter items mutating the stocks of other objects. This was solved after enforcing a system wide rule that the grid’s addToStock and reduceStock methods would utilise the item implementation of the stock manipulation methods wherever possible.

The final challenge of note with this project was to ensure that the transporter classes were refusing to transport any stock across the grid if there was a failed transport condition prior to the a successful transport condition. For example; a scenario ocurred in the stacscheck that a row would be [Corn, __ , HT, __, Beaver, __, HT, __, Corn]. The first HT would empty the first Corn of stock, the second HT would see there was no stock left in the first Corn object and ignore the second.

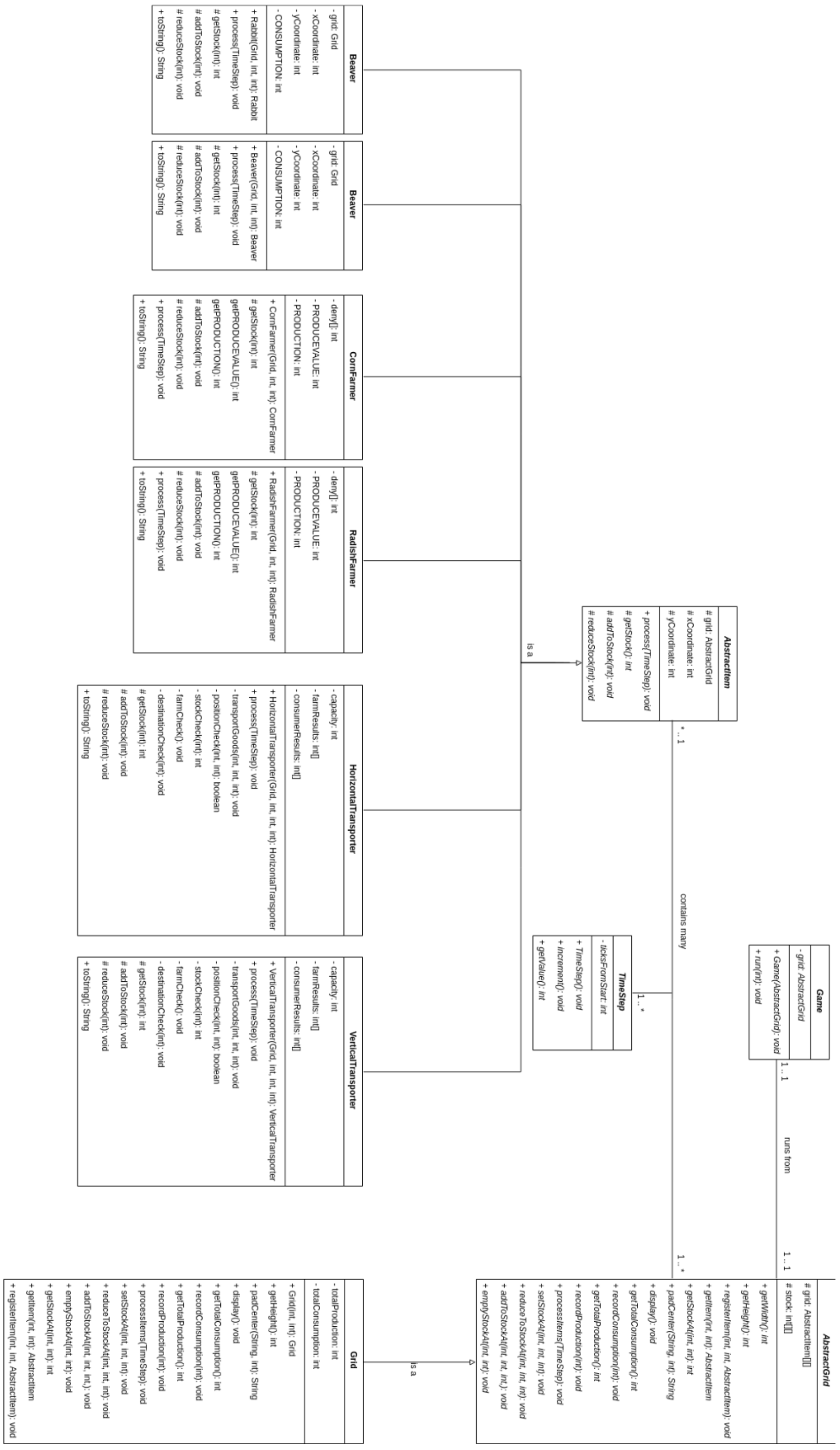
The solution to this problem was to have the transporter objects find all farmer objects with stock in the row. Then look on the other side of the transporter for a consumer. The closest consumer would get the stock.

Testing

To test this project, Test Driven Development was used from the start to finish. The highest code coverage as possible was aimed for. Each method was tested for normal, edge and failed cases wherever possible. In total the test coverage is 94% of lines of code covered by the JUnit tests.

Additionally, the stacscheck basic tests were ran on the project at regular intervals. The project passed 43/43 tests.

Figure 1: UML Class Diagram for Coursework Project


Beaver
- grid: Grid
- xCoordinate: int
- yCoordinate: int
- CONSUMPTION: int
+ Rabbit(Grid, int, int): Rabbit
+ process(TimeStep): void
# getStock(int): int
# addToStock(int): void
# reduceStock(int): void
+ toString(): String
Beaver
- grid: Grid
- xCoordinate: int
- yCoordinate: int
- CONSUMPTION: int
+ Beaver(Grid, int, int): Beaver
+ process(TimeStep): void
# getStock(int): int
# addToStock(int): void
# reduceStock(int): void
+ toString(): String
CornFarmer
- deny[]: int
- PRODUCEVALUE: int
- PRODUCTION: int
+ CornFarmer(Grid, int, int): CornFarmer
# getStock(int): int
getPRODUCEVALUE(): int
getPRODUCTION(): int
# addToStock(int): void
# reduceStock(int): void
+ process(TimeStep): void
+ toString(): String
RadishFarmer
- deny[]: int
- PRODUCEVALUE: int
- PRODUCTION: int
+ RadishFarmer(Grid, int, int): RadishFarmer
# getStock(int): int
getPRODUCEVALUE(): int
getPRODUCTION(): int
# addToStock(int): void
# reduceStock(int): void
+ process(TimeStep): void
+ toString(): String
HorizontalTransporter
- capacity: int
- farmResults: int[]
- consumerResults: int[]
+ HorizontalTransporter(Grid, int, int, int): HorizontalTransporter
+ process(TimeStep): void
- transportGoods(int, int, int): void
- positionCheck(int, int): boolean
- stockCheck(int): int
- farmCheck(): void
- destinationCheck(int): void
# getStock(int): int
# addToStock(int): void
# reduceStock(int): void
+ toString(): String
VerticalTransporter
- capacity: int
- farmResults: int[]
- consumerResults: int[]
+ VerticalTransporter(Grid, int, int, int): VerticalTransporter
+ process(TimeStep): void
- transportGoods(int, int, int): void
- positionCheck(int, int): boolean
- stockCheck(int): int
- farmCheck(): void
- destinationCheck(int): void
# getStock(int): int
# addToStock(int): void
# reduceStock(int): void
+ toString(): String
is a
AbstractItem
# grid: AbstractGrid
# xCoordinate: int
# yCoordinate: int
+ process(TimeStep): void
# getStock(): int
# addToStock(int): void
# reduceStock(int): void
* .. 1
contains many
TimeStep
- ticksFromStart: int
+ TimeStep(): void
+ increment(): void
+ getValue(): int
1 .. *
Game
- grid: AbstractGrid
+ Game(AbstractGrid): void
+ run(int): void
1 .. 1
runs from
1 .. 1
1 .. *
AbstractGrid
# grid: AbstractItem[][]
# stock: int[][]
+ getWidth(): int
+ getHeight(): int
+ registerItem(int, int, AbstractItem): void
+ getItem(int, int): AbstractItem
+ getStockAt(int, int): int
+ padCenter(String, int): String
+ display(): void
+ getTotalConsumption(): int
+ recordConsumption(int): void
+ getTotalProduction(): int
+ recordProduction(int): void
+ processItems(TimeStep): void
+ setStockAt(int, int, int): void
+ reduceToStockAt(int, int, int): void
+ addToStockAt(int, int, int): void
+ emptyStockAt(int, int): void
is a
Grid
- totalProduction: int
- totalConsumption: int
+ Grid(int, int): Grid
+ getHeight(): int
+ padCenter(String, int): String
+ display(): void
+ getTotalConsumption(): int
+ recordConsumption(int): void
+ getTotalProduction(): int
+ recordProduction(int): void
+ processItems(TimeStep): void
+ setStockAt(int, int, int): void
+ reduceToStockAt(int, int, int): void
+ addToStockAt(int, int, int): void
+ emptyStockAt(int, int): void
+ getStockAt(int, int): int
+ getItem(int, int): AbstractItem
+ registerItem(int, int, AbstractItem): void