SWEN30006 Project 1
Workshop Fri 13:15, Team 03

Ethan Hawkins 1355842
William Spongberg 1354263
Joshua Linehan 1173490

Part 1: An analysis of the current design

The Driver, MapGrid and PropertiesLoader classes all have High Cohesion and Low Coupling, and correctly follow the GRASP principles. The Driver acts as a Creator by initialising the program and creating the appropriate objects necessary to achieve the program's goal: safely testing and exploring autonomous mining strategies. The MapGrid acts as an Information Expert by holding all the possible game map configurations and decoding them into ElementTypes, which are then used by OreSim and for testing purposes. The PropertiesLoader acts as Pure Fabrication by converting the complex flow of information from the inputted properties file into a usable Properties object.

However, the OreSim class, the class doing most of the work for this program, has a variety of issues, those primarily being Low Cohesion and High Coupling.

OreSim has too much functional responsibility. It should be for simulating the game logic alone, as its name suggests. Correctly, it does simulate the game in runApp, but this should be the primary, if not only, method of OreSim.

OreSim also renders the game in drawActors and drawBoard, which should be moved to a seperate Renderer class to enable better Cohesion. The method drawActors should also be broken up and modified, as it is currently both drawing the Actors and then creating them by reading the decoded MapGrid. The method of creating Actors should be moved to a separate Creator class, or better yet, be split among several classes using a Factory pattern to enable different creation rules for different Actors. Following this idea of Actors controlling their own methods, the moveOre method should be moved to the Ore class itself, as it makes sense for entities to control their own movement handling.

The performance of updating the statistics of the program is also done within OreSim, but as these are complex methods they should be moved to a separate Statistics class as a Pure Fabrication to act as a helper. The same thinking should apply to the interpretation of user input, as currently GGKeyListener is being used by OreSim to take advantage of keyPressed and keyReleased. This implementation of GGKeyListener and its methods should be moved to a seperate Controller class to allow different user input interpretations by the different vehicle classes. Having the user input tied to OreSim also highly couples any future usage of it to OreSim.

The private classes Target, Ore, Pusher, Bulldozer, Excavator, Rock, and Clay are all inner classes of OreSim and cannot be accessed by any other classes, even the other inner

classes. They all have high dependencies on OreSim as they are created within and by it, with Pusher even directly using OreSim's private variables as inputs for its own methods.

This creates several issues for Bulldozer and Excavator and any other future vehicles, as they now cannot inherit from Pusher to use its automatic movement code nor obstacle interaction code. It creates the need for duplicate code, more complex maintainability, and limited future reusability between the vehicle classes.

The public inner Enum class ElementTypes contributes to potentially forcing the creation of Actors and the updating of statistics to be coupled within OreSim. However, this class unfortunately cannot be removed due to its use in the unmodifiable testing classes.

Part 2: Proposed new version

Whilst it is true that OreSim, which is a GameGrid child instantiated by Driver, functions as an information expert by having access to all the Actor instances that exist within oreSim implicitly, we can abstract much of what it once did into isolated classes to promote the single responsibility principle. This assigns OreSim the main responsibility of being a Creator in GRASP principal terminology (although it will inevitably be an information expert due to engine design), as the external JGameGrid library assumes that all Actors have a GameGrid, which is achieved through the GameGrid creating the Actors. This will increase the cohesion of the whole program, and the subsequent design benefits will be discussed as the design class diagram of the new system is analysed.

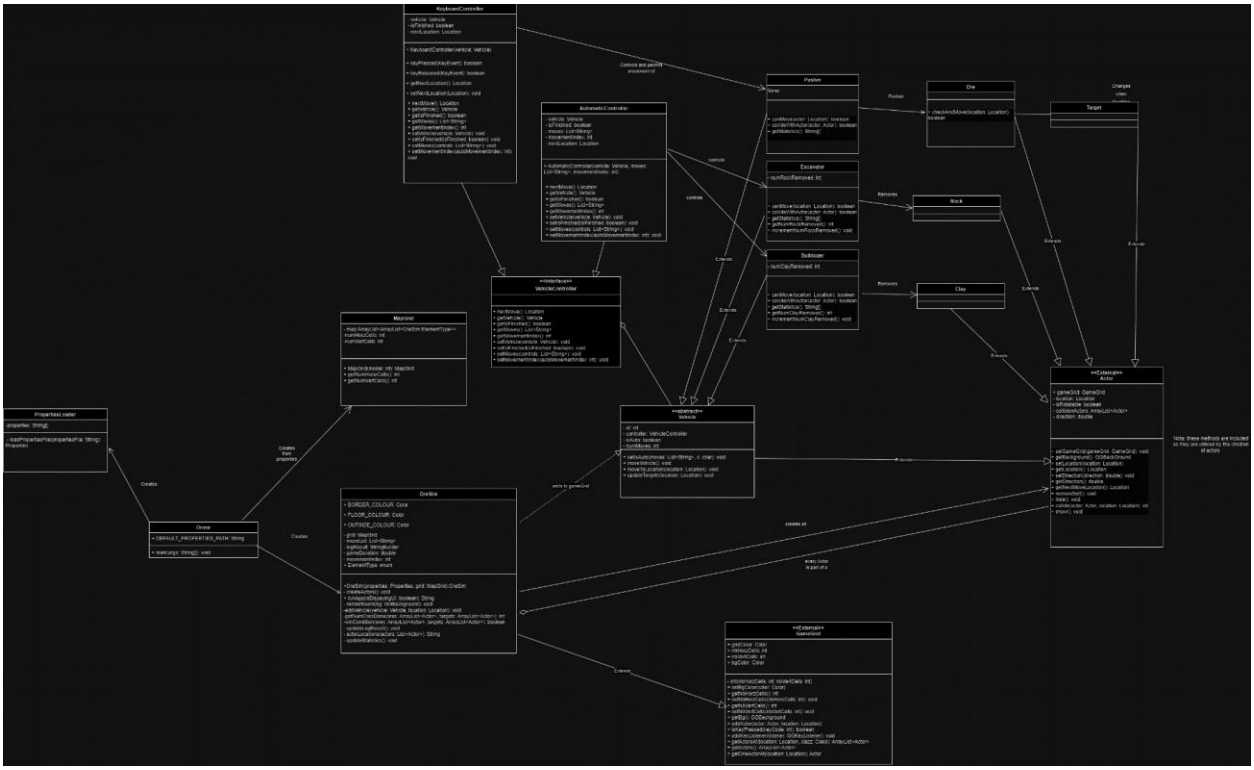

Figure 1: Design Class Diagram for the refactored OreSim Project

Goals of the new solution

This solution intends to abstract away many of the responsibilities that overburden OreSim and make it difficult to define its specific responsibility, as we should be able to within

GRASP principles to solve future problems. Additionally, instead of pointlessly making interfaces and objects to handle logic the existing JGameGrid framework already supplies, we will make use of the methods within Actor and GameGrid, but simply delegate such methods to relevant child classes.

Vehicles

With the intention to add more vehicles than a simple pusher, it makes immediate sense to abstract vehicles into their own class. Our new solution uses an abstract Vehicle class, as although vehicles can have different purposes and functions within the simulation, they will have a uniform set of actions that apply to all of them, such as setting if they are controllable or automatic, checking whether they are able to move, moving, and colliding with actors. This increases the cohesion of the Bulldozer, Pusher, and Excavator class, as now their exact purposes and functions are explicitly defined by an abstract class, and there is no code repetition. To aid with the cohesion of the solution, VehicleController interface was added which is implemented by two classes, AutomaticController and KeyboardController. This follows the GRASP principle of a Controller and is delegated to telling and handling where a vehicle can next move. This reduces the complexity of the vehicle classes, and deals with all events related to the use case of moving vehicles, eliminating copied code and generalizing such behaviors. This is a pure fabrication, as it is beyond the domain problem, but it streamlines our solution by easily managing all vehicles. The implementation of controllers also supports the indirection GRASP pattern, as it reduces the coupling between oreSim and each vehicle by mediating the inputs to them.

Entities

We defined an “Entity” package which describes actors on the GameGrid OreSim. This is to abstract them away from that class, instead of being defined internally. This enhances the open / closed principle by still having the security of the Actor methods but being easier to understand and modify. As of now, Clay and Rock have no functionality as Bulldozers and Excavators are responsible for calling their removeSelf method within the superclass, but their individual class makes interactions with them clearer, whilst still retaining their functionality as actors. In our design class diagram (figure 1), we illustrate what methods from the superclass we use to interact with these objects. Ore has received additional changes however, with it checking if it can move when a pusher attempts to move it and changing its sprite when moved. This increases the cohesion of the class, as now Ore is more clearly responsible for the actions defined within the domain that

exclusively pertain to it, and reduces coupling, as a pusher no longer moves the Ore and checks for precisely what the Ore should do, Ore is responsible for that itself. Pusher doesn't have to know anything about Ore, the only connection is that Pusher tells Ore to check if it can move.

OreSim

We have managed to reduce OreSim to just running the app with the runApp function as seen in the design class diagram, as well as displaying UI and creating the actors (vehicles and objects), and handling specific things it should be responsible for related to its function as a simulator in the domain problem: updating the win condition and handling the log of movements and interactions. As a result, it is now appropriately focussed and therefore has high cohesion. It is perhaps not ideal that it still acts as a factory and an information expert, but we found it impossible to create actors outside of OreSim, since they have a protected method for setting which gameGrid they are a part of that depends on if the gameGrid (OreSim) creates them.

Flaws with the proposed solution

Regardless of the attempt to abstract the functions of oreSim into good design paradigms, the fact that it is a gameGrid remains. The problem with this is that it unavoidably must become a factory pattern object whilst also being an information expert, as because all actors on the grid are fundamentally tied to the grid (having a location, etc), they must be created within the grid also, whilst the gameGrid also has all information about every actor on its grid. As such, Actors call gameGrid methods as they know they are a part of oreSim, and since all actors must do this (to know if they are colliding, for example), there is a high degree of coupling between Actor and gameGrid that cannot be avoided due to the design of the engine. The implication of this coupling is that it is difficult to trace back interactions between objects, as they are done through calls to gameGrid methods, which in turn, damages the cohesion and the validity of the GRASP principles we wish to implement. It is, however, difficult to see an alternative scenario to this in the realm of game engines, but the ideal scenario would be to make use of an engine where positions and locations are relative, and whilst there would be universal coordinates, it would not be the responsibility of an object like oreSim to know all of these and deal with them.

Future extensions

By increasing the cohesion of the solution through a defined and generalized implementation of vehicles displaying the GRASP principle of polymorphism, it is very simple to add new vehicles and behaviours for vehicles, as necessary behaviours are designated through the use of the VehicleController interface and Vehicle abstract class,

whilst also allowing for them to be tailored to specific new needs as required. The Actor class, whilst lacking cohesion with its breadth of methods, combined with our designation of obstacles to a specific package, makes it clear how they can be extended upon with new behaviours or objects. Now that it is explicit what OreSim is responsible for, being a factory and information expert in GRASP, means that simulation behaviour can additionally be extended upon with little overhead, other than needing to understand the functionality and limitations of the JGameGrid library.

Part 3: Proposed Design of Extended Version

Additional Mining Machines

The specification requested the ability to simulate additional vehicles outside of the pusher. To accommodate the simulation of different vehicle types, an abstract class “Vehicle” was added to implement the common behaviours between the mining machines. This allowed the Excavator and Bulldozer to be implemented by inheriting from the vehicle class, where only vehicle-specific behaviours need to be included in the subclasses. The generalisation-specialisation relationship between Vehicle and its subclasses means that (after instantiation) OreSim only needs to interact with Vehicle and not its subclasses, thus reducing coupling.

Operational Statistics

The recording of operational data was also requested. The responsibility of recording operational statistics was delegated to the Vehicle class in accordance with the information expert principle: each vehicle “knows” when it moves or performs an action so is able to record this information. This allows easy retrieval of the operational statistics: since OreSim keeps a record of each vehicle it is able to “ask” each vehicle for its statistics at the conclusion of the simulation.

Future Development

The specification also requested the extended version be extendable to the implementation of additional features in the future.

Multiple Machines

The provided automatic movement instructions refer to a vehicle type and not its ID; other than this, multiple machines are already possible in the extended version of OreSim. Vehicles are given an ID upon creation, meaning that different vehicles are able to be assigned different IDs. Furthermore, the abstract Vehicle class means that all vehicles can be controlled by OreSim (because Vehicle.moveVehicle() can be called on each of them) and that the operational statistics can be retrieved from all vehicles (because Vehicle.getStatistics() can be called on each). Finally, each vehicle can be assigned its own unique controller thanks to the implementation of the VehicleController interface.

Additional Machine Types

The addition of other types of vehicles is made possible by the Vehicle class. A new type of machine would simply inherit from Vehicle and define its own vehicle-specific behaviour in its own class, as do Pusher, Bulldozer, and Excavator.

More types of obstacles

The existing obstacles, Rock and Clay, inherit from Actor; an additional obstacle would do the same and would only need to provide a new sprite to represent it. Additional logic would not be required as this is handled by Vehicles.

Additional Control Methods

The adaptor pattern features in the design of OreSim in the implementation of the VehicleController interface. An object that implements this interface is included in each vehicle; to create an additional control method, another class that implements VehicleController could be created, and this would be compatible with Vehicle.