

SWEN30006 Design Analysis Report - Project 1 Mine Maze

Part 1: From Problem Domain to Domain

To ensure we had a solid understanding of the MineMaze domain, we constructed a domain model (see Figure 1) to formally capture the business requirements of the game. The diagram shows the core conceptual entities, their associations, and multiplicities.

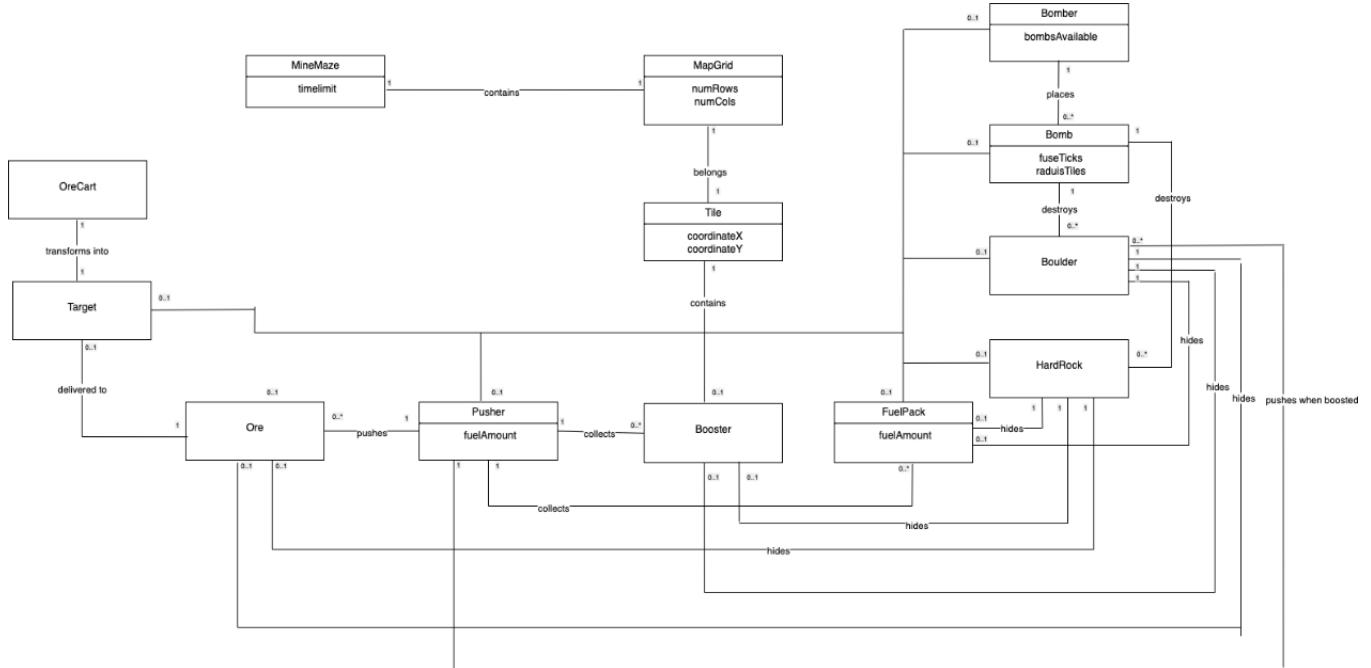


Figure 1: Domain model of MineMaze

Part 2: Analysis of Existing System

In the existing implementation, MineMaze acts as a God class. The game's control flow is tightly linked to the user interface, meaning that system operations cannot be separated from user input. For example, user actions are handled directly in the `mouseEvent` method.

Board rendering and state updates are managed by methods such as `drawBoard` and `drawActors`. Movement planning and interactions are executed through methods like `canMove`, `canMoveWithOrePushing`, `executeNextPathStep`, and `guidePusherToLocation`. The main game loop, located in `runApp`, advances the simulation and processes ticks. Variables like `currentPathIndex` and `autoMovementIndex` are used to schedule movement and control logic within the central `MineMaze` class.

Game objects such as `Boulder`, `Ore`, `Hardrock`, and `Target` all extend from the decompiled `Actor` class. However, these objects are implemented passively, they are primarily used for display

and do not contain their own interaction logic. Instead, their behavior is controlled externally by MineMaze.

From a GRASP perspective, the system has low cohesion: MineMaze is responsible for a wide range of tasks, from object creation to movement computation and log assembly. The system also suffers from high coupling, as MineMaze maintains references to nearly every game entity. Additionally, MineMaze is tightly bound to the MapGrid, the UI framework (GameGrid), and the properties configuration through the PropertiesLoader class.

There is no indirection in the design; MineMaze directly invokes domain behaviors, rather than using an intermediary to separate UI events from domain operations. The Information Expert principle is also largely ignored because responsibilities are not delegated to the game entities. Instead, the controlling class, MineMaze, determines when objects can move and when the game is won or lost.

While MineMaze does act as a creator, instantiating objects and parsing configuration, it also handles too many unrelated tasks. Interactions between objects, such as the ore, boulder, or pusher, depend on background color checks and conditional statements instead of using polymorphism. This weakens the system's ability to support protected variations.

Overall, the presentation, control, and domain logic are all tightly coupled. This design makes the existing system difficult to maintain, test, and extend.

Part 3: From Domain Model to Design Model

Transitioning from the domain to the design model means moving from real-world concepts to a structured software solution. In developing the design class diagram, including the three new features, we applied GRASP principles to ensure thoughtful responsibility assignment, low coupling, high cohesion, and clear control. This model (see Figure 2) clarifies how each feature fits into and supports the overall project, both structurally and in implementation.

Feature 1: Bomber

Involved Classes & Relationships

The new design introduces an abstract parent class called Machine and its child class Bomber, which manages a collection of Bomb objects. The Bomber is responsible for placing bombs, tracking their availability, and returning to its starting position. Each Bomb is responsible for its own lifecycle, including activation, ticking, explosion, and interacting with obstacles (e.g., Boulder, HardRock). The system also uses interfaces such as Useable, promoting flexible interactions. MapGrid includes bombs and obstacles, and MineMaze controls the Bomber and

manages game flow. MineMazeCreator is the main GRASP Creator for actors (including Bomber and Bomb), as it has the necessary map and configuration knowledge.

GRASP Principles Applied

Creator: MineMazeCreator is responsible for instantiating Bomber and Bomb, as it has the necessary map and configuration knowledge (Information Expert). Bomber manages bomb placement and state but does not instantiate itself or bombs directly.

Controller: MineMaze delegates bomb placement and movement to Bomber.

Polymorphism: Bomber and Pusher share the Machine superclass so movement and control logic can be reused.

Information Expert: Bomb knows how to explode and affect its surroundings; Bomber knows its own state and bomb count.

Low Coupling: MineMaze interacts with Bomber and Bomb through their interfaces or abstract types.

Potential Improvements / Alternatives

One potential improvement is to use a Destructible interface for obstacles, so that new destructible types can be added without modifying the Bomb class, thereby promoting protected variation. Additionally, pure fabrication can be applied by introducing a BombManager class responsible for handling bomb timing and explosions, which would reduce the responsibilities of the Bomber. Finally, if more machines are to be added in the future, a MachineFactory could be considered for the instantiation of machine objects, further supporting extensibility and maintainability.

Feature 2: Fuel

Involved Classes & Relationships

The fuel system centers around the Pusher, which has a fuel attribute that decreases with each move. The Pusher can collect FuelPack objects, which are placed on the map by the MapGrid. When the Pusher moves onto a tile containing a FuelPack, it collects the pack, restoring its fuel and removing the pack from the map. The MineMaze class oversees the overall game flow and delegates movement and collection actions to the Pusher. MineMazeCreator creates FuelPack objects and places them on the map.

GRASP Principles Applied

Information Expert: Pusher manages its own fuel and collection logic.

Creator: MapGrid creates and manages FuelPack objects.

Alternative Design

Currently, low coupling and polymorphism are not fully realized because FuelPack and Booster do not implement a shared interface or abstract class such as CollectableResource. As a result, Pusher must interact with each collectable type directly, leading to duplicated logic and tighter coupling. To address this, a CollectableResource abstract class or Collectable interface should be introduced and implemented by all collectable items. This would allow Pusher to interact with all collectables in a generic, polymorphic way, unifying collection logic and reducing dependencies on specific classes. Additionally, introducing a ResourceManager to handle the spawning and tracking of collectables would further decouple resource management from the map and actors, improving maintainability and extensibility.

Feature 3: Boosters

Relationships and Implementation

The booster feature involves the Pusher, which can collect a Booster from the map, also managed by the MapGrid. Once collected, the Booster temporarily enables the Pusher to push Boulder objects, which would otherwise be immovable. The activation and deactivation of the booster are managed within the Pusher, and the MineMaze class coordinates the overall game logic and delegates collection and pushing actions. The Booster is managed as an Actor, and the lack of a shared interface or superclass for collectables means Pusher must handle each type separately.

GRASP Principles Applied

Information Expert: The Pusher class is responsible for managing its own booster state and activation, as it has the necessary information about its current state and actions.

Controller: MineMaze acts as the controller, delegating collection and activation actions to Pusher.

High Cohesion: Each class has a focused responsibility (e.g., Booster represents the power-up, Pusher manages collection and use).

Alternative Design

Currently, the design lacks low coupling and polymorphism for collectable resources. To improve, a common interface or abstract class (e.g., CollectableResource) should be introduced and implemented by all collectable items, such as Booster and FuelPack. This would allow Pusher to interact with collectables in a generic way, reducing dependencies and unifying

collection logic. Additionally, introducing a Pushable interface for objects like Boulder would make it easier to add new pushable types in the future. These changes would enhance extensibility, maintainability, and adherence to GRASP principles.

Part 4: GRASP Principles Outside the New Features

Controller: MineMaze is the central controller, handling UI events, the game loop, and delegating responsibilities to other classes.

Creator: MineMazeCreator is a pure fabrication that centralizes the creation of all actors and pickups, reducing coupling and following the Creator and Indirection principles.

Information Expert: Each class is responsible for its own data and behavior (e.g., MapGrid for cell contents, BoardRenderer for drawing, HudRenderer for HUD).

Low Coupling: Rendering and creation are delegated out of MineMaze, reducing its responsibilities and coupling. Each actor class is focused and interacts with others through well-defined interfaces.

High Cohesion: Each class has a single, clear responsibility (e.g., BoardRenderer only draws the board, HudRenderer only draws the HUD, etc.).

Indirection: MineMazeCreator acts as an indirection layer for actor creation, isolating construction logic.

Part 5: Overall Potential Improvements

Introduce Polymorphism for Collectables: Implement a CollectableResource interface or abstract class for Fuel, Booster, and any future pickups. This would allow Pusher to interact with all collectables generically, reducing duplicated logic and coupling.

Protected Variations: Use interfaces like Destructible for obstacles and Pushable for objects that can be pushed, so new types can be added without modifying existing logic.

Pure Fabrication for Managers: Introduce managers (e.g., BombManager, ResourceManager, ObsticalManager) to handle complex behaviors, further reducing the responsibilities of actors like Bomber and Pusher.

Event-Driven or Observer Pattern: For actions like bomb explosions, resource collection, or win/loss conditions, consider using event-driven patterns to decouple classes and improve extensibility.

Reduce God Class Risk: Continue to delegate responsibilities from MineMaze to specialized classes as the project grows.

Figure 2: Design Model of MineMaze

