SWEN30006 Report - Project 1

Workshop 2, Team 09
Yan Gong 1292879
Jiayi Wang 1305350
Yiru Liu 1350975
(3 member team)

The current design was found to violate some GRASP principles and the limited structure was not conducive to the implementation of the expansion plan. Our main goal was to involve increasing cohesion of the OreSim class by assigning some of its functionalities to the fabricated (abstract) classes and lowering coupling and supporting protected variation among classes by creating interfaces, without altering too much of the structure of current design. And with the addition of interfaces/classes, it makes the design more flexible and maintainable to deal with future extension.

In our report, we will address the limitations of the current design and present a new design concept based on GRASP principles, and finally discuss the appropriate version for future extensibility.

Part 1. Modifications to Support GRASP Principles

1.1 Converting Inner Classes to Top-level Classes

In the current design, the OreSim class embedding inner classes like Ore and Target reduces cohesion and makes refactoring difficult. Adding inner classes increases the responsibilities of the OreSim class, as it has to handle specific behaviors of each inner class, this violates the high cohesion principle which indicates that OreSim class should be focusing on a main functionality, and leads to a less maintainable code. Meanwhile, this design results in tight coupling between the OreSim class and each of its inner classes, since inner classes are dependent with the structure and context of OreSim. This would reduce code reusability and extensibility. To reduce the negative impacts of inner classes, we refactored the simple version so that inner classes are converted to top-level classes. This design helps to potentially achieve **polymorphism** by creating abstract classes that can define common functionalities among the previous inner classes (more on this in 1.2 and 1.3). Hence, this enhances cohesion of OreSim by reducing its functionality of controlling the objects, and **lowers coupling** by weakening the interaction between Oresim and any of the objects.

Meanwhile, the design of the bulldozers, excavators and bulldozers has not yet been completed. First, in the simplified version, excavators and bulldozers are not yet mobile. Therefore, we have assigned each vehicle responsibilities that it should perform, such as

rules for moving in automatic mode, identifying obstacles ahead to determine whether it can move, to improve the current design.

1.2 Creating Machine Abstract Class

Based on the **Information Expert** principle, we believe that OreSim should have the responsibility to move each vehicle in automatic mode since it knows the most information. But OreSim should be more focused on the running of the game itself, and if it is also responsible for other events such as machine movements, it cannot satisfy the cohesion within the class. Therefore, based on the **Pure Fabication** Principle, we decided to add a Machine class and make it responsible for managing the automatic driving of the vehicle (e.g. the autoMoveNext() method) and other related behaviors. In addition , assume that without the Machine class, although Bulldozer, Excavator, and Pusher inner classes have been converted to classes, this design does not support extension due to the duplication of content between classes (each class has an autoMoveNext() method and performs the same behavior). To sum up, we create an abstract class called Machine and let the Bulldozer, Excavator, and Pusher classes inherit the common functionality of Machine.

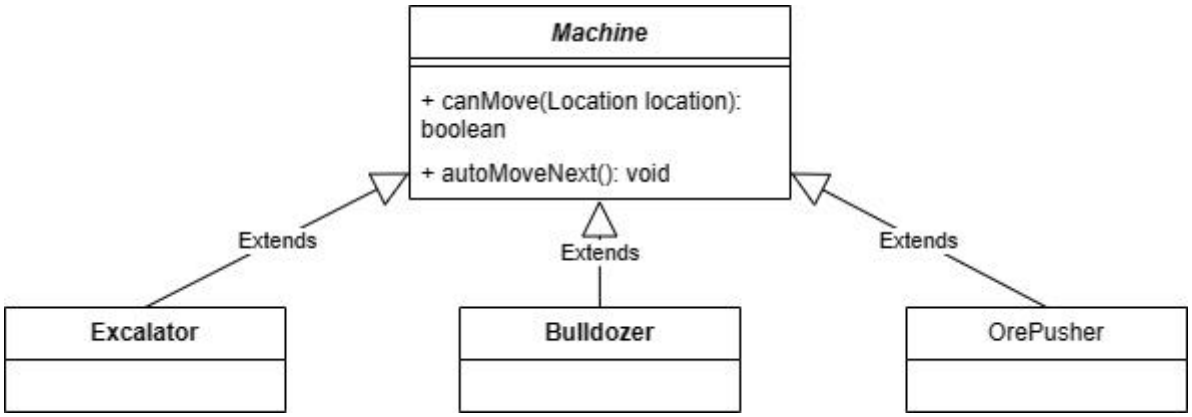

Figure 1. Machine Abstract class and its subclasses

The Machine class concentrates common methods in its subclasses and allows subclasses to have their own unique functionality by overwriting some methods, which both reduces code redundancy and supports **polymorphism**. In addition, through polymorphism, all subclasses are managed collectively, which helps protect variation. Therefore, the abstract class Machine guarantees scalability when new machines are introduced into the software in the future.

1.3. Creating ObstacleAbstract Class

Similar to the purpose of creating the Machine, we created another abstract class named Obstacle and let the Clay and Rock classes extend it. Again, this **polymorphic** behavior enhances the extensibility of the design for more types of variables.

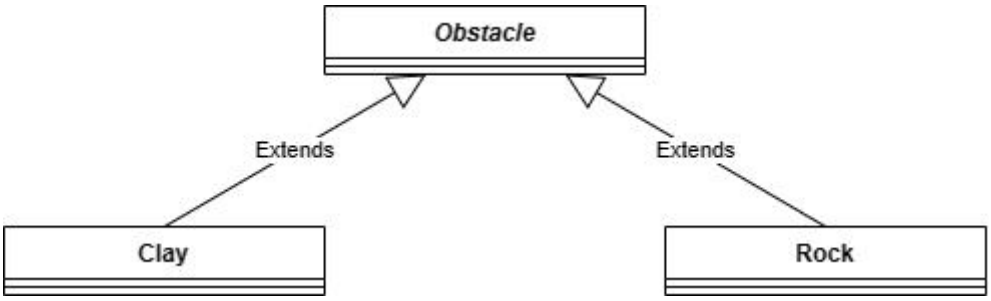

Figure 2. Abstract class Obstacle and its subclasses

1.4. Creating ConcreteActorFactory, GameLogger and Statistics Classes for OreSim high Cohesion

There are concerns that OreSim is now still left with several distinct functionalities including game environment configuration, game control, logging statistics and instantiating objects, which not satisfy both high cohesion and low coupling.To make the OreSim more focused on fewer responsibilities, ConcreteActorFactory and GameLogger were introduced to take some responsibilities from OreSim.

The ConcreteActorFactory class provides an interface or method for creating objects, including machine objects, obstacle objects, Ore, and Target. Referencing the factory pattern, the ConcreteActorFactory is a direct implementation of the **Creator** principle, reducing the dependency between concrete elements and the OreSim simulator. Furthermore, the rest of the OreSim is protected from changes during creation through the ConcreteActorFactory encapsulated creation logic, following the **Protected Variation** principle.

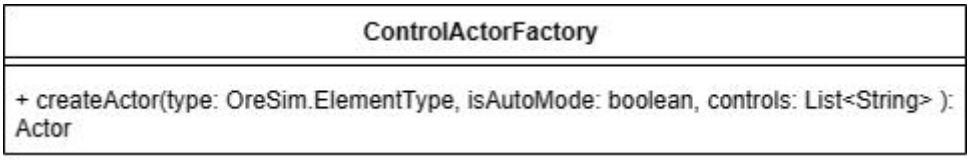

Figure 3. ControlActorFactory Class

Furthermore, we also moved updateLog() and actorLocations() methods from the OreSim to the GameLogger class, letting the GameLogger to provide the log of dynamic movement of objects, so that it can support further testing of the OreSim. which As a form of pure

fabrication, the GameLogger also helps to maintain **high cohesion** by separating the responsibilities of OreSim.

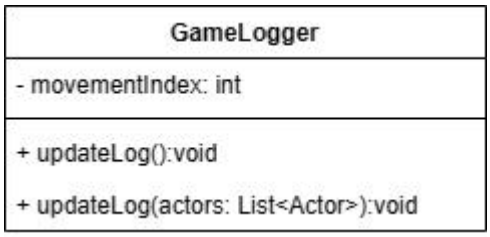

Figure 4. GameLogger Class

With the **Pure Fabrication** principle in mind, we also created a class dedicated to statistics and assigned it the updateStatistics() method originally in OreSim to support OreSim's high cohesion and future reusability. Similarly, having a class focus on only one primary function (e.g. Track operational statistics) ensures **high cohesion**. We have modified the updateStatistics() method so that the software displays the appropriate output for the extended version.

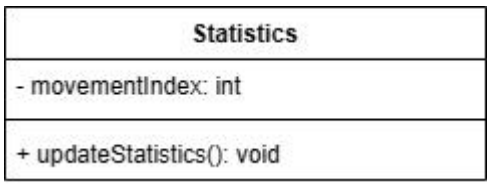

Figure 5. Statistics Class

Part 2. Future Extension Support

2.1 Introducing Multiple Machines of Different Types

To distinguish machines of the same type (ie. Pusher, Excavator or Bulldozer), we created a new attribute called id (int) in the abstract class Machine so that each machine can be identified, and we created pushers (ArrayList<Pusher>), bulldozers (ArrayList<Bulldozer>) and excavator (ArrayList<Excavator>) to record different machines. This allows each additional machine to use the same method as the machines in the extended version.

2.2 More Types of Machine With Different Capabilities

Since we have created a Machine abstract class, this extension is enabled by letting each new type of machine extend the Machine abstract class. Thus we can reuse some code like the autoMoveNext() function if the movement behavior is the same, and override some methods like canMove() to allow distinct features.

2.3 More Types of Obstacle

This extension is guaranteed by the creation of the abstract class Obstacle. More types of obstacles can be added by extending the Obstacle abstract class, simply by passing their corresponding images to construct.

2.4 Adding Various Ways to Control Machines

The simple version is much less likely to cope with varying control strategies, since highly redundant code needs to be added and the structure might potentially be changed. To accommodate different ways of controls in the future, we created StatisticsHandler, LogHandler and Controller interfaces.

2.4.1 Creating Controller Interface

To control the machine in different ways, the Controller interface can be created and let any existing or future controlling strategy implement this interface.

There are two existing strategies, ie, manually control using a keyboard and automatic control by the pre-defined sequence. The keyboard control is achieved by the keyPressed() and keyReleased() methods that were originally in OreSim. To help future extension we created a KeyPressed class to implement the two methods. Similar to keyboard, the methods (autoMoveNext() and canMove()) in the Machine class support pre-defined sequence control, and we can also create the PredefinedAuto to implement the methods. The Controller interface defines two public methods, ie. move() and canMove(), which are based on summarizing the behaviors of KeyPressed and PredefinedAuto classes. The keyPressed() and autoMoveNext() methods resemble the behavior of moving a machine, and the keyReleased() and canMove() methods resemble the behavior of judging if a machine can move. We have renamed the methods in KeyPressed and PredefinedAuto classes into move and canMove with different parameters, since interface allows methods overloading.

Figure 6 also illustrates the three examples of new control strategies. We can add the controls using a remote stick and using an autonomous planner by implementing move() and canMove() methods in RemoteControlStick and AutonomousPlanner respectively. The use of interface allows code reuse, guarantees different behaviors of control strategies by overloading the methods, and achieves **Polymorphism** and **Protected Variation**.

The strategy of controlling machines directly may behave differently from the rest of other strategies, since it is independent of a third party. Therefore, we put the controlling meth0d inside each type of machines to give it direct control.

Please be advised this design is independent of our extended version. When we try to implement this design in our code, since the Test file is immutable, which means we can't change the connection between OreSim and mapGrid, adding Controller would make the

design model look complicated, so we will only discuss the desired future design here and not actually implemented.

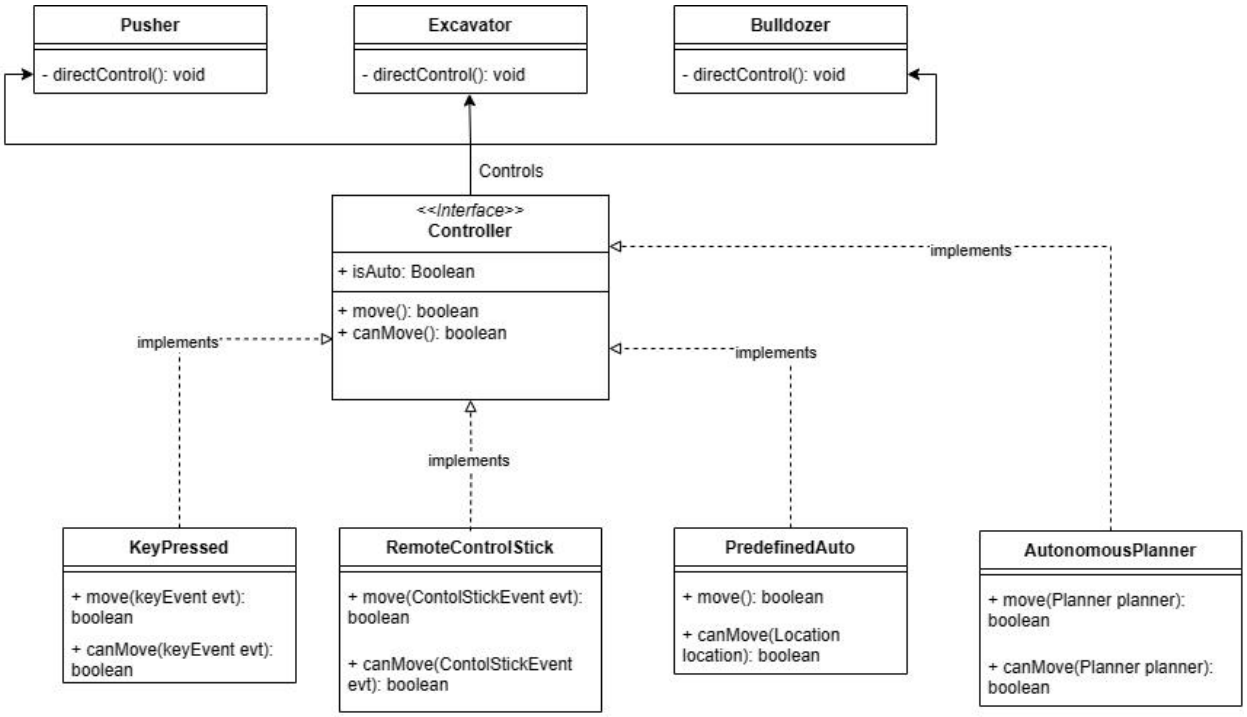

Figure 6. Examples of how Controller interface is used to support new control strategies

2.4.2 Creating StatisticsHandler and LogHandler Interfaces

The StatisticsHandler interface is implemented by the Statistics class, and the LogHandler is implemented by the GameLogger class. The design logic of StatisticsHandler and LogHandler are quite similar: considering the operational statistics may change due to different control strategy, other types of statistics trackers and loggers can still implement the same interfaces, since an interface support polymorphism by enabling multiple classes with different behavior to implement it. This allows other types of statistics trackers and loggers to interact with the objects in OreSim. **Polymorphism** also achieves protected variation by encapsulating the varying areas (ie. different control strategies) inside stable interfaces.

In Conclusion, there is a trade-off between high cohesion and low coupling: our design choices support high cohesion more in some specific cases. This is because sometimes adding classes, while satisfying the centrality of each class, may inevitably increase the relationships between classes. However, we tried some changes to achieve low coupling, but due to the fixed associations between some classes in the base design that could not be modified (causing the test file to change), and the requirements suggested that we should not change too much structure for future design requirements. Therefore, our design is

more in line with the design principle of high cohesion, which is also easy to understand, and also ensures the maintainability and enhancement of the code, with deep consideration for future expansion.