SMD Project 1:
By Alex de Vryer, Aidan Butler and Mithun Rithvik Ayyasamy Sivakumar

Analysis of the Current Design:
Observing the current design of OreSim, we can see that it does not follow general GRASP design principles. As explained below, we take issue with multiple decisions taken in the original design of OreSim:

Low Cohesion:
Firstly, the OreSim class has very low cohesion. The OreSim class contains code for multiple differing responsibilities, such as running the game, checking if piece moves are legal as well as defining the classes for the individual game pieces/vehicles. This makes the class very bloated and the code hard to understand as OreSim has no single responsibility; rather, it has multiple responsibilities. The OreSim class would instead have higher cohesion if these responsibilities were delegated to different classes.

Lack of Polymorphism:
Additionally, the internal classes in OreSim do not currently support polymorphism. There are multiple related classes within the game, for example the Pusher, Excavator and Bulldozer are all Vehicles and the Ore, Rock and Clay can all be classified as Elements. The current design does not reflect that there is such a relationship between classes, meaning that code cannot be re-used between the current classes even though they have similar behaviour.

For example, the autoMoveNext() function in the Pusher class in the original design could be re-used for the Bulldozer and Excavator class, but the lack of polymorphism means this currently cannot occur. Additionally, the moveOre() and canMove() functions (for Ores and Pushers respectively) have very similar behaviour and could also be potentially delegated to a superclass which could validate movement for all movable objects.

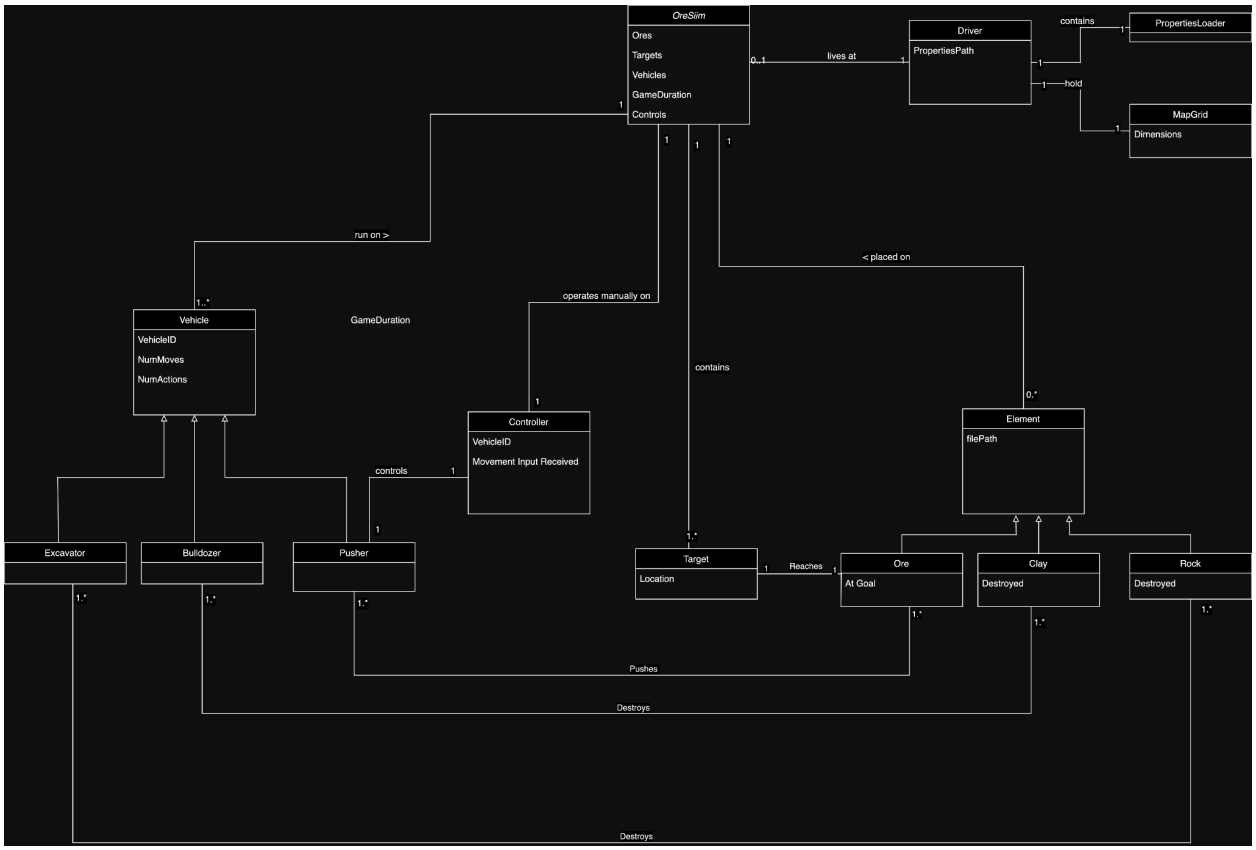
Lack of Protected Variation:
The code also does not currently support protected variation. At the moment, there is no possibility for multiple vehicles to be added to the game (due to the lack of ArrayLists to store vehicle objects) and there is also no possibility of adding new elements or vehicles without extensively rewriting the code.

Logic Errors and Poor Data Structure Choices:
Furthermore, there are also some logic errors in the existing code. For example, if the game is in auto mode, a player can still use the arrow keys to control the pusher using the keyPressed() function, even though it should only be controlled by the pre-defined movement sequence. Additionally, the current logic in the moveOre() function allows the player to move an ore into a space which is currently occupied by another vehicle, which is undesired behaviour.

Moreover, the data structures used to keep track of the Ores and Targets are inflexible. Arrays are used instead of ArrayLists, which means that we require another variable (the size of the

array) to iterate through the arrays which is less flexible than being able to iterate through all elements of the array (which ArrayLists permit).

Proposed new design of the Simple Version:
We created a domain class diagram to capture the basics of our planned design for the new version as shown below. Our initial objectives were to make the OreSim class more cohesive, make the inner classes into standalone classes and delegate vehicle movement responsibilities to a completely fabricated class.



From the domain diagram, we then created the design class diagram which our fully-working solution is based off of (please refer to the attached PDF's to view the Design Class and Design Sequence Diagrams - they are too big to fit onto a document page). Compared to the original design, we have made several changes such as:

Increasing Cohesion in OreSim through Information Expert and Delegation:
Our first major task in refactoring the original design is to increase the cohesion of the OreSim class as it is currently very bloated. We achieved this through creating separate classes for all the internal subclasses in the OreSim file. We created new classes for the Pusher, Excavator, Bulldozer, Target, Ore, Rock and Clay objects as seen in the design class diagram. Furthermore, by Information Expert, we also delegated the responsibility of moving the Pusher

and Ore objects (contained in canMove() and moveOre()), as well as the movement logic for the Pusher (contained in autoMoveNext() and keyPressed()) which were originally contained in OreSim to the Pusher (and by extension the Vehicle superclass) and Ore (and by extension the Movable interface) classes respectively. As these objects move across the board, they should theoretically be the classes to control their own movement. This decision would also further increase cohesion within the OreSim class.

Use of Polymorphism and Interfaces for Vehicles and Elements:
As shown in our design class diagram, we used polymorphism by creating a Vehicle superclass (of which Pusher, Bulldozer and Excavator would extend from) and an Element superclass (of which Ore, Rock and Clay would extend from). This allows us to implement common logic across all vehicles, such as movement logic in the moveObject() method to deliver on the requirement that additional mining machines can be controlled.

This also allows us to deliver on one of the requirements of the new game; the ability to keep track of vehicle movement and action statistics. As shown in the design sequence diagram, each vehicle keeps track of each move and action it makes through the numMoves and numActions attributes and the getStatistics() abstract method in the Vehicle superclass. By Information Expert, we have left the creation of the statistics file to the OreSim class as OreSim contains all vehicles. At the end of the game, these statistics are fetched by OreSim from these classes.

We also implemented two new interfaces; a Movable interface which contains the canMove() signature (of which Ore, Bulldozer, Excavator and Pusher would implement) and a Destroyable interface which contains the destroy() signature (of which Rock and Clay would implement). This helped us define behaviour which is common to all objects which can be moved and destroyed and further delegated responsibilities originally assigned to OreSim to other classes and interfaces. It also helped us implement the logic required to allow Bulldozers and Excavators to move as per the new requirements.
Delegation of Controller Responsibilities through Pure Fabrication:
Furthermore, through Pure Fabrication, we created AutoController and ManualController classes to manage movement logic for all vehicles. To avoid the Vehicle classes becoming bloated and incohesive, we delegated the processing of movement input (be it through a key press or a list of movement commands) to the Controller class. This means that the Vehicle class only receives the coordinate where the vehicle will move next, the Controller class processes whether the input command was intended for the specific vehicle and returns the direction the vehicle should move in.

These two classes were created through polymorphism (as both these classes extend from a Controller superclass). We also used generics in these classes to allow for a variety of inputs to be handled by the classes (e.g. for ManualController, we would be able to send an Integer keycode, a specific KeyEvent, a String or any other object to move the Vehicle) which allows for Protected Variation for a variety of input devices to be used in future versions.

OreSim's Current Responsibilities as outlined by Information Expert and Creator:
We did, however, leave a bit of logic in the OreSim class. For example, we have left the responsibility of creating each individual vehicle or element in the OreSim class when it is constructing the board. By Creator, OreSim compositely aggregates as well as possesses the initialising data for each individual game object, so we kept this responsibility in the OreSim class in the drawActors() method. Furthermore, by Information Expert, we left responsibilities such as drawBoard(), checkOresDone(), updateStatistics(), keyPressed(), actorLocations() and updateLogResult() in OreSim as it contains all the information required (i.e. OreSim compositely aggregates all Vehicles, Elements and Targets as well as defines the structure of the game board) to carry out these responsibilities. Removing these functions from OreSim and placing them in other classes would require high coupling.

Proposed design of the Extended Version:
The use of Polymorphism and Protected Variation in our updated version of the game allows for the planned additions to be easily implemented in future:

Multiple Machines During Gameplay:
As shown in our design class diagram, OreSim can compositely aggregate many Vehicles in this new version. We implemented a Vehicle ArrayList in OreSim which allows us to have multiple different vehicles on the board at any one time. Furthermore, we also included a Constructor method in each Vehicle subclass which allows for a custom ID to be set for that particular vehicle to allow for more vehicles in the future.

More Types of Machines and Obstacles:
As discussed previously, we used polymorphism in our new design to create an abstract Vehicle class from which all current vehicles (Pusher, Bulldozer and Excavator) extend from as well as a Movable interface which all vehicles implement. We also created new Controller classes through Pure Fabrication to control movement logic. If we wanted to add new vehicles in the future, this would be much easier as most of the code can be easily reused; we would only need to write a canMove() function specific to that vehicle. The same applies for new Obstacles (or Elements as we named them), a superclass already exists for these objects and through polymorphism and inheritance from the Movable and Destroyable interfaces, we can easily create and define behaviour for new elements.

Various Ways of Controlling Machines:
Through our newly-defined Controller class, we have allowed for multiple different inputs to be used in future versions of OreSim. In the ManualController class, we have used Java Generics for the directional keybinds (e.g. which button/input is received for Left, Right, Up and Down commands respectively) and the getMove() function. This means that instead of the current Integer keybinds which are being used in this version, we could add other inputs (e.g. from other input devices such as a joystick) to control the machines.

Additionally, we have allowed easy extensibility of the automatic controller for future versions. The moveObject() method in Vehicle allows for a direct movement instruction to be sent to the

vehicle which can then be analysed by an AutoController object, meaning that intelligent agents or autonomous planners would be able to map out the best paths or send input to the vehicle directly.