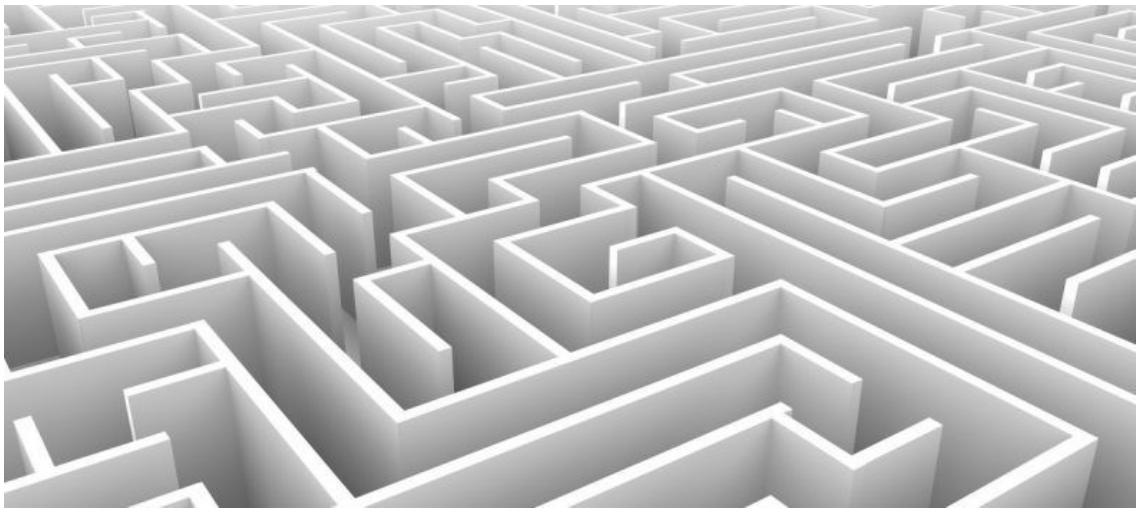
Maze Survivor

CS221 Programming 2 Assignment Report



Team Members

Khaled Barie	21
Khaled Abdelfattah	22
Sajed Hassan	28
Hesham Medhat	70

Overview

The game is designed as a top-down survival maze game where the player has to survive the maze using his low resources and facing many monsters, blockades and traps.

Development was in JavaFX. We tried to adhere to the principles and common practices of game development regarding the creation and update of game objects, assets management etc..

Design

The design of the game followed the MVC architecture separating most of the game logic from the GUI.

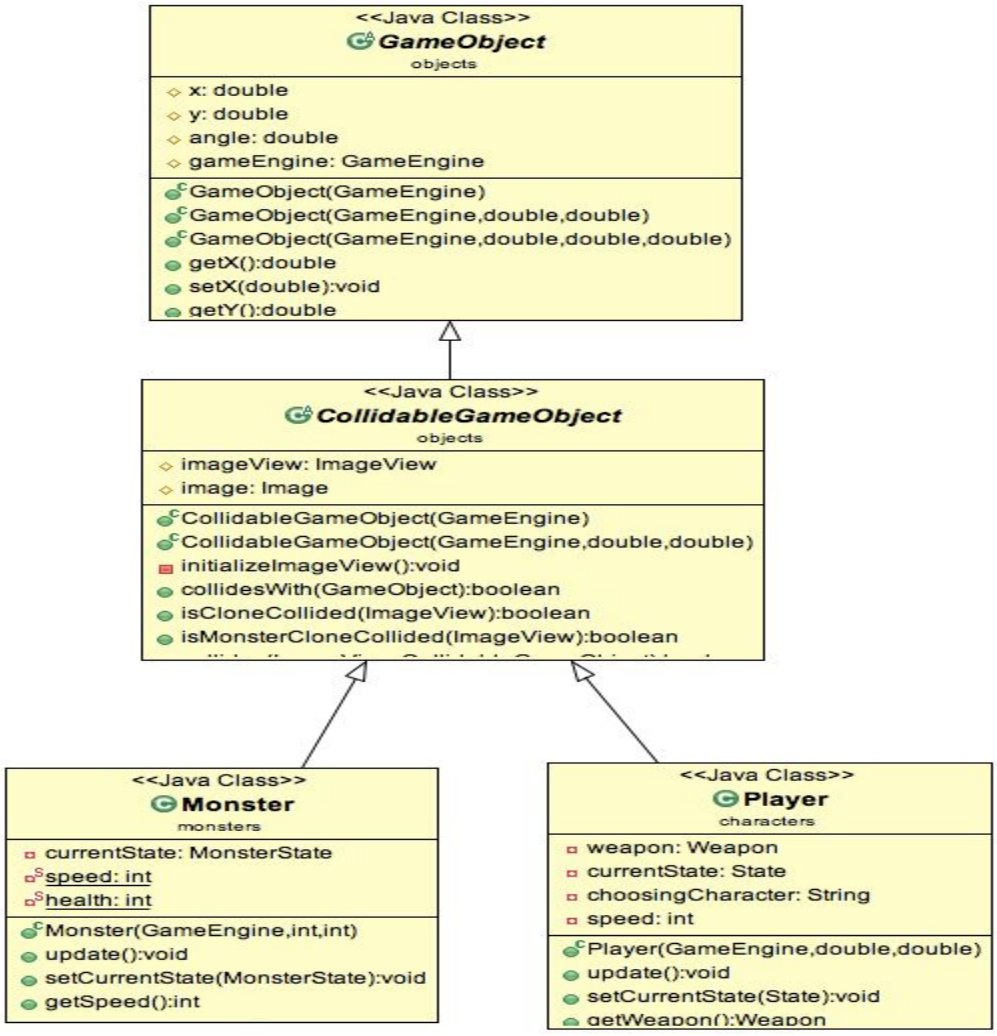
The game loop iterates on all *GameObjects* in the game and runs their update method. The update method is responsible for the regular behaviour of a GameObject. Special behaviours/transitions of GameObjects mainly made use of the state design pattern to control their behaviour.

UML Diagrams

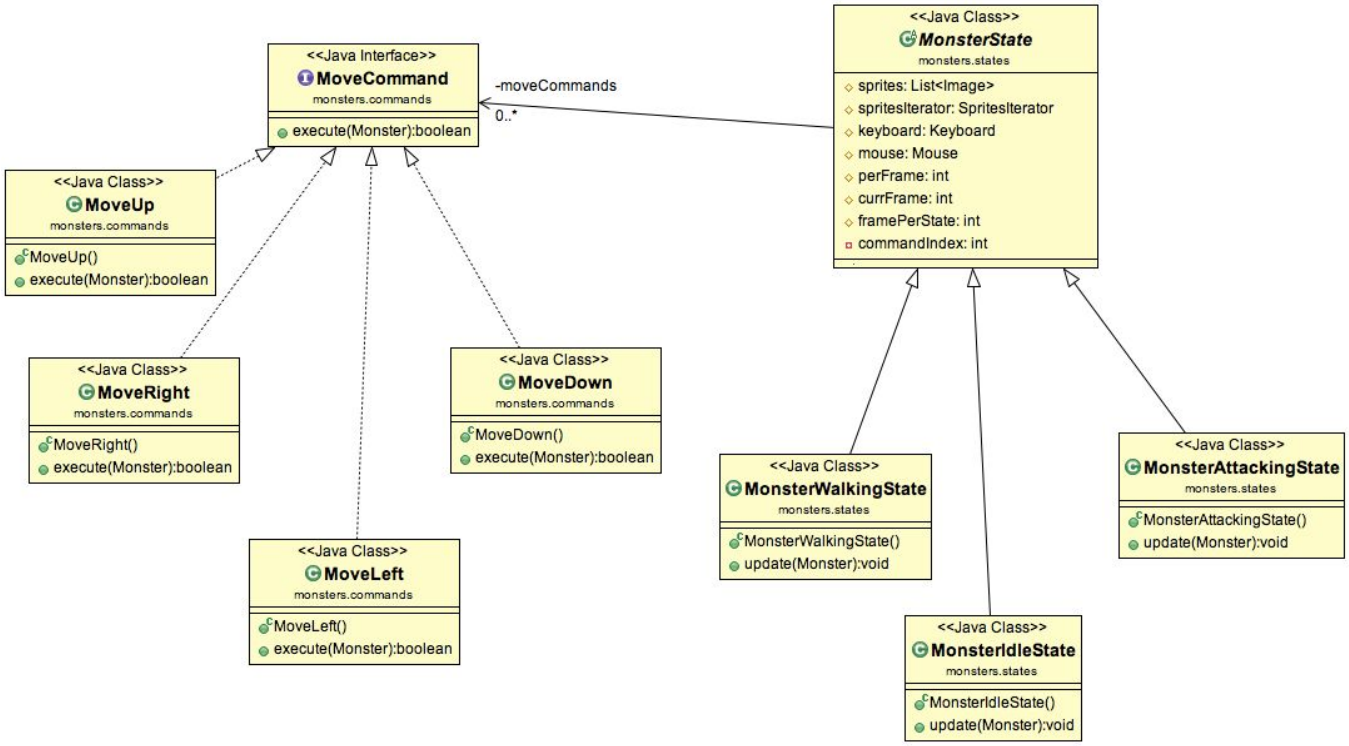
- Class Diagrams
    - Class diagram for Game objects

      The GameObject abstract class is the base class of all game related objects. It contains the update method that is to be run every frame of the game to set and control a game entity's behaviour.

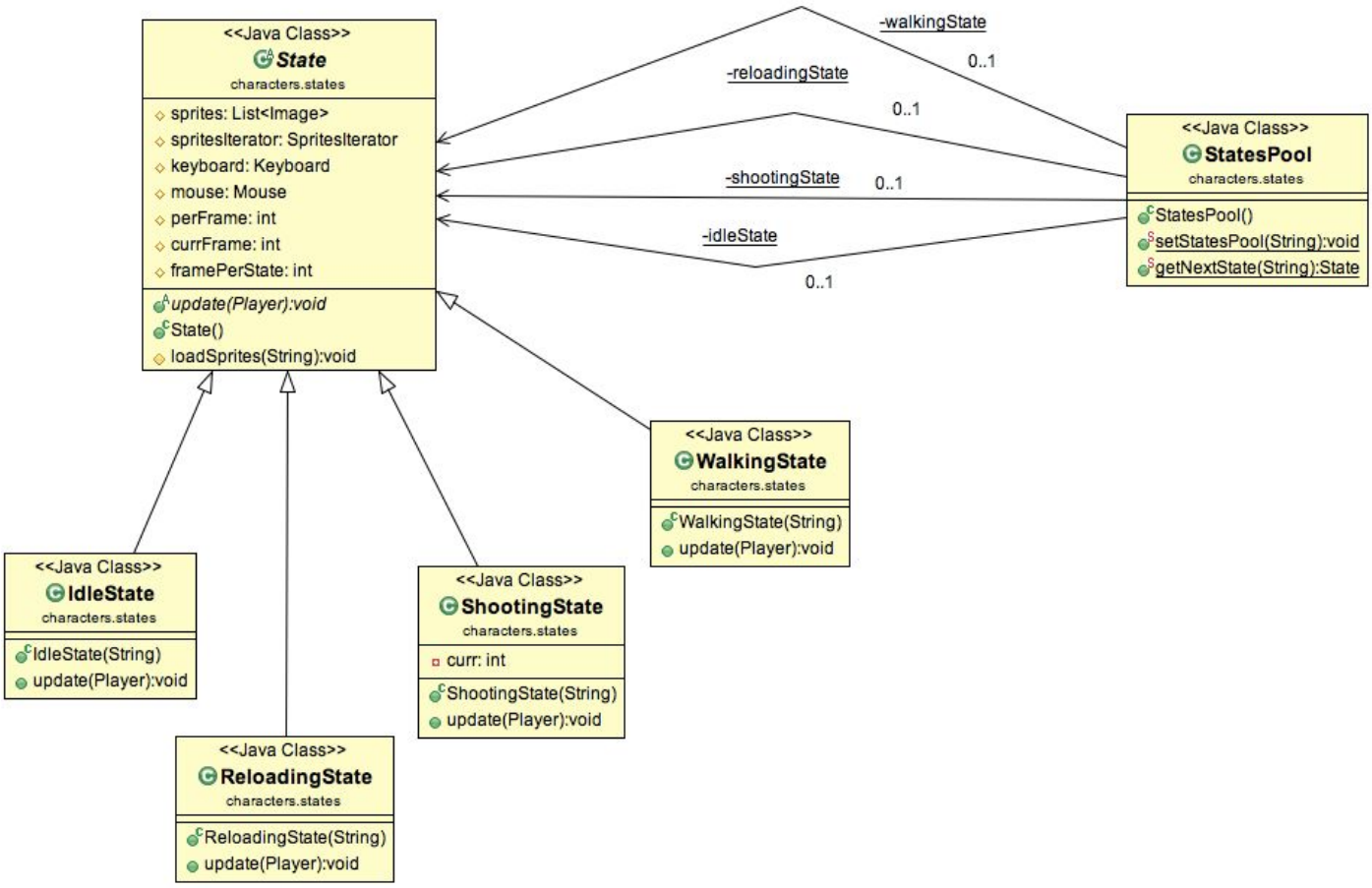
      The abstract class encapsulates a wide concept. Some special concretions of the class include objects to handle the game camera, HUD etc..


<<Java Class>>
GameObject
objects
x: double
y: double
angle: double
gameEngine: GameEngine
GameObject(GameEngine)
GameObject(GameEngine,double,double)
GameObject(GameEngine,double,double,double)
getX():double
setX(double):void
getY():double
<<Java Class>>
CollidableGameObject
objects
imageView: ImageView
image: Image
CollidableGameObject(GameEngine)
CollidableGameObject(GameEngine,double,double)
initializeImageView():void
collidesWith(GameObject):boolean
isCloneCollided(ImageView):boolean
isMonsterCloneCollided(ImageView):boolean
<<Java Class>>
Monster
monsters
currentState: MonsterState
speed: int
health: int
Monster(GameEngine,int,int)
update():void
setCurrentState(MonsterState):void
getSpeed():int
<<Java Class>>
Player
characters
weapon: Weapon
currentState: State
choosingCharacter: String
speed: int
Player(GameEngine,double,double)
update():void
setCurrentState(State):void

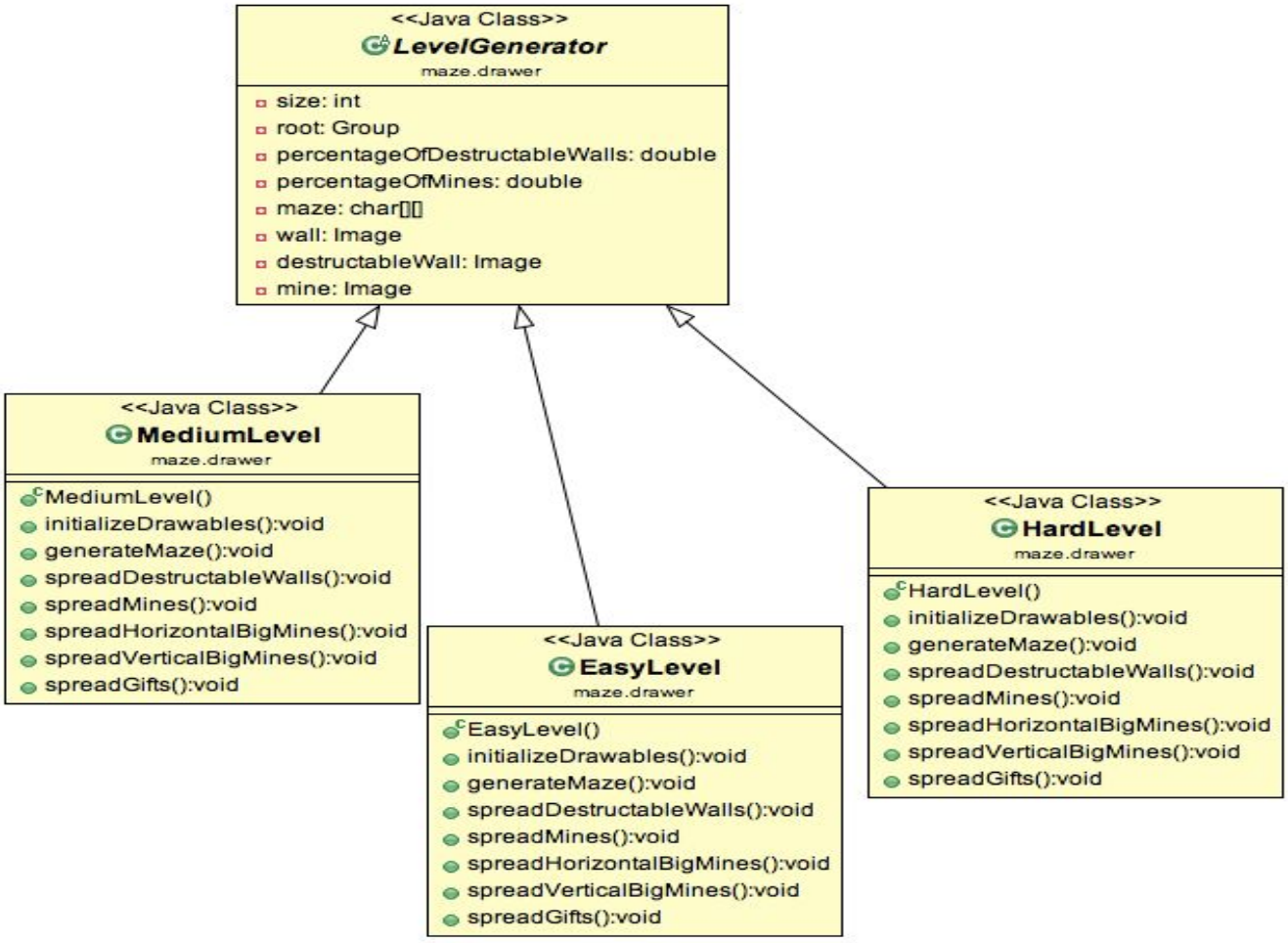
- Command and state pattern for monsters



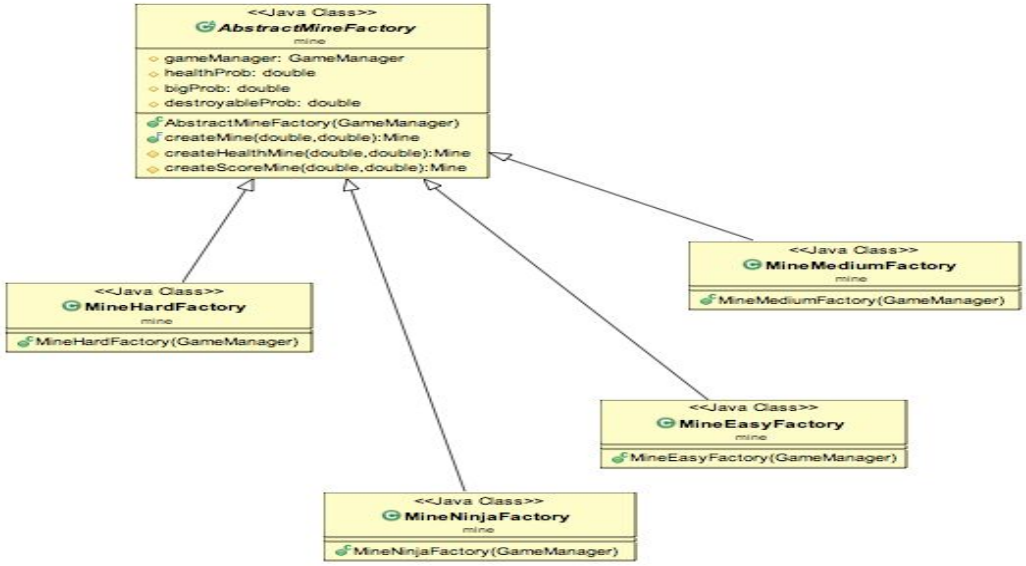
- State and Pool pattern for player



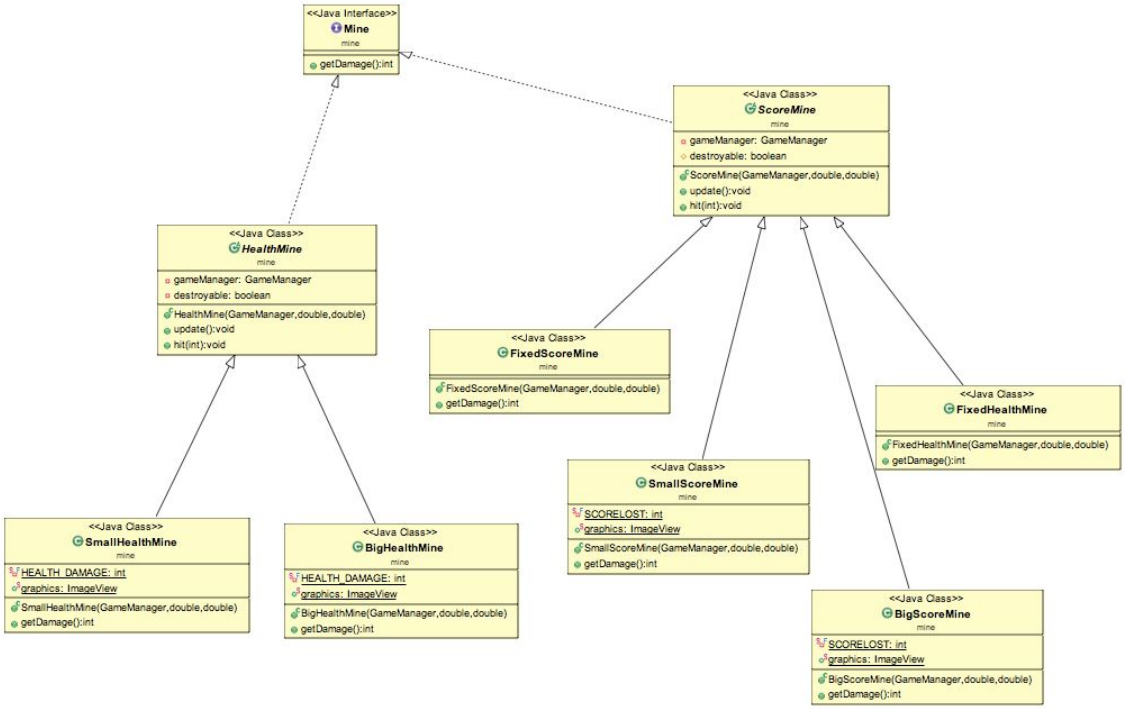
- Template method pattern for generating each level's maze



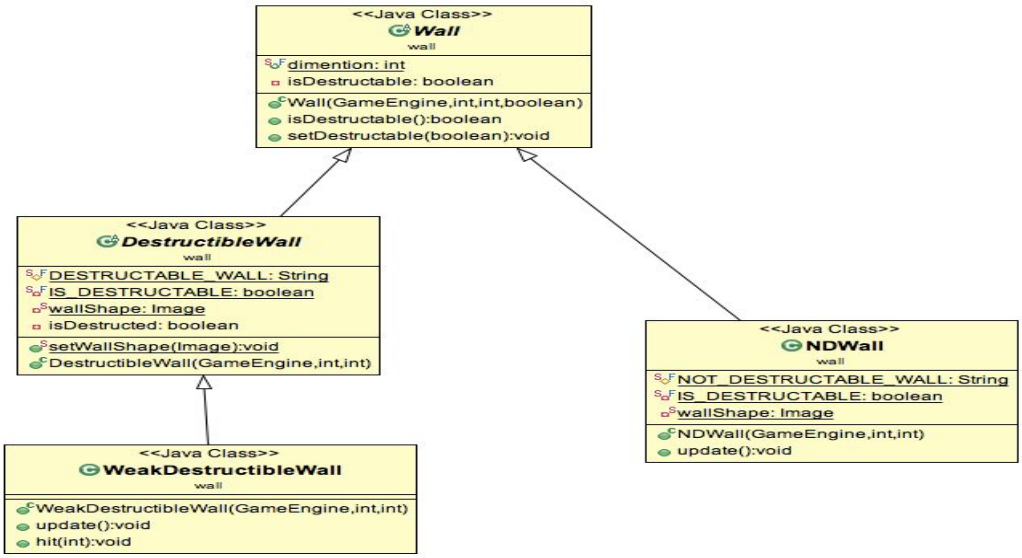
- Abstract factory and factory pattern for mines strategy



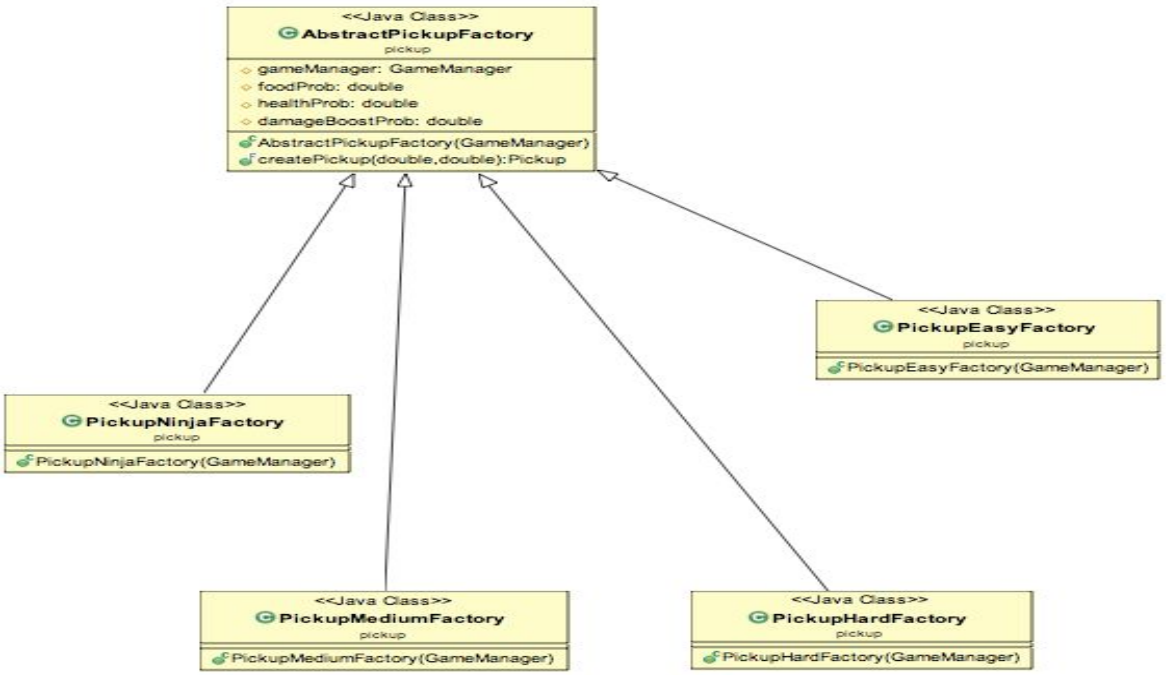
- Strategy of mines

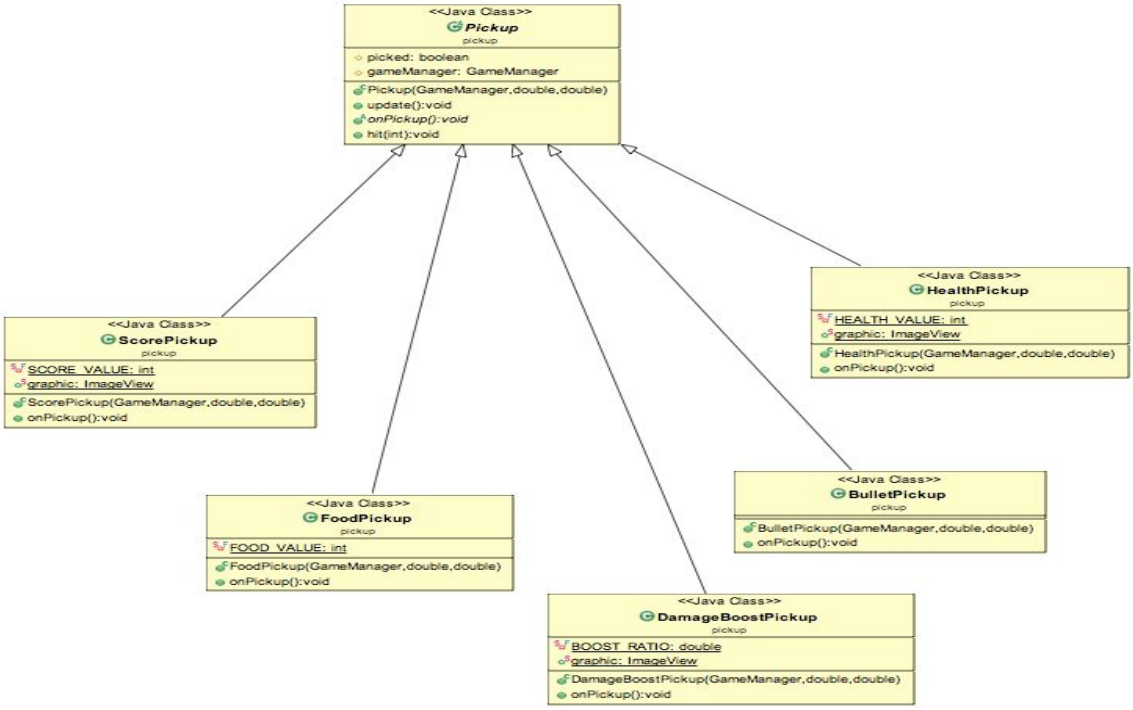


- Class diagram for walls

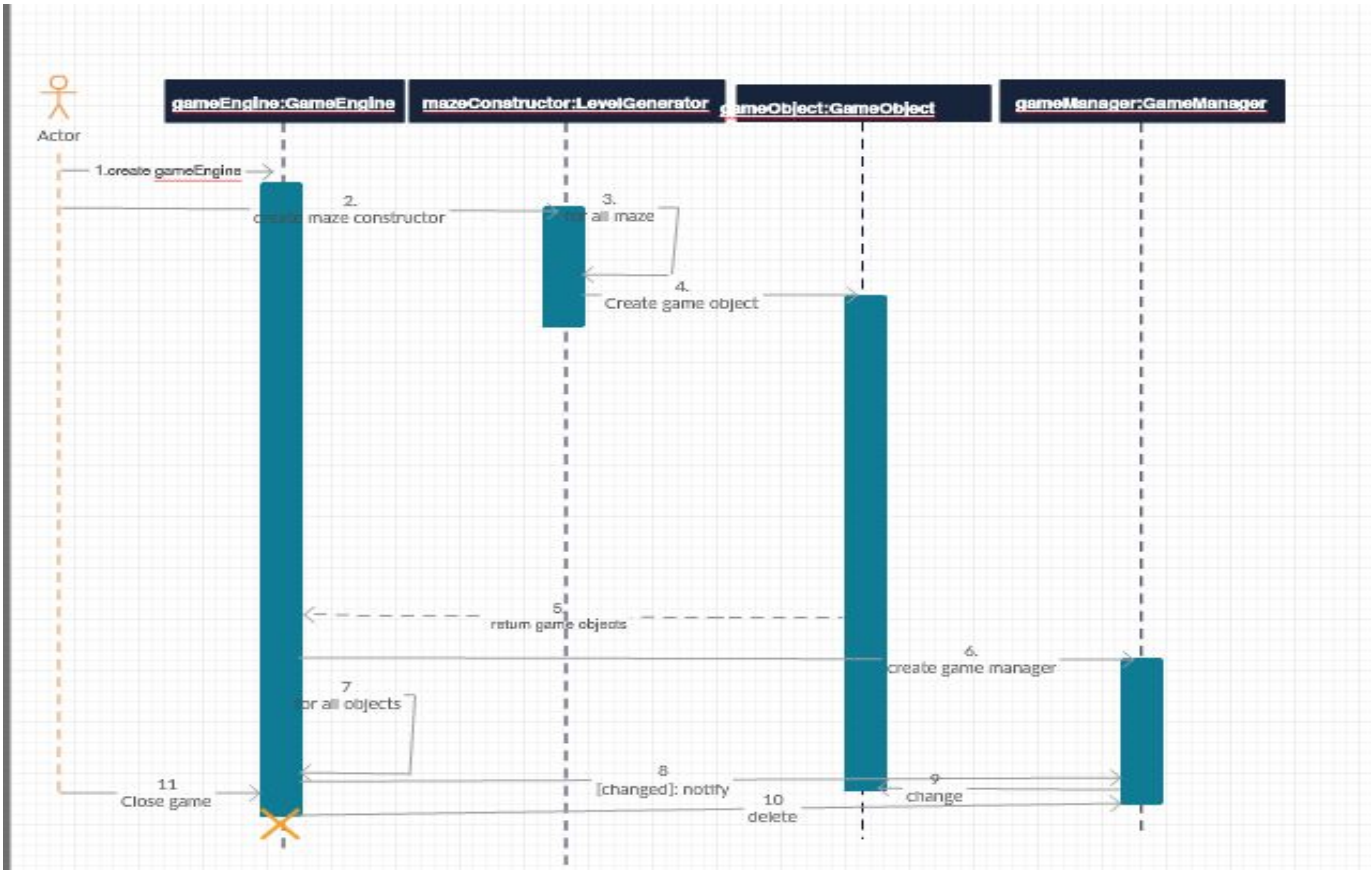


- Pickups factories and strategy




<<Java Class>>
Pickup
pickup
picked: boolean
gameManager: GameManager
Pickup(GameManager,double,double)
update():void
onPickup():void
hit(int):void
<<Java Class>>
ScorePickup
pickup
SCORE_VALUE: int
graphic: ImageView
ScorePickup(GameManager,double,double)
onPickup():void
<<Java Class>>
HealthPickup
pickup
HEALTH_VALUE: int
graphic: ImageView
HealthPickup(GameManager,double,double)
onPickup():void
<<Java Class>>
FoodPickup
pickup
FOOD_VALUE: int
FoodPickup(GameManager,double,double)
onPickup():void
<<Java Class>>
BulletPickup
pickup
BulletPickup(GameManager,double,double)
onPickup():void
<<Java Class>>
DamageBoostPickup
pickup
BOOST_RATIO: double
graphic: ImageView
DamageBoostPickup(GameManager,double,double)
onPickup():void

- **Sequence Diagram**



Design Patterns

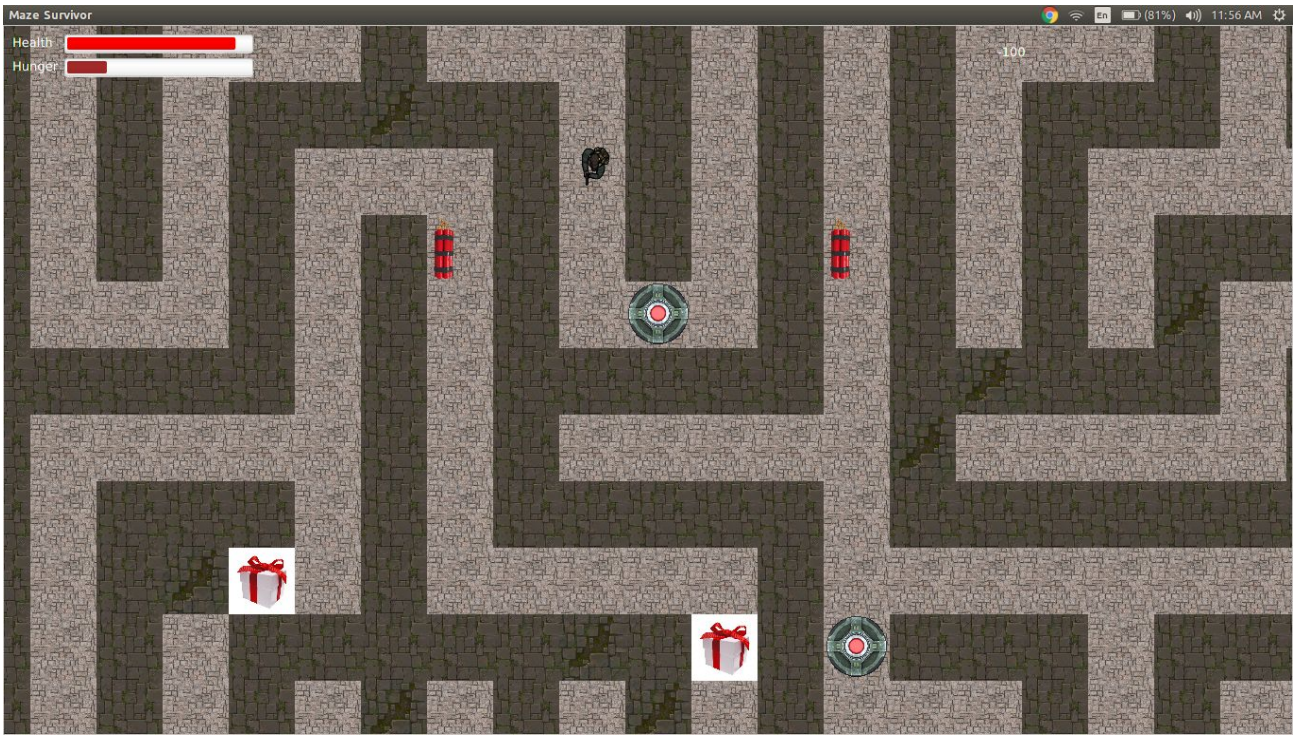
- **Singleton:** Used in the *GameEngine* which is responsible for running the game and managing the game loop.
- **Builder:** Used in building the player object and preparing his weapon, ammo and necessary initializations.
- **Observer:** If a *GameObject* is to be tracked throughout the game, it subscribes to the game engine. As the *GameEngine* runs the game, it calls an “update()” method on all the *GameObjects*.
- **Factory:** Used in handling creation of different kinds of mines, walls and pickups.
- **Abstract Factory:** Used to handle different families of the mines, walls and pickups modules to handle their parameters (damage/bonus/health) according to the difficulty.

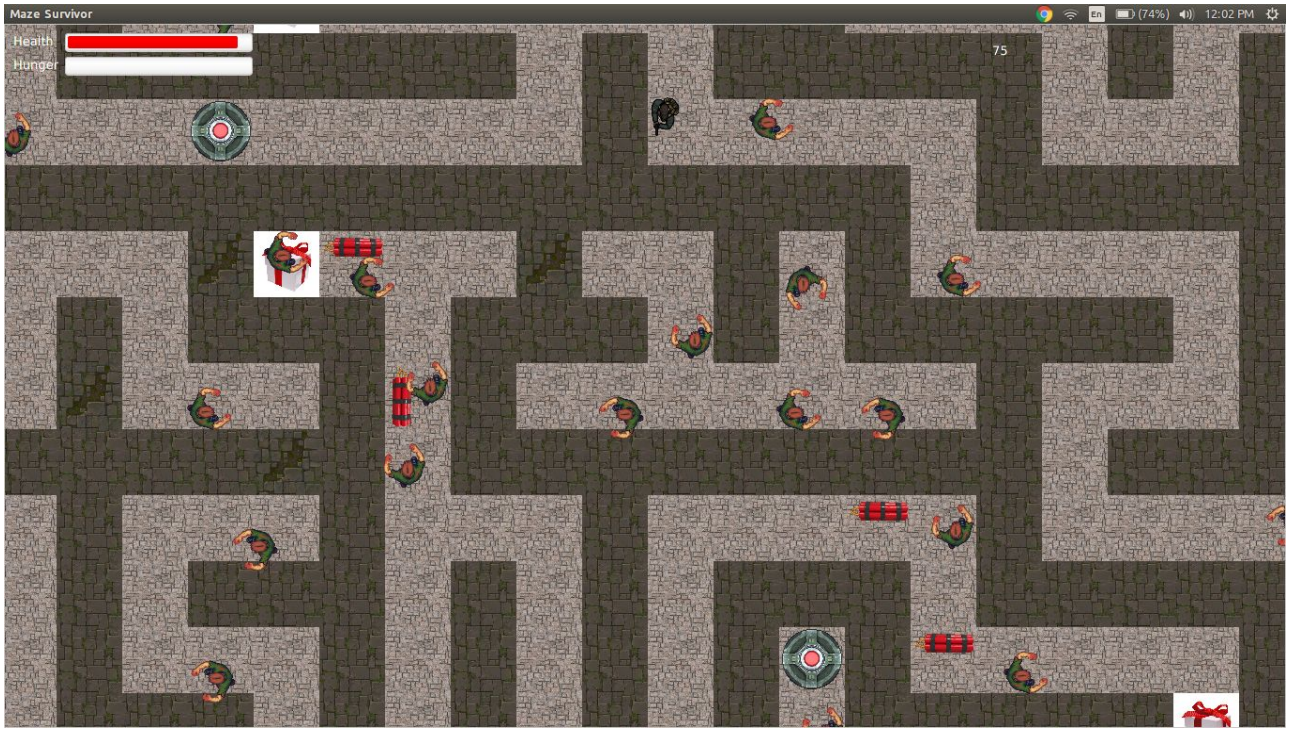
- **Pool:** Used twice for *Bull*ets and *S*tates for the player.
- **Flyweight:** Used in handling expensive resources and providing an access for objects needing them such as assets like sprites and sound clips.
- **Dynamic Linkage:** used to allow the user to dynamically set the sprites of the pickups.
- **Proxy:** Used a virtual proxy for handling bullets so as to reduce the overhead of constantly creating them.
- **Command:** The monster uses a pathfinding algorithm then uses command objects to decide on the direction to move at each instance.
- **State:** Used to handle the behaviour and transitions of both the player and the character objects. States pool save the resources by initially creating some instances of the state objects that can be reused when needed.
- **Template method:** for handling level difficulty using hooks for determining the number of mines, monsters, destructible walls, etc..
- **Iterator:** Used in iterating on the bullets inside the bullet pool. Also used in iterating over sprites for sprite animations.
- **Facade:** Used in the implementation of the GameManager component.

User Guide

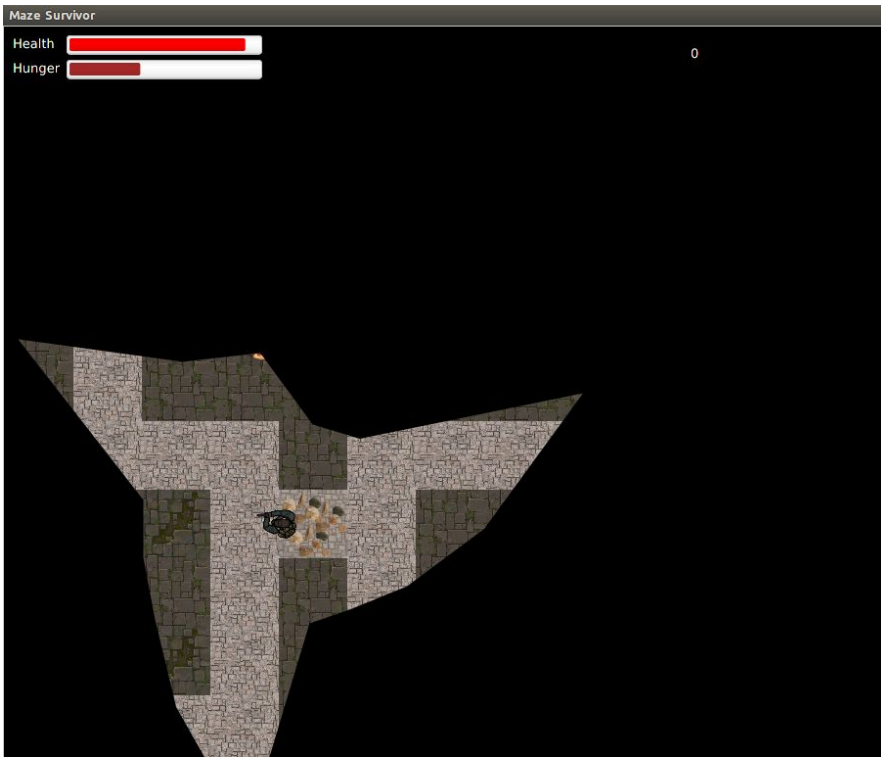
- On launching the jar, the user will be prompted to the main menu.
- The user can choose to play a game and he will be prompted to select options for the game:
    - The difficulty of the game: Easy, Normal, Hard and Ninja.
    - The weapon to start the game with.
- The game then starts with the player at the beginning of the maze. The player must navigate to the end of the maze.
- Game controls:
    - Movement by WASD keys
    - Firing the weapon by Space/Primary mouse button
- The user should avoid/destroy monsters and mines they encounter.

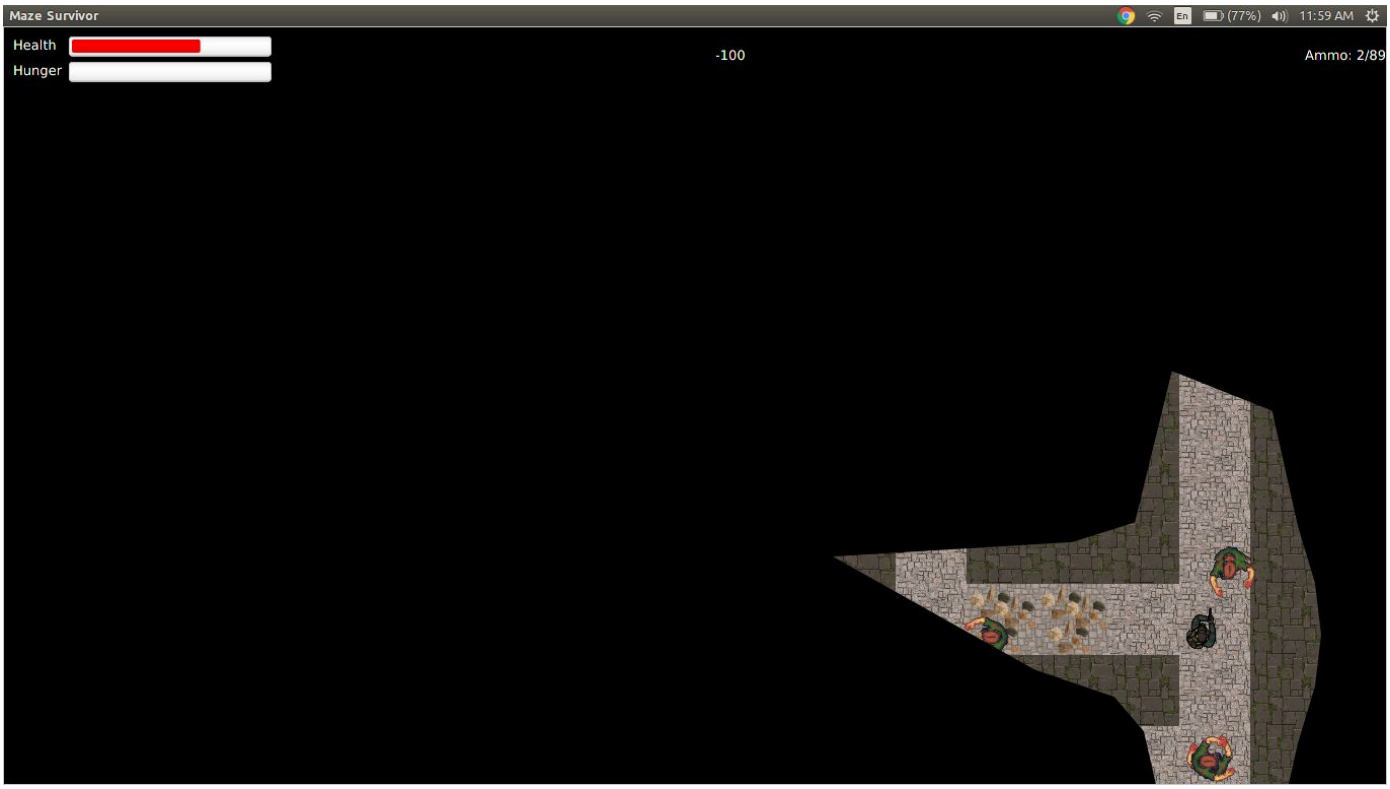
Snapshots


Maze Survivor
Health
Hunger
100


Maze Survivor
Health
Hunger
75

A preview of the game with the shadows effect





r