SWEN30006 PROJECT 1 REPORT

Workshop [Thu13:00] Team 02

Jess Lee (1260948)
Bella Kwon (1346599)
Samuel Imanuel Gunawan (1371743)

An analysis of the current design:

Considering from the perspective of the GRASP principle, the current design of the OreSim program lacks in few principles, potentially hindering the extensions of the new features.

Firstly, the presence of low cohesion in the OreSim codebase prevents its reusability and maintainability. In the original model, the OreSim class serves as a single entity with overly wide responsibilities including game flow control, actor management and statistical calculation. Such ambiguity in the class purpose results in low cohesion, reducing the modularity and readability of the program. Moreover, as it holds majority of the data structures, there is a high coupling between the OreSim class and its inner classes for game objects (eg. Pusher, Bulldozer, Rock, etc), granting unnecessary access to other attributes and methods. As a result, any modifications or extensions to the program risk destructing disparate attributes and functionalities within the OreSim class, complicating the addition of new features and actors for further development of the game dynamics. The addressing of low cohesion and high coupling are vital as they significantly violate the Open-Closed principle.

The current design also neglects polymorphism and protected variation, bringing about an inadaptable and error-prone codebase. In particular, the handling of game actors as inner classes within the OreSim class hinder the encapsulation of shared behaviors and attributes among closely related actor types, further preventing their uniform modulation. For example, although the inner classes Pusher, Bulldozer, Excavator share common characteristics such as movement logic and rotation, handling those closely related behaviors based on their element type is a complex process in our design. Such deficiency in polymorphism not only leads to low flexibility and extensibility of the program, but also significantly escalates the risk of code duplication and inflexibility when it comes to extensions of new actors with similar functionalities, resulting in low protected variation.

As a result, numerous data structures and responsibilities are crammed together in limited classes, prompting the absence of information expert principle.

Proposed new design of the simple version
In an effort to address the above flaws, the design was restructured to decouple the OreSim inner classes and reorganize them into distinct public classes. Enhancing modularity and extensibility, this restructuring led to the creation of two new abstract classes: OreSimObject and OreSimMachines. The OreSimObject class, which extends from the base Actor class, serves as a foundational component for all game objects, while OreSimMachines extends from OreSimObject, which are responsible for the movement mechanics of objects thus adhering to GRASP's controller principle OreSim class to focus handling the system's operation.

Building upon this design, methods such as **`setUpMachine()`**, **`moveOre()`** and attributes such as **`autoMoveIndex`** have been moved into corresponding classes for better encapsulation of related functionality. Furthermore, implementing abstract methods like **`autoMoveNext()`**and **`canMove()`** allows subclasses (e.g. Pusher and Excavator) to provide their own implementations, providing low coupling through polymorphism. Similarly, a new helper method named **`moveMachine()`** was introduced. This method abstracts the responsibility of machine movement from **`autoMoveNext()`**, following the Single Responsibility Principle (SRP) of Object-Oriented Programming (OOP) Design, and enhancing the reusability and maintainability of the code.

This new architecture not only facilitates future enhancements through the principle of inheritance but also simplifies the integration of new object types or machine features.

Proposed design of the extended version:
To implement *feature 1,* additional mining machines needed to be incorporated in the game. To achieve this, the **`autoMoveNext()`** method was refactored from the Pusher class to the OreSimMachine abstract class. This enabled overriding each Pusher, Excavator and Bulldozer classes, and the application of specific implementations, such as the behavior of a machine and its valid moves.

As part of this logic, **`clearObject()`** and **`canMove()`** methods were designed to accept all OreSimObject types, and were implemented inside the OreSimMachine class to be used by multiple machine types. **`clearObject()`** method retrieves a list of objects at a location and defines whether a specific machine can clear this object. Hence following the project specifications, the current program only allows Excavator to clear a Rock, and Bulldozer to clear a Clay. Additionally, the **`canMove()`**abstract method was refined to shift from evaluating each object type individually at a given location to a more streamlined check for the presence of any OreSimObject and other specific object types as needed. The current program now includes logic to determine whether Pusher, Excavator and Bulldozer can move to certain locations in an encapsulated and polymorphic manner.

Feature 2 involved recording the operational data to a file, indicating the total number of moves and total number of obstacles encountered by each machine. For improved modularity and encapsulation, the logic to print the statistics was implemented in a separate method, **`updateStatistics()`**. Similar to the design of feature 1, a dedicated method for this feature enables any future changes to be made in a single location. The key counter variables (**`machineMoves`** and **`objectRemoved`**) have been retained in the OreSimMachine abstract class to centralize tracking of important statistics related to machine behavior. This logic of incrementing the **`machineMoves`** and **`objectRemoved`** variables have been integrated inside each overriding **`canMove()`** method, incrementing when the specific machine has successfully moved and/or removed an object in its current location.

Ultimately, the design decisions to our proposed new design along with the extended features improves code reusability, maintainability, and readability, contributing to a more robust and adaptable codebase to accommodate future extensions and development. While the current version has been refactored to significantly improve the program's design, it may be further developed by creating an array of each machine type instantiated in OreSim. This way, when we add a new machine, we can simply append the machine to the array and iterate through to implement all common methods like **`autoMoveNext()`** and **`canMove()`**, rather than manually calling each method per machine type. This approach can also be applied to OreSimObjects for future enhancement.