SWEN30006 Project 1

Workshop Thurs 11AM-1PM, Team 06
Pavit Vathna, 1235224
Daniel Duong, 1356850

Analysis of Current Design

In the current designs (Figure 1), the OreSim class is overloaded with responsibilities such as checking if ores can move and moving ores as well as updating statistics and log results amongst other responsibilities when these responsibilities should be delegated to other classes. In addition, as a consequence of containing the enum class, OreSim ends up having classes of machines and obstacles and handles the responsibility of initializing objects of many classes in the game. This makes the class become bloated without well-defined responsibilities and difficult to understand due to unideal application of **Creator** and I**nformation Expert** patterns. This issue introduces high complexity resulting in **Low Cohesion** and **High Coupling** which then makes the design not open for future extensions as it requires modifications in multiple places to add new types of machines or obstacles for example. This makes it difficult to add the logic of new types of machines such as the excavator and bulldozer as that would mean having to change the logic within the OreSim class.

Moreover, there is a lack of **Polymorphism**, particularly within the different types of machines such as the pusher, excavator and bulldozer despite sharing similar features and behavior. Consequently, it makes the design less extensible with repetitive code from having to recode the same behavior. For instance, all the machines would have a canMove() method which would have to be re-coded in all of the classes separately. The OreSim would also have to make changes adapting to the new class which can convolute the logic.

Proposed New Design of Simple Version

For a better design with **Low Coupling** and **High Cohesion**, the enumeration class in OreSim should be a separate class from OreSim. From Figure 2, The object classes such as *Clay, Rock, Ore, Target, Excavator, Bulldozer* should be separated from OreSim and their constructors to initialize themselves should be in their respective classes instead of being in OreSim. As each type of machine shares similar features and performs similar operations, inheritance is used to have each type of machine (*Excavator* and *Bulldozer*) a subclass of the superclass *Machine* and each type of obstacles (*Clay* and *Rock*) to a superclass *Obstacle*. *Machine* and *Obstacle* will be a subclass of the superclass *Actor* as each object has to move location in a game. Ore and Target will also be subclasses of superclass *Actor* due to setting their location or moving somewhere on the map. This makes it easier to change and add more functionalities without breaking other application logic and easier to understand.

Moreover, this allows the new types of machines and obstacles to be added as subclasses of their respective superclass making the design open for extension. This **Polymorphic behavior** defines a well-separated responsibilities for each class resulting in the OreSim class having less responsibilities.

From Figure 1, OreSim also has the responsibility to track and update performance statistics of each machine so we created another class *Analyzer* in order to produce the required performance statistics.

In resolving this issue of **High Coupling** in previous design (Figure 1), it wouldn't be effective to utilize indirection due to a lack of an intermediary class to move the application logic. Although, with a **pure fabrication** and **indirection**, it can help move responsibilities away from the OreSim class which will ultimately improve **cohesion** and **reduce coupling**.

Alternatively, a **controller pattern** can be utilized to reduce the responsibilities of the OreSim. Utilizing a controller, the OreSim can have a focus on creating the graphical interface whilst the controller would be in charge of managing the movement, machinery and ore's logic. This would greatly improve cohesion as OreSim would only have a single purpose which is to manage the graphical interface.

Proposed Design of Extended Version

As previously discussed in our design, our new design follows the design class in figure 3. In this new design, we have implemented abstract classes *Machine* and *Obstacle* as all their superclasses share similar attributes and they have abstract methods: autoMoveNext() and canMove() to perform the automatic movement of machines and checking if they are able to move. The subclasses of *Machine* and *Obstacle* will override these methods due to having different conditions for being able to move as in Figure 4. As illustrated in the new design diagram, we have implemented the *Actor* as a superclass of *Machine* and *Obstacle* as all objects in the game that could move to a new location or have a location somewhere on the map in the game. This use of inheritance allows an extensible design when introducing new types of machines and obstacles into the game with minimal amount of changes in coding the new logic. Most importantly, the game now has a design of **High Cohesion** and **Low Coupling**.

As for producing performance statistics, the *Analyser* class is made with methods such as updateLogResult(), updateStatistics(), getActorsLocation() in order to track and update statistics while the game is running in OreSim.

Overall, all the GRASP principles are taken into consideration to support extensions when refactoring the given design class to the new design class as proposed in figure 3.

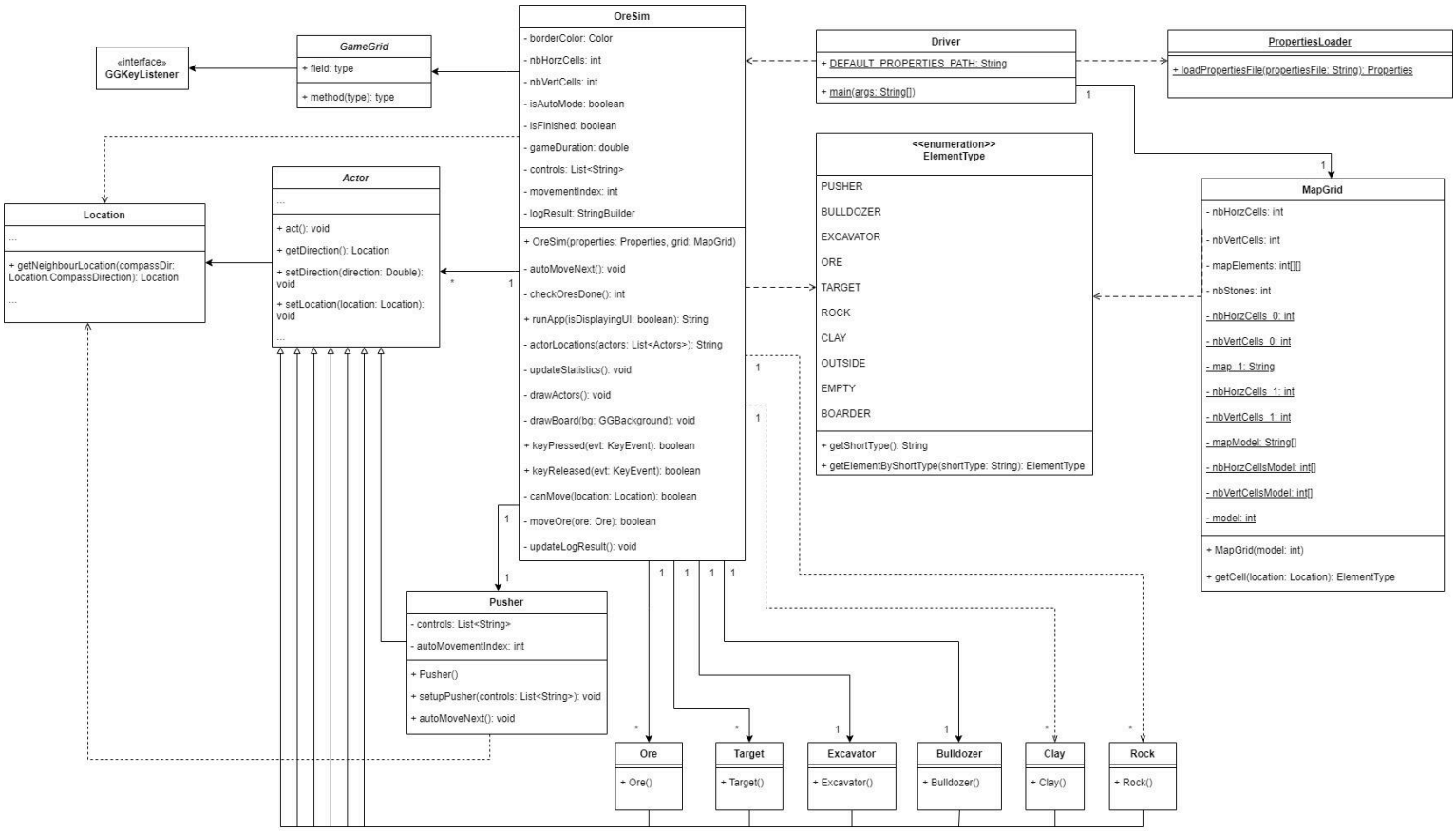

Figure 1: Design Class Without Extension

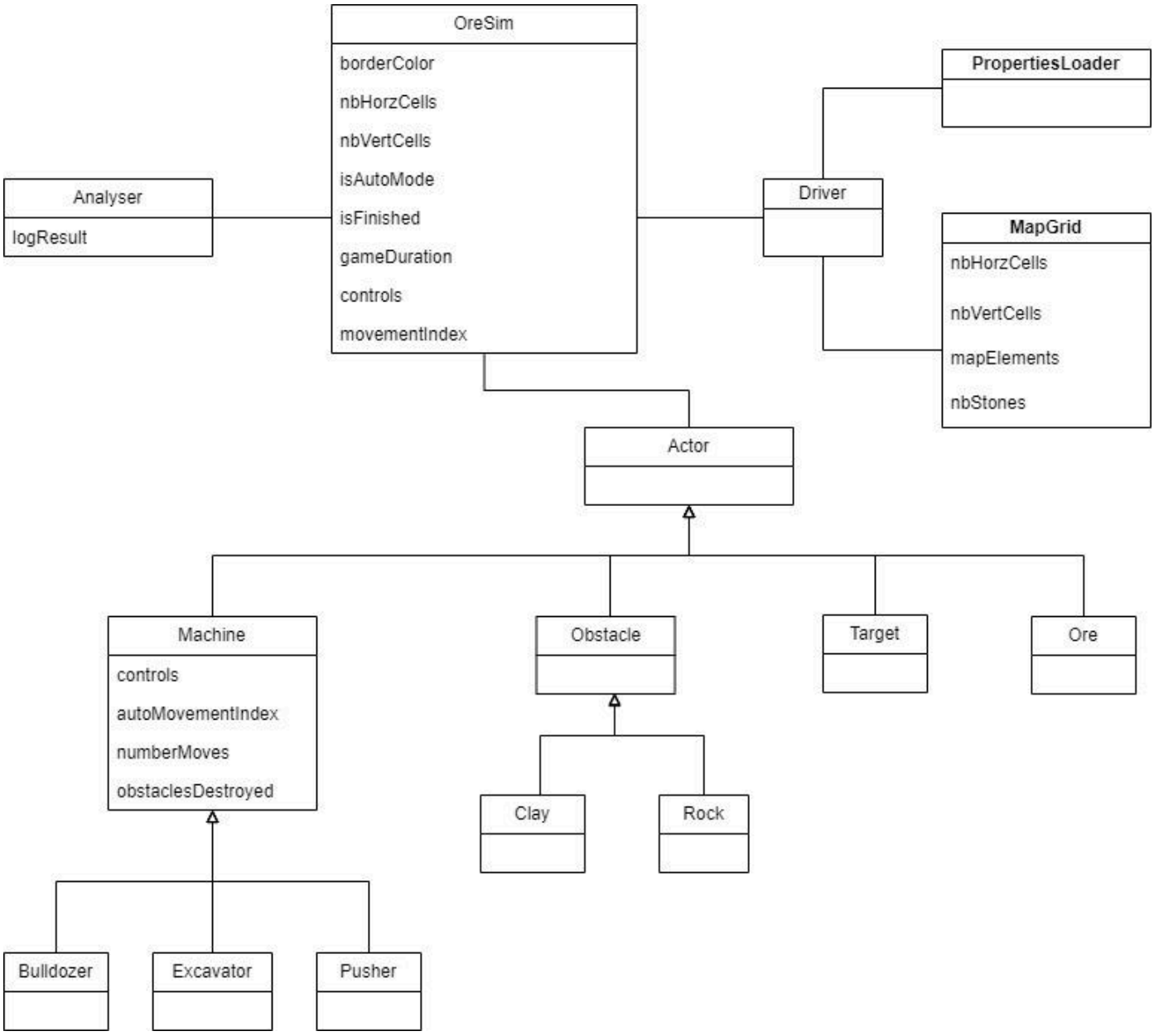

Figure 2: Domain Model Diagram with Extension

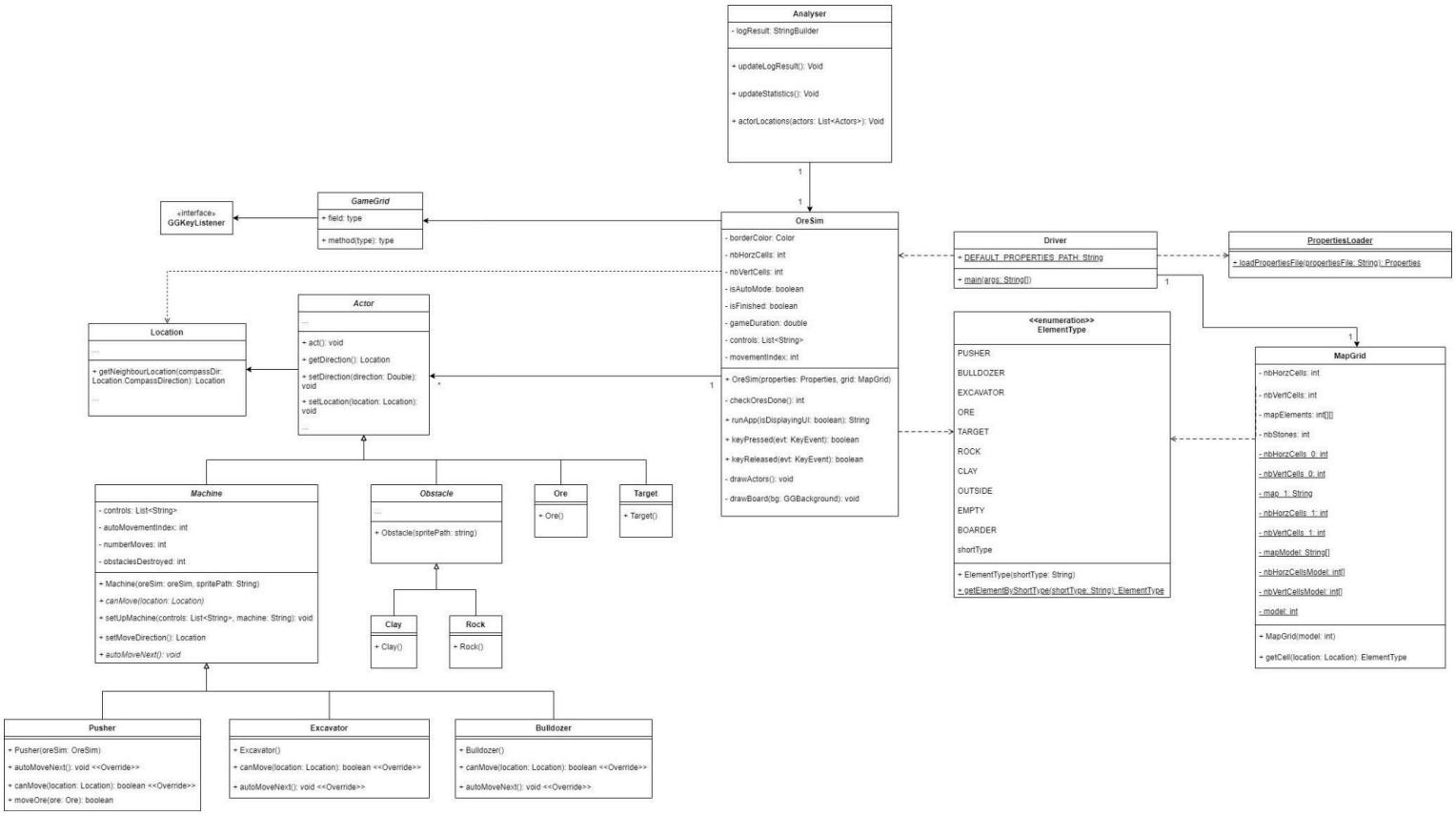

Figure 3: Design Class Diagram with Extension

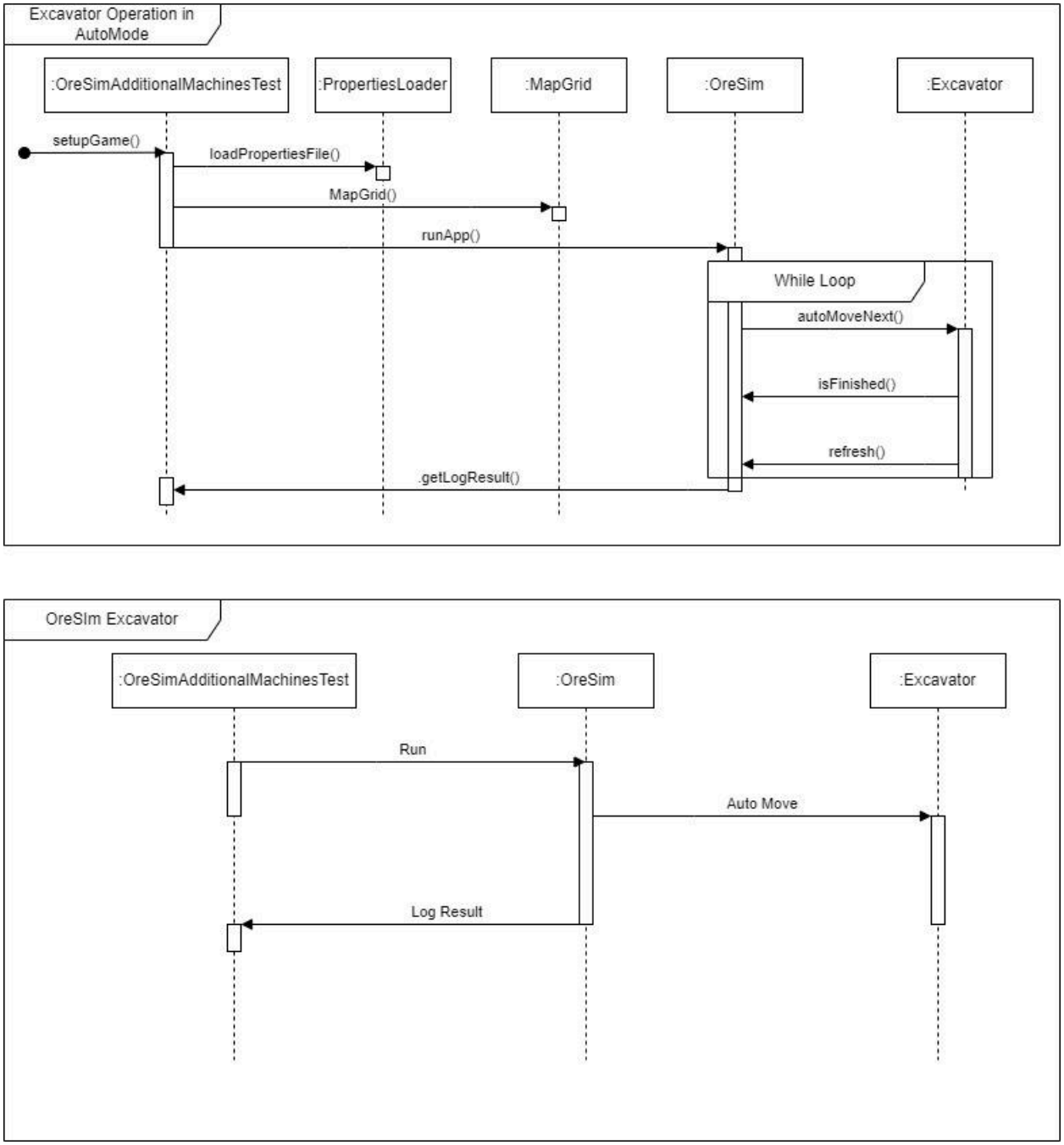

Figure 4: System Sequence and Design Sequence Diagrams of Automatic Excavator Operation