SWEN30006 Project 1 Report

Monday Workshop, Team #9
Min Thu Han, Arja Das, Yiyan Zhang

P1:

In elevating the current design of the game “Ore Simulator” using the GRASP (General Responsibility Assignment Software Patterns/Principles) guidelines, it is evident that the existing project architecture contains several areas of concern, particularly relating to low cohesion, high coupling, suboptimal encapsulation, and limited use of polymorphism.

The first of many issues is the vast range of responsibilities the classes currently have. For example, ‘OreSim’ has to handle UI management, such as drawing the board and actors. It also had to manage the game state, where it has to handle running the application, updating statistics, and dealing with the auto mode, ‘OreSim’ also had direct control of the game entities, thus tightly coupling them, as any changes that had to be made required having to change the code of ‘OreSim’.

Following on from the point of ‘OreSim’ handling game logic (movements and interactions) and UI logic (drawing the board and actors), This mix of logic leads to low cohesion as the class has no clear purpose, making it difficult to manage and extend. Another problem are the ‘runApp’ and ‘autoMoveNext’ functions as they handle too many aspects of the simulation, as they have to handle timing, actor movements, and do checks for game completion.

There is also an issue with ‘OreSim’ being directly responsible for the instantiating and management of classes such as ‘Pusher’, ‘Ore’, and ‘Target’. This leads to a design where too much of the control is in ‘OreSim’, these types of responsibilities should be done only by classes that have the relevant information. Likewise , the ‘runApp’ method acts as a controller for too many actions, these responsibilities should be spread to other classes that are solely responsible for their respective actions.

The current design also lacks Polymorphism, there should be super classes or interfaces in the design. This can be seen as there are likely to be extensions in the future that will share common functionalities. For example, ‘Pusher’ and the future implementation of ‘Bulldozer’ will likely share common functionalities like moving, or moving objects. Instead the specific behaviors are embedded in ‘OreSim’, reducing flexibility and the reusability of the code.

P2:

In refactoring the code of the “Ore Simulator” as depicted in the provided UML diagram seen below, by aligning it with the GRASP principles, we decided to address the several concerns addressed above in part one, especially those that are related to low cohesion, high coupling, suboptimal optimisation, and the limited use of polymorphism.

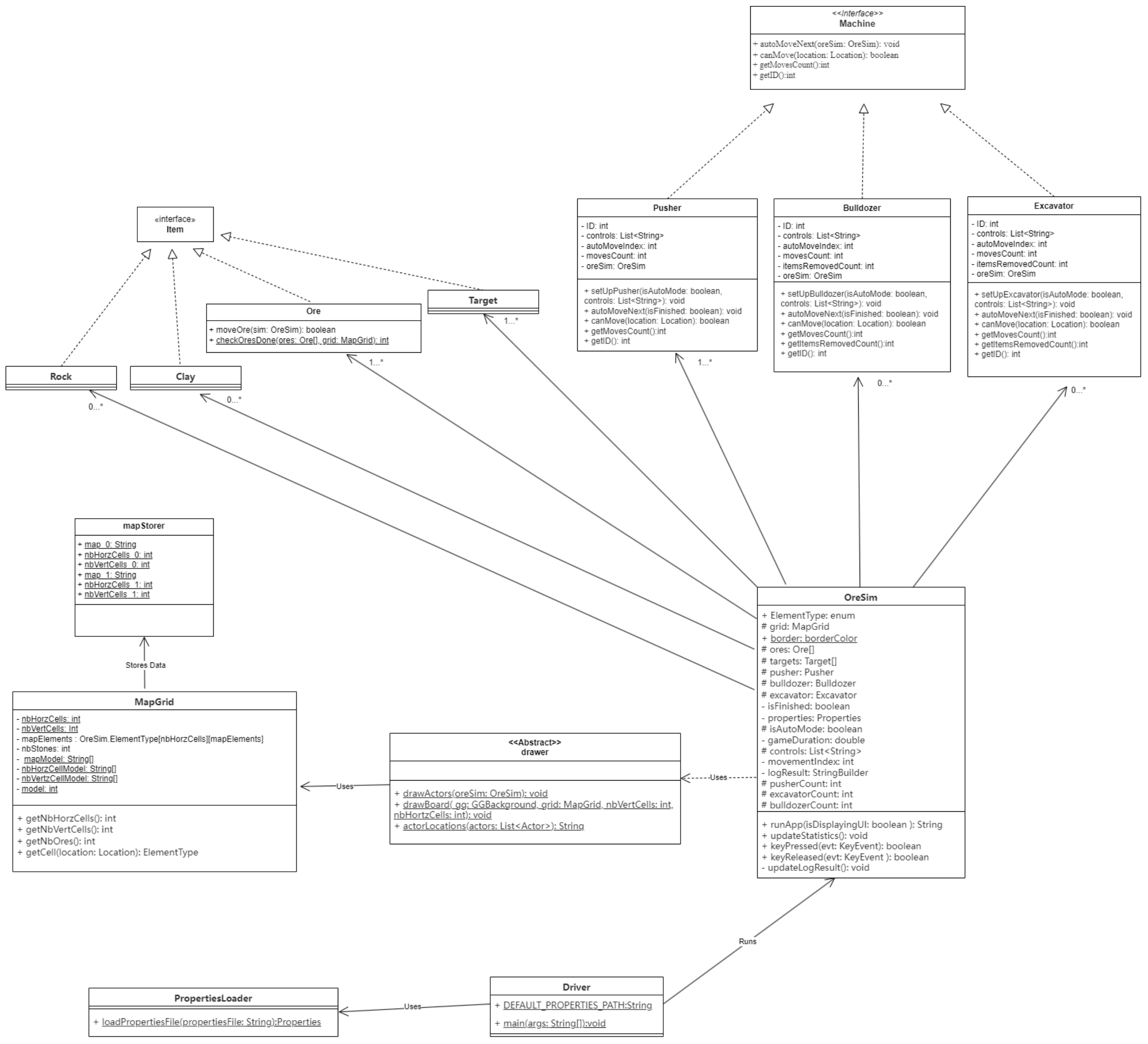
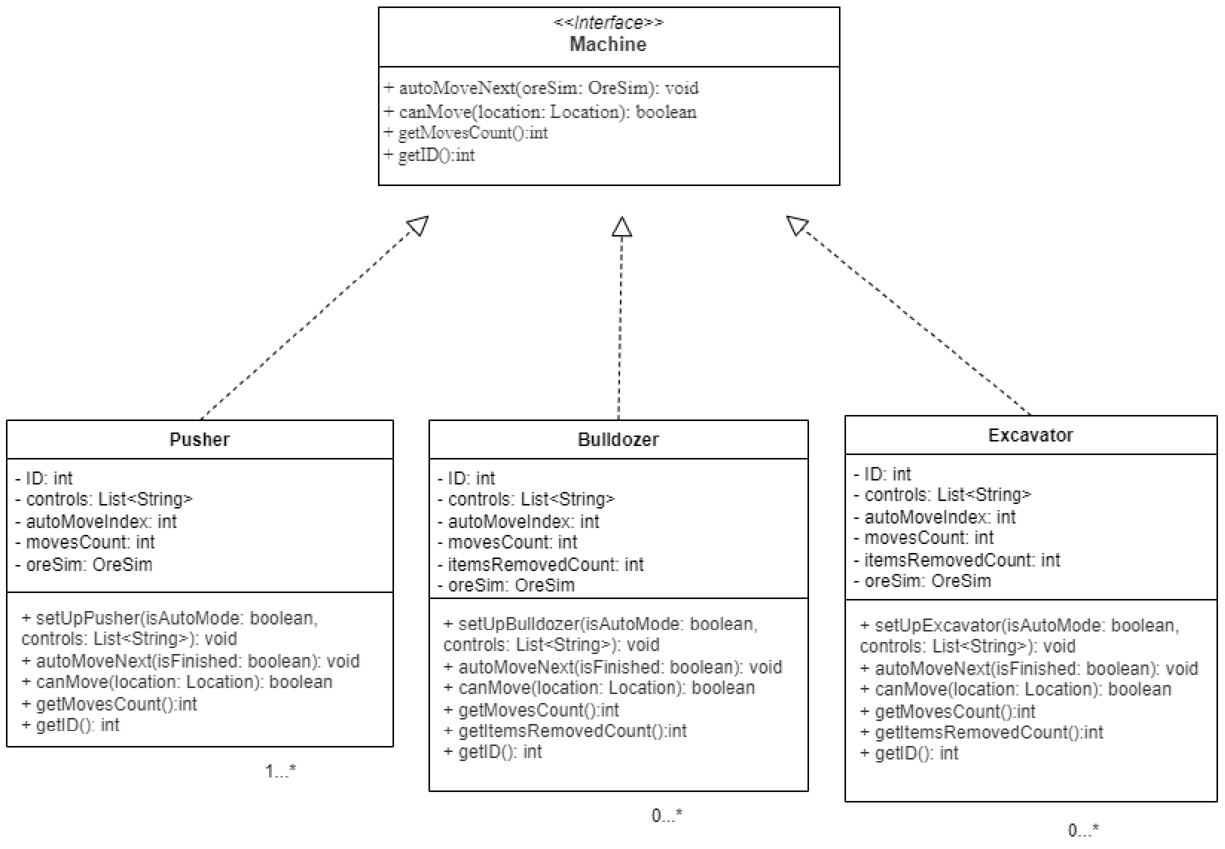

<<Interface>>
Machine
+ autoMoveNext(oreSim: OreSim): void
+ canMove(location: Location): boolean
+ getMovesCount():int
+ getID():int
Pusher
- ID: int
- controls: List<String>
- autoMoveIndex: int
- movesCount: int
- oreSim: OreSim
+ setUpPusher(isAutoMode: boolean, controls: List<String>): void
+ autoMoveNext(isFinished: boolean): void
+ canMove(location: Location): boolean
+ getMovesCount():int
+ getID(): int
Bulldozer
- ID: int
- controls: List<String>
- autoMoveIndex: int
- movesCount: int
- itemsRemovedCount: int
- oreSim: OreSim
+ setUpBulldozer(isAutoMode: boolean, controls: List<String>): void
+ autoMoveNext(isFinished: boolean): void
+ canMove(location: Location): boolean
+ getMovesCount():int
+ getItemsRemovedCount():int
+ getID(): int
Excavator
- ID: int
- controls: List<String>
- autoMoveIndex: int
- movesCount: int
- itemsRemovedCount: int
- oreSim: OreSim
+ setUpExcavator(isAutoMode: boolean, controls: List<String>): void
+ autoMoveNext(isFinished: boolean): void
+ canMove(location: Location): boolean
+ getMovesCount():int
+ getItemsRemovedCount():int
+ getID(): int
«interface»
Item
Ore
+ moveOre(sim: OreSim): boolean
+ checkOresDone(ores: Ore[], grid: MapGrid): int
Target
Rock
Clay
0...*
0...*
1...*
1...*
1...*
0...*
0...*
mapStorer
+ map_0: String
+ nbHorzCells_0: int
+ nbVertCells_0: int
+ map_1: String
+ nbHorzCells_1: int
+ nbVertCells_1: int
Stores Data
MapGrid
- nbHorzCells: int
- nbVertCells: Int
- mapElements : OreSim.ElementType[nbHorzCells][mapElements]
- nbStones: int
- mapModel: String[]
- nbHorzCellModel: String[]
- nbVertzCellModel: String[]
- model: int
+ getNbHorzCells(): int
+ getNbVertCells(): int
+ getNbOres(): int
+ getCell(location: Location): ElementType
Uses
<<Abstract>>
drawer
+ drawActors(oreSim: OreSim): void
+ drawBoard(gg: GGBackground, grid: MapGrid, nbVertCells: int, nbHortzCells: int): void
+ actorLocations(actors: List<Actor>): String
Uses
OreSim
+ ElementType: enum
# grid: MapGrid
+ border: borderColor
# ores: Ore[]
# targets: Target[]
# pusher: Pusher
# bulldozer: Bulldozer
# excavator: Excavator
- isFinished: boolean
- properties: Properties
# isAutoMode: boolean
- gameDuration: double
# controls: List<String>
- movementIndex: int
- logResult: StringBuilder
# pusherCount: int
# excavatorCount: int
# bulldozerCount: int
+ runApp(isDisplayingUI: boolean): String
+ updateStatistics(): void
+ keyPressed(evt: KeyEvent): boolean
+ keyReleased(evt: KeyEvent): boolean
- updateLogResult(): void
Runs
PropertiesLoader
+ loadPropertiesFile(propertiesFile: String):Properties
Uses
Driver
+ DEFAULT_PROPERTIES_PATH:String
+ main(args: String[]):void

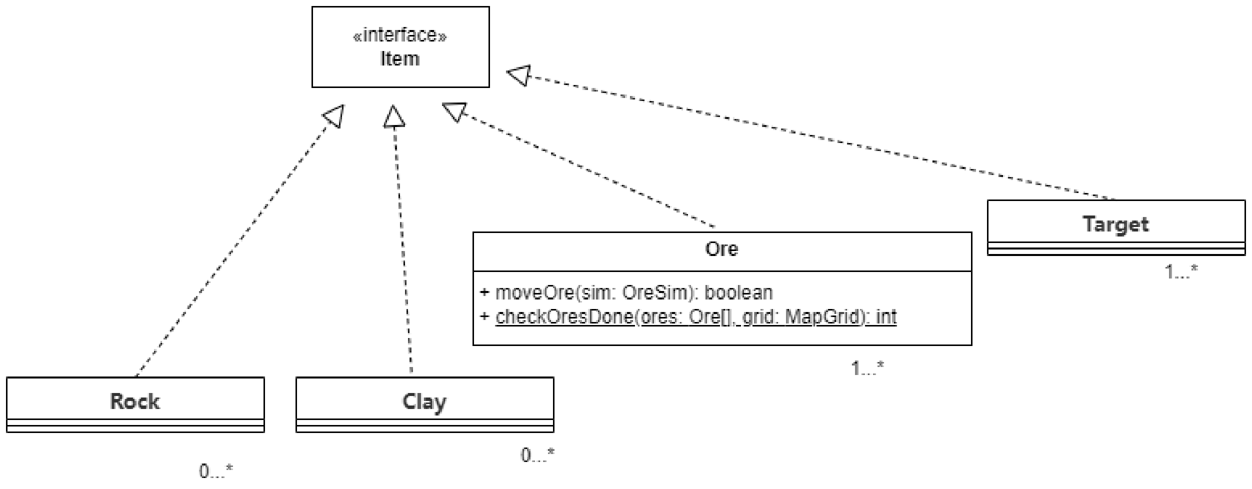
FIGURE 1 - DESIGN CLASS DIAGRAM

One of the changes we have decided to make was the implementation of the Polymorphism, as shown in the figures 1, 2, and 3, we have introduced new interface classes such as ‘Machine’ which the classes ‘Pusher’, ‘Bulldozer’, and ‘Excavator’ now implement, and ‘Item’ which is implemented by ‘Rock’, ‘Clay’, ‘Target’ and ‘Ore’. This design decision leads to the many benefits, now allowing for flexible interactions and the easy modification or extension of these classes.



<<Interface>>
Machine
+ autoMoveNext(oreSim: OreSim): void
+ canMove(location: Location): boolean
+ getMovesCount():int
+ getID():int
Pusher
- ID: int
- controls: List<String>
- autoMoveIndex: int
- movesCount: int
- oreSim: OreSim
+ setUpPusher(isAutoMode: boolean, controls: List<String>): void
+ autoMoveNext(isFinished: boolean): void
+ canMove(location: Location): boolean
+ getMovesCount():int
+ getID(): int
1...*
Bulldozer
- ID: int
- controls: List<String>
- autoMoveIndex: int
- movesCount: int
- itemsRemovedCount: int
- oreSim: OreSim
+ setUpBulldozer(isAutoMode: boolean, controls: List<String>): void
+ autoMoveNext(isFinished: boolean): void
+ canMove(location: Location): boolean
+ getMovesCount():int
+ getItemsRemovedCount():int
+ getID(): int
0...*
Excavator
- ID: int
- controls: List<String>
- autoMoveIndex: int
- movesCount: int
- itemsRemovedCount: int
- oreSim: OreSim
+ setUpExcavator(isAutoMode: boolean, controls: List<String>): void
+ autoMoveNext(isFinished: boolean): void
+ canMove(location: Location): boolean
+ getMovesCount():int
+ getItemsRemovedCount():int
+ getID(): int
0...*

FIGURE 2 - MACHINE INTERFACE



«interface»
Item
Ore
+ moveOre(sim: OreSim): boolean
+ checkOresDone(ores: Ore[], grid: MapGrid): int
1...*
Target
1...*
Rock
0...*
Clay
0...*

FIGURE 3 - ITEM INTERFACE

We additionally decided to incorporate the Information Expert principle by allocating responsibilities exclusively to classes that hold the necessary information. ‘MapGrid.java’ has been streamlined to focus solely on the representation of the game grid. Relevant methods have been shifted from the OreSim class to their respective classes, such as, canMove() and moveOre() methods are transferred to the Ore class. Furthermore, entities like ‘Pusher’, ‘Bulldozer’ and ‘Excavator’ are now responsible for their own movement and interaction logic, thus enhancing cohesion and maintaining a clear separation of logics.

We have also introduced a ‘drawer.java’ class, as it further heightens cohesion by taking responsibility for the rendering of the game state, thus we can separate the visual and logical components of the simulation into their own specialized classes.

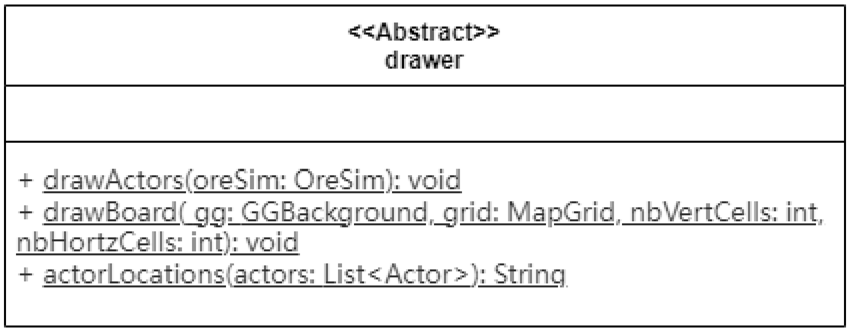

<<Abstract>>
drawer
+ drawActors(oreSim: OreSim): void
+ drawBoard(gg: GGBackground, grid: MapGrid, nbVertCells: int, nbHortzCells: int): void
+ actorLocations(actors: List<Actor>): String

FIGURE 4 - DRAWER CLASS

Furthermore, the introduction of ‘mapStorer’ significantly improves the data management strategy. By being solely responsible for the retrieval and containment of map information, we have instead introduced a more modular data handling approach, allowing for more maps to be included in later stages.

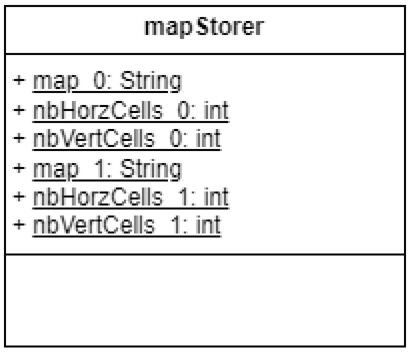


FIGURE 5 - MAPSTORER CLASS

By adopting these GRASP principles, as demonstrated in the model, the system is now better organized, with clear boundaries and responsibilities among the classes. Each class has become an expert in its specific domain, with high cohesion and low coupling. The new design will now be more maintainable with high extensibility.

P3:

1) One variable of the machine classes is an unique ID variable, when new instances are created for the different machine types, each of them would be given a corresponding ID, this

aligns with the Information Expert principle, as each machine becomes an expert in its own identification and operation data.

2) Introducing future machine types are simplified as they can be introduced by implementing the ‘Machine’ interface. Classes such as the mentioned ‘HaulTruck’ and ‘Crusher’ can implement the shared functionalities of Machines, while implementing their own unique capabilities. This is another application of the Polymorphism principle.

3) For new obstacles, we implement a similar strategy to the new Machine types, but instead the new obstacles will implement ‘Items’, although the movement logic of the machines may have to be altered depending on their interaction with the new obstacles. This is another application of the Polymorphism principle

4) To accommodate various control schemes, another class specifically for the handling of inputs can be introduced such as a ‘ControlStratergy’ interface, with each machine utilizing different implementations of this interface, this would decouple control mechanisms from the machine themselves, and would be in line with the Strategy and Polymorphism principles.

5) A significant change in the updateStatistics method in OreSim is that it now has the capability to check if a machine exists before writing about its stats, otherwise, it just skips writing about that particular machine. It also calls the respective ID and data of the machine, so can be used in a loop to print statistics for multiple machines.