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 issue of low cohesion and high coupling, the logic of the simulator is predominantly contained in two files, ‘MapGrid.java’ and ‘OreSim.java’, the lack of well designed classes and responsibilities have lead to a complicated code base, creating a challenging issue when introducing new and modifying existing features without affecting other parts of the system. Patterns such as the ‘Creator Pattern’ and the ‘Information Expert’ can be used to combat these issues.

Creator Pattern: Applying this principle, a ‘MapCreator’ class could be introduced, being responsible for generating and initializing map elements could significantly declutter ‘MapGrid’. The new class would also centralize map creation responsibilities, enhancing the ability to extend and maintain it.
Information Expert: Applying this principle, ‘MapGrid.java’ should solely focus on grid management, delegating the handling of map states and logic to other specialized classes, this would also improve cohesion.

The issue of suboptimal encapsulation and limited polymorphism, the current design inadequately encapsulates the functionalities of different elements within the game, such as ores, machinery, and obstacles all within the same class. The lack of encapsulation not only makes the code less secure but hampers readability and flexibility. Patterns such as ‘Polymorphism’, ‘Low Coupling’ and ‘Protected Variations’ could be used to combat these issues.

Polymorphism: Through the use of polymorphism, we can create abstract classes or interfaces for common shared functions by the different elements, such as machinery (Pusher, Bulldozer, Excavator) and obstacles (Rock, Clay). Through this, easier additions and modifications of element types can be facilitated in the future.
Low Coupling: Applying this principle, we can separate the UI logic from the game logic, this can be done through a new ‘GameManager’ class. The separation would help ensure that changes to either logic would have minimal impact on each other, further enhancing modularity.
Protected Variations: Applying this principle, introducing new specialized classes such as a ‘GridManager’ for grid-related operations and an ‘ElementTypeManager’ for managing different types of elements could help protect the system from variations by localizing said changes to specific parts.

P2:

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

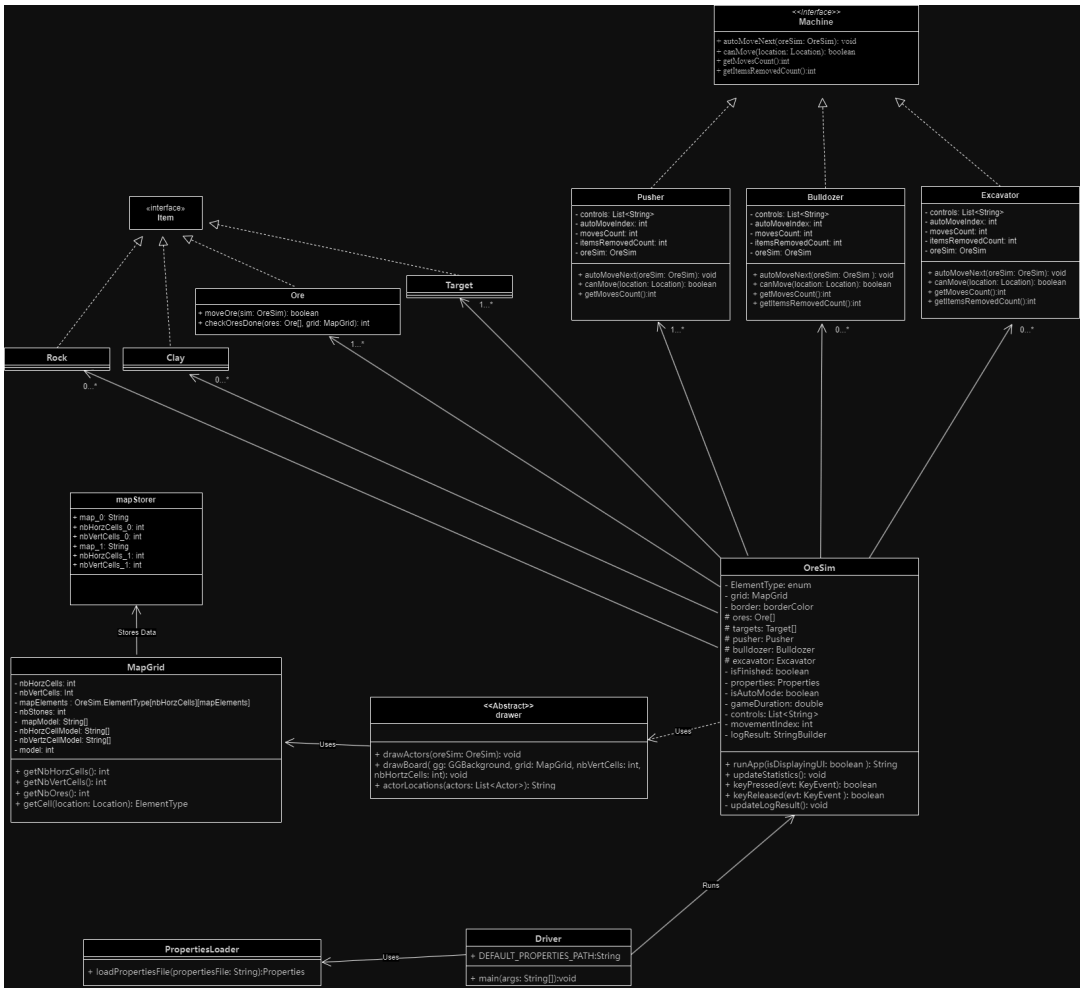

FIGURE 1 - DESIGN CLASS DIAGRAM

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.

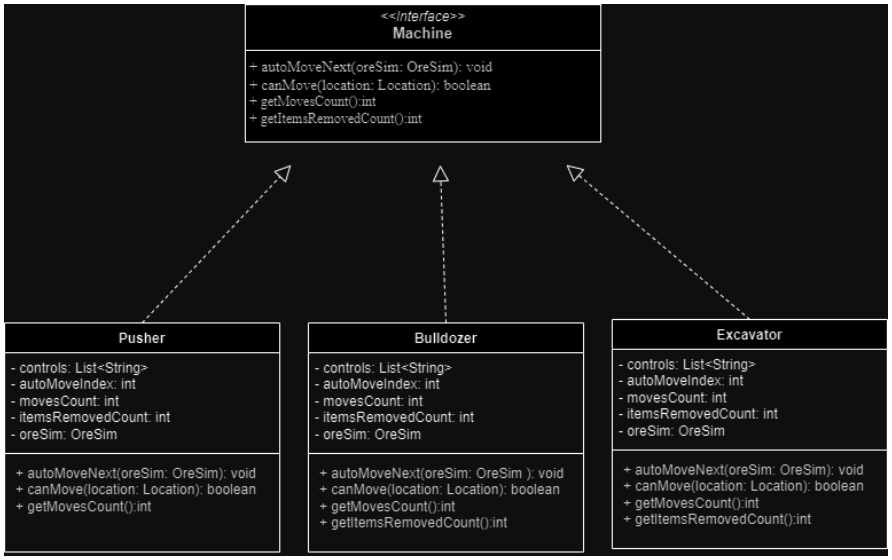

FIGURE 2 - MACHINE INTERFACE

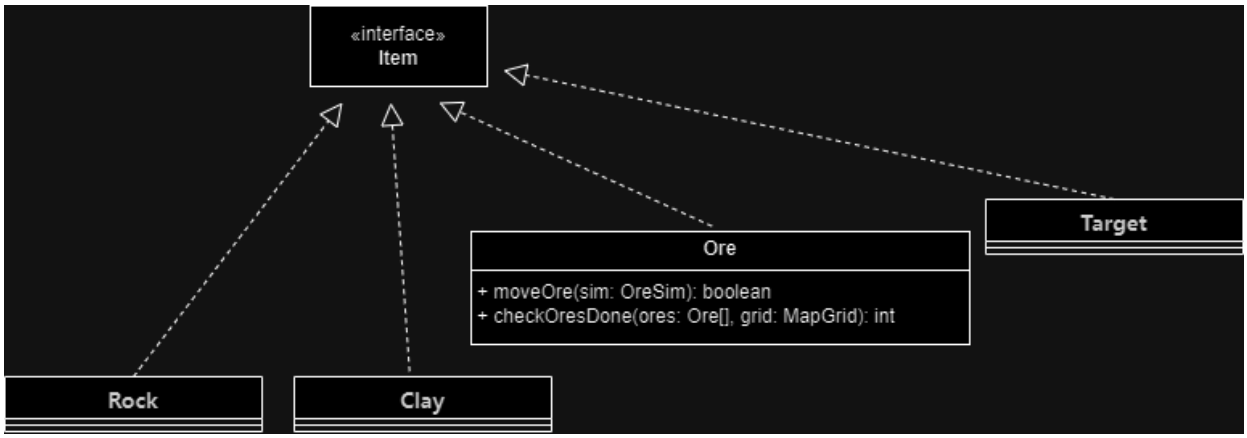

FIGURE 3 - ITEM INTERFACE

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. Furthermore, entities like ‘Pusher’ 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.

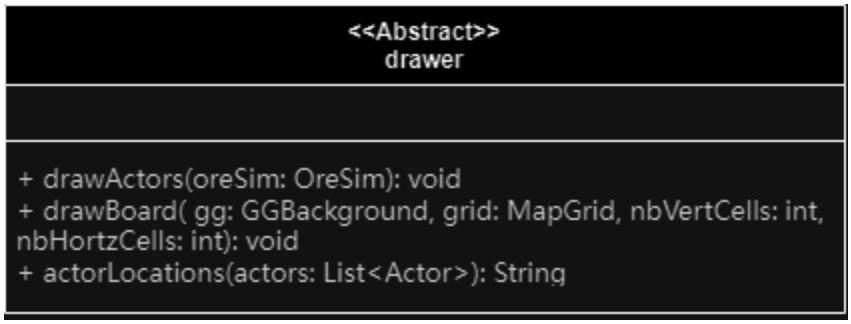

FIGURE 4 - DRAWER CLASS

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.

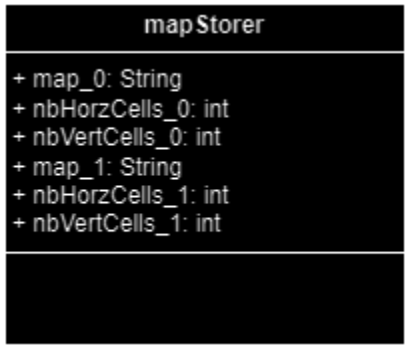

FIGURE 5 - MAPSTORER CLASS

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.