SWEN 30006 Project 1
Erich Wiguna 1389444
Jessica Aurelia Wijaya 1364255
Reagen Purnama 1389448

P1: Analysis of the Current Design

The current design of OreSim exhibits several concerns when analyzed through the General Responsibility Assignment Software Principles (GRASP):

The OreSim class manages many aspects of the game, from handling user inputs and managing game states to controlling all types of vehicle movements and interactions. This tight coupling makes the system less flexible and more fragile, complicating the addition of new features such as different types of vehicles or new gameplay mechanics.

Other than that, OreSim encompasses a wide range of functionalities that are not necessarily related, such as UI control, game logic, and entity management. Low cohesion within this class makes it harder to understand, maintain, and scale. OreSim is also directly responsible for creating and managing many different objects (vehicles, obstacles, etc), which should ideally be handled by separate classes to promote modularity.

Moreover, Acting as the central controller, OreSim directly handles too many decisions and actions, which overloads its responsibilities. Therefore, these issues could significantly hinder the system's ability to integrate new features, such as multiple machines with distinct behaviors or alternative control schemes, without substantial refactoring.

P2: Proposed New Design of the Simple Version

To address these concerns, the simple version of OreSim can be refactored using a factory pattern. Implement a factory pattern for creating game entities like vehicles and obstacles. This reduces the OreSim class's responsibilities by delegating object creation to specialized factories, enhancing modularity and scalability.

These changes aim to enhance modularity, reduce coupling, and increase cohesion, thereby making the system more robust, easier to maintain, and ready for future extensions.

P3: Proposed Design of the Extended Version

For the extended version that includes multiple machines and varied control schemes, Implementation of Strategy pattern can be used for vehicle movement and interaction behaviors. This allows each vehicle type to encapsulate its unique behavior that can be changed dynamically at runtime, supporting varied and extensible vehicle actions.

Moreover, an Observer pattern can be utilized to monitor and respond to game state changes (like scoring). This pattern facilitates a decoupled way to update game elements or other parts of the system in response to game events.

These design enhancements will support the introduction of new machine types and interaction models while maintaining system integrity and extensibility. The design leverages established patterns to ensure that adding or modifying features is feasible without comprehensive rewrites.

To address the identified concerns, we propose to use strategy patterns such as Factory, Observer, and Strategy pattern. Not only that these patterns would improve the program design, but it would also improve efficiency and would prepare OreSim for future innovations.

Software Models

Part 1: From Problem Domain to Domain Model
In order to establish a solid understanding of the domain, we've crafted a partial domain model (refer to Figure 1) to comprehensively capture the operational needs of OreStrike Corporation's OreSim software.

Part 2: From Domain Model to Design Model
Our partial design diagram (refer to Figure 2) reflects the actual implementation of the Ore Simulation system. The sections below provide a detailed account of the design decisions and modifications made to the design model.

2.1. Vehicle Movement Control
To manage the movement of vehicles within the simulation, we introduced the concept of a 'Moveable' interface. This interface defines a method 'canMove()' which allows vehicles to determine whether they can move to a specified location. By implementing this interface in vehicle classes such as 'Pusher'. 'Bulldozer', and 'Excavator', we achieve a flexible and decoupled approach to controlling their movement.

2.2. Actor Abstraction
We abstracted the common properties and behaviors of actors in the simulation by introducing the 'Actor' class. This class serves as a base for various types of actors such as 'Target', 'Ore', 'Rock', 'Clay', and 'Obstacle'. By encapsulating shared functionality within the 'Actor' class, we promote code reuse and maintainability.

2.3 Vehicle Hierarchy
We established inheritance of vehicles in the simulation by defining abstract classes through the parent class - 'Vehicle' and specific vehicle types as the child classes like 'Pusher', 'Bulldozer', and 'Excavator' that extend it. This hierarchical structure allows for consistent behavior and easy extension of vehicle types for unique behaviors in the future.

2.4 Automatic Movement

To support automatic movement of vehicles based on predefined instructions, we implemented the ‘autoMoveNext()’ method in vehicle classes. This method executes the next movement instruction according to a predefined control sequence, facilitating automated vehicle operation within the simulation.

2.5. Collision Detection

To prevent collisions between vehicles and obstacles, we implemented collision detection logic within the ‘canMove()’ method of vehicle classes. This logic checks for obstacles such as rocks, clay, and other vehicles in the intended movement path and prevents movement if a collision is detected.

2.6. Game Logic

We encapsulated the core game logic within the ‘OreSim’ class, which serves as the main orchestrator of the simulation. This class manages the grid layout, actors, player input, automatic movement, game duration, and simulation period, providing a centralized control mechanism for the entire simulation.

2.7. Properties Loading

To facilitate configuration of simulation parameters, we introduced the ‘PropertiesLoader’ class. This class is responsible for loading properties from a configuration file, allowing for easy customization of simulation settings without modifying the source code.

Conclusion

By adopting these structured design improvements, the OreSim simulation software will be better positioned to accommodate future expansions and enhancements. These changes aim to reduce system coupling, improve cohesion, and enhance the flexibility and maintainability of the codebase, ensuring that OreSim evolves in response to new challenges and technological advancements.

This refined structure and the inclusion of specific software engineering principles will ensure your report is comprehensive, focused, and aligned with the objectives of this project, showcasing a thorough understanding of both theoretical and practical aspects of software design and modeling.

Appendices:

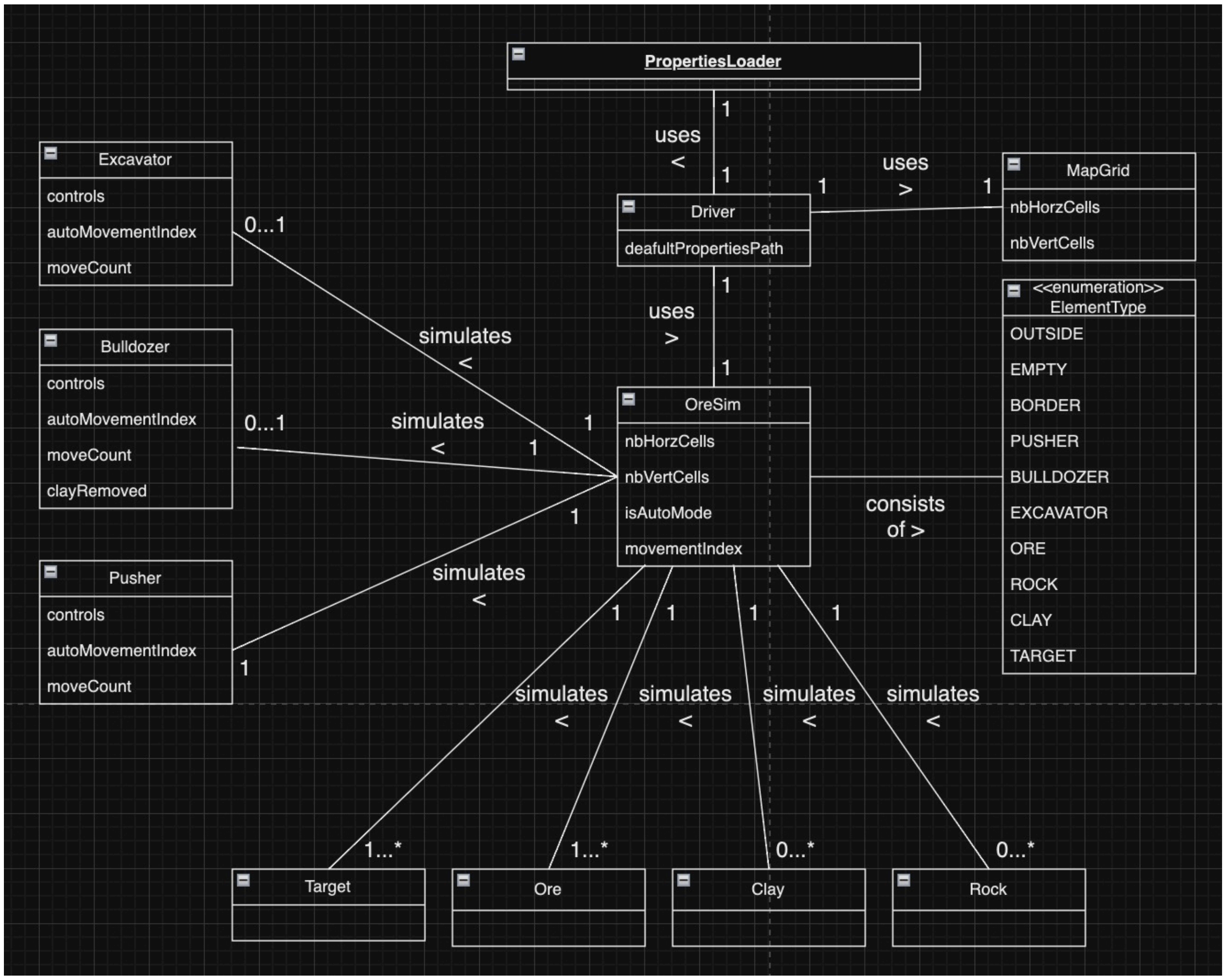

Figure 1: Partial domain diagram describing the OreSim based on the requirements in the project specification

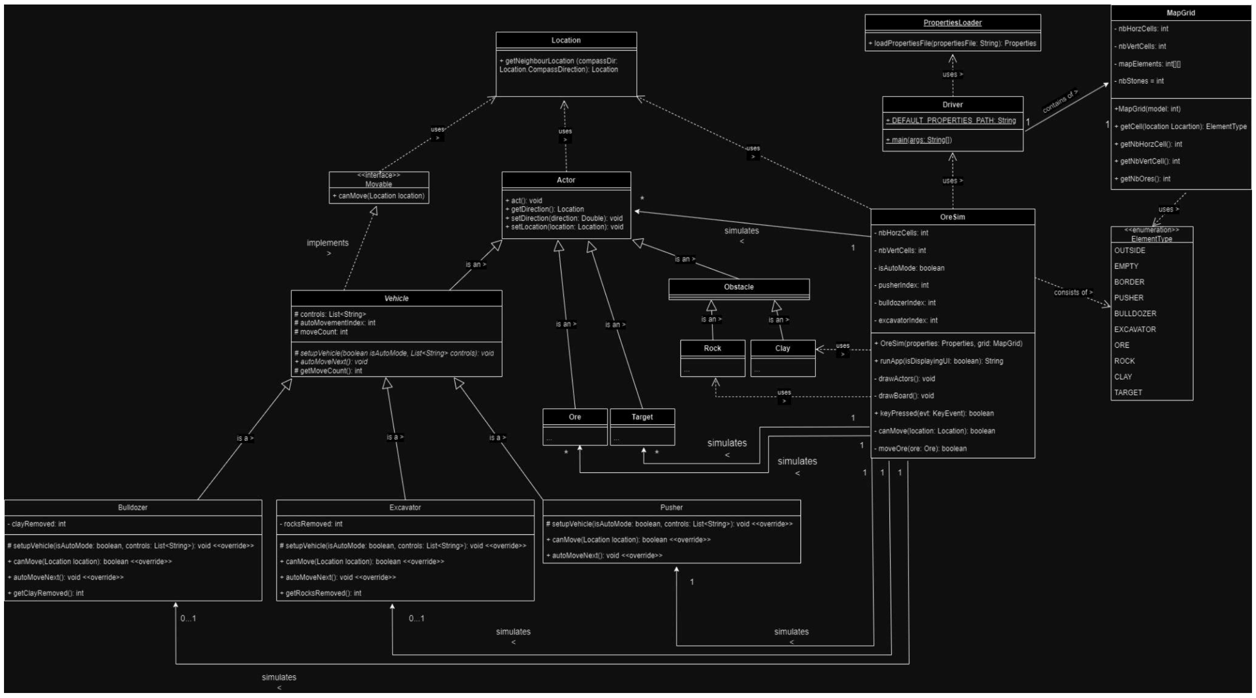
Location
+ getNeighbourLocation (compassDir: Location.CompassDirection): Location
uses >
uses >
uses >
<<interface>> Movable
+ canMove(Location location)
Actor
+ act(): void
+ getDirection(): Location
+ setDirection(direction: Double): void
+ setLocation(location: Location): void
implements >
is an >
Vehicle
# controls: List<String>
# autoMovementIndex: int
# moveCount: int
# setupVehicle(boolean isAutoMode, List<String> controls): void
+ autoMoveNext(): void
# getMoveCount(): int
is an >
is an >
is an >
Obstacle
is an >
is an >
Rock
...
Clay
...
Ore
...
Target
...
PropertiesLoader
+ loadPropertiesFile(propertiesFile: String): Properties
uses >
Driver
+ DEFAULT_PROPERTIES_PATH: String
+ main(args: String[])
contains of >
1
1
MapGrid
- nbHorzCells: int
- nbVertCells: int
- mapElements: int[][]
- nbStones = int
+MapGrid(model: int)
+ getCell(location Locartion): ElementType
+ getNbHorzCell(): int
+ getNbVertCell(): int
+ getNbOres(): int
uses >
uses >
OreSim
- nbHorzCells: int
- nbVertCells: int
- isAutoMode: boolean
- pusherIndex: int
- bulldozerIndex: int
- excavatorIndex: int
+ OreSim(properties: Properties, grid: MapGrid)
+ runApp(isDisplayingUI: boolean): String
- drawActors(): void
- drawBoard(): void
+ keyPressed(evt: KeyEvent): boolean
- canMove(location: Location): boolean
- moveOre(ore: Ore): boolean
consists of >
<<enumeration>> ElementType
OUTSIDE
EMPTY
BORDER
PUSHER
BULLDOZER
EXCAVATOR
ORE
ROCK
CLAY
TARGET
simulates <
*
1
uses >
uses >
simulates <
simulates <
1
1
*
*
1 1 1
is a >
is a >
is a >
Bulldozer
- clayRemoved: int
# setupVehicle(isAutoMode: boolean, controls: List<String>): void <<override>>
+ canMove(Location location): boolean <<override>>
+ autoMoveNext(): void <<override>>
+ getClayRemoved(): int
Excavator
- rocksRemoved: int
# setupVehicle(isAutoMode: boolean, controls: List<String>): void <<override>>
+ canMove(Location location): boolean <<override>>
+ autoMoveNext(): void <<override>>
+ getRocksRemoved(): int
Pusher
# setupVehicle(isAutoMode: boolean, controls: List<String>): void <<override>>
+ canMove(Location location): boolean <<override>>
+ autoMoveNext(): void <<override>>
0...1
0...1
1
simulates <
simulates <
simulates <

Figure 2: A Design Class Diagram of OreSim classes and elements.

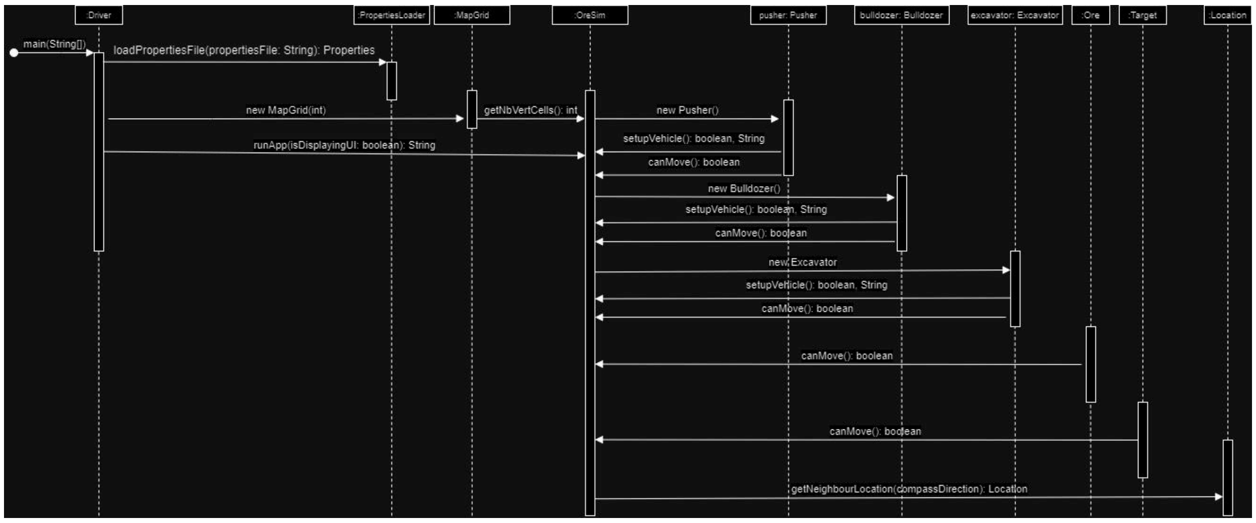

Figure 3: A Dynamic Design Model of OreSim