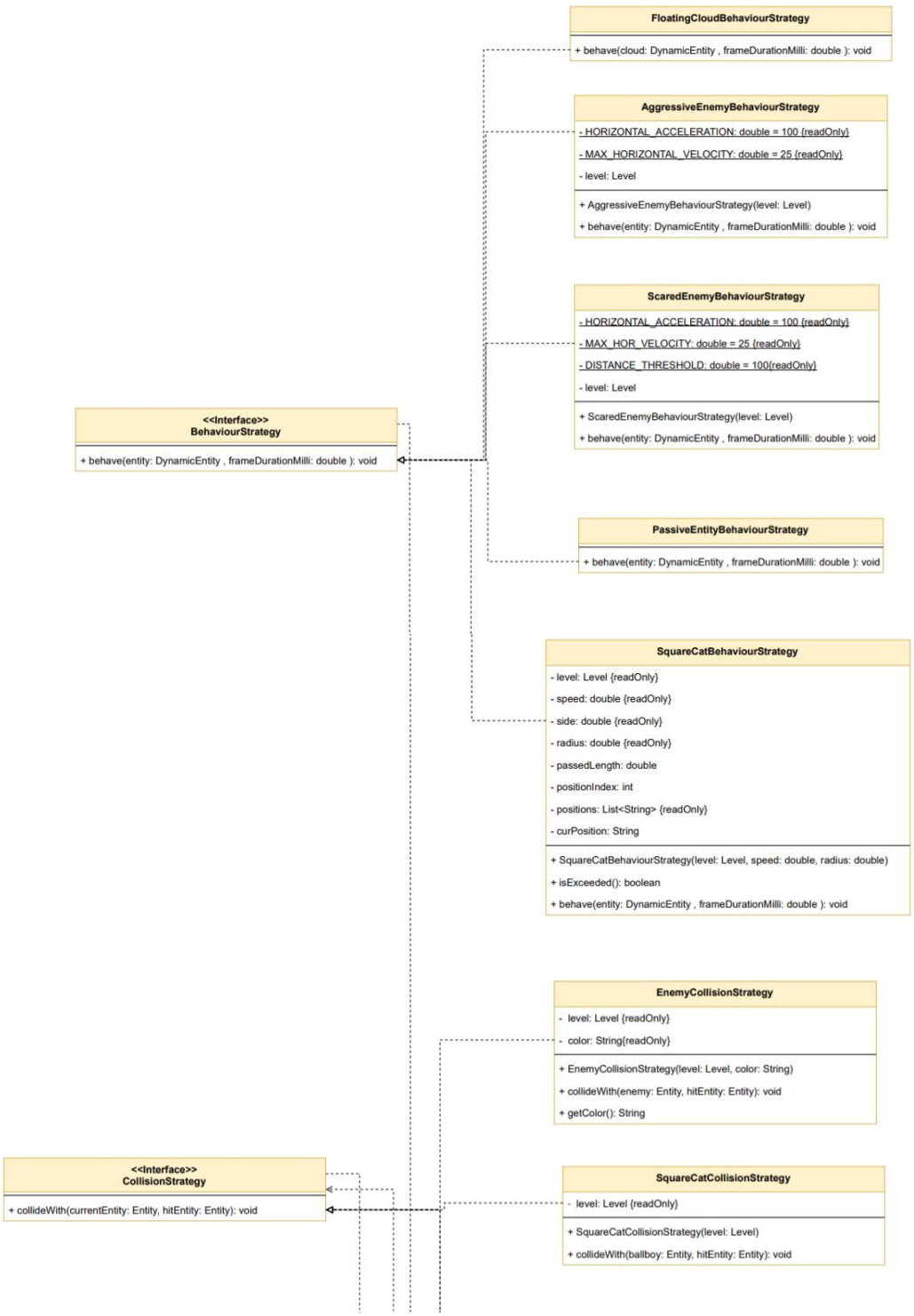
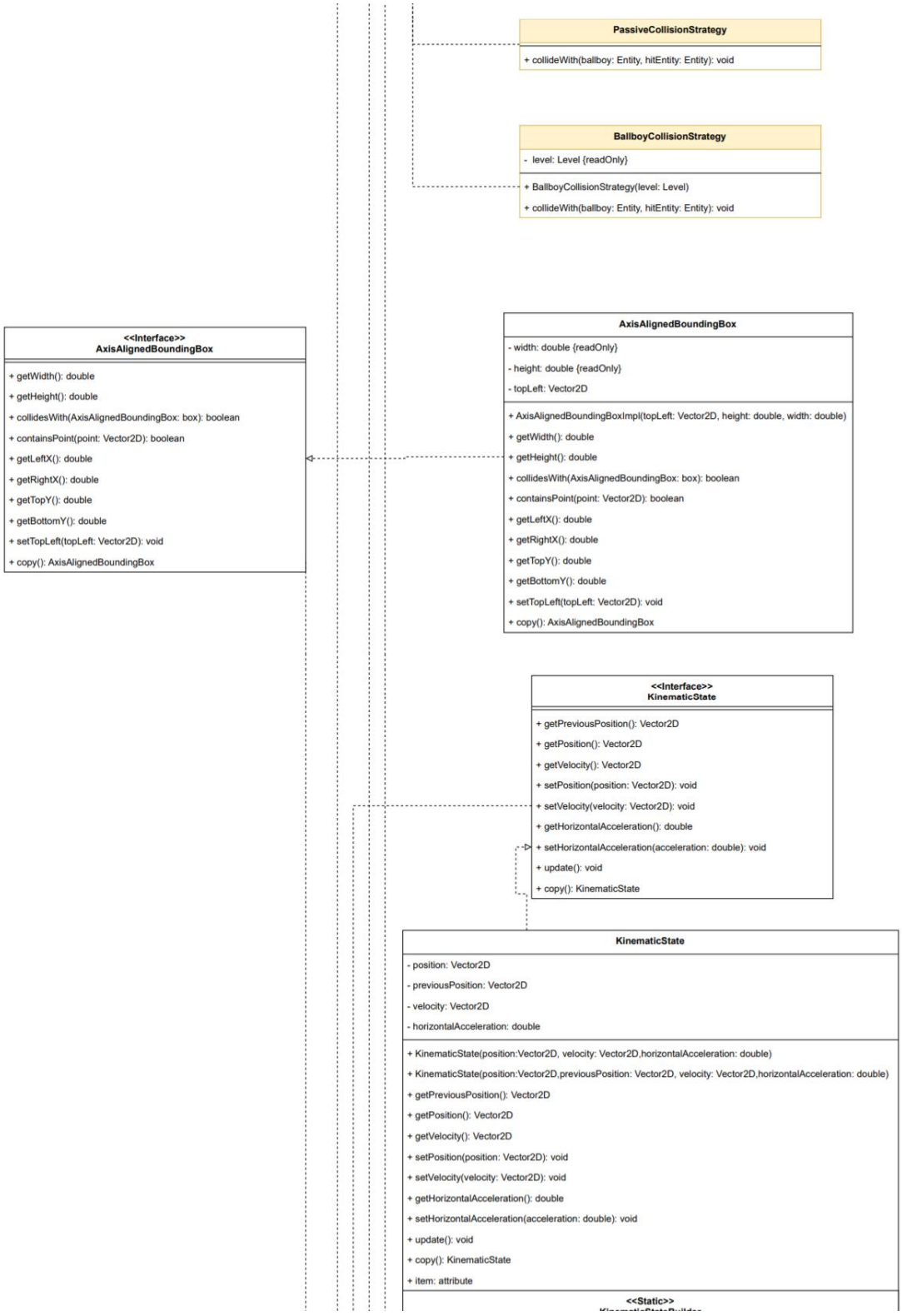
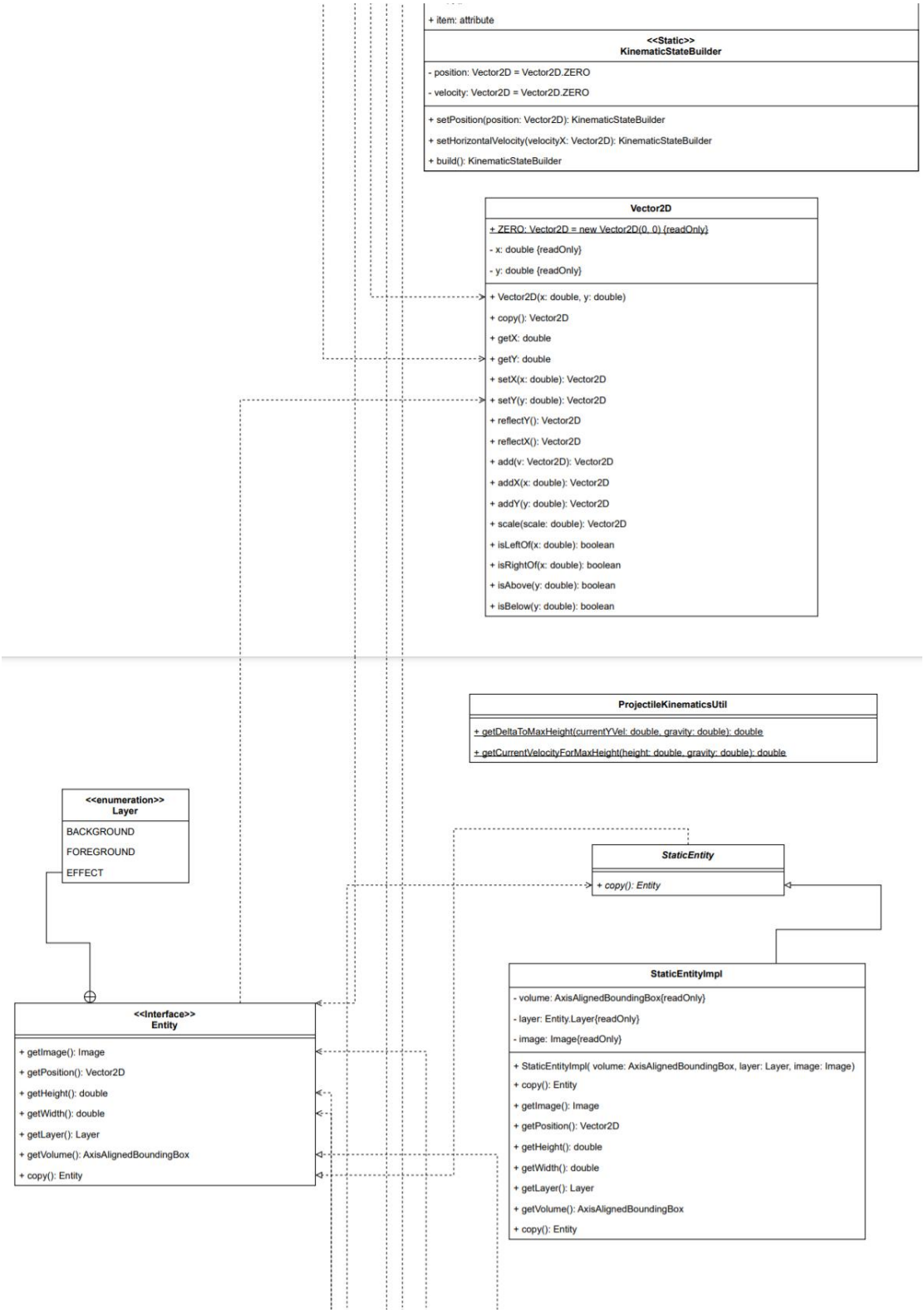
Report

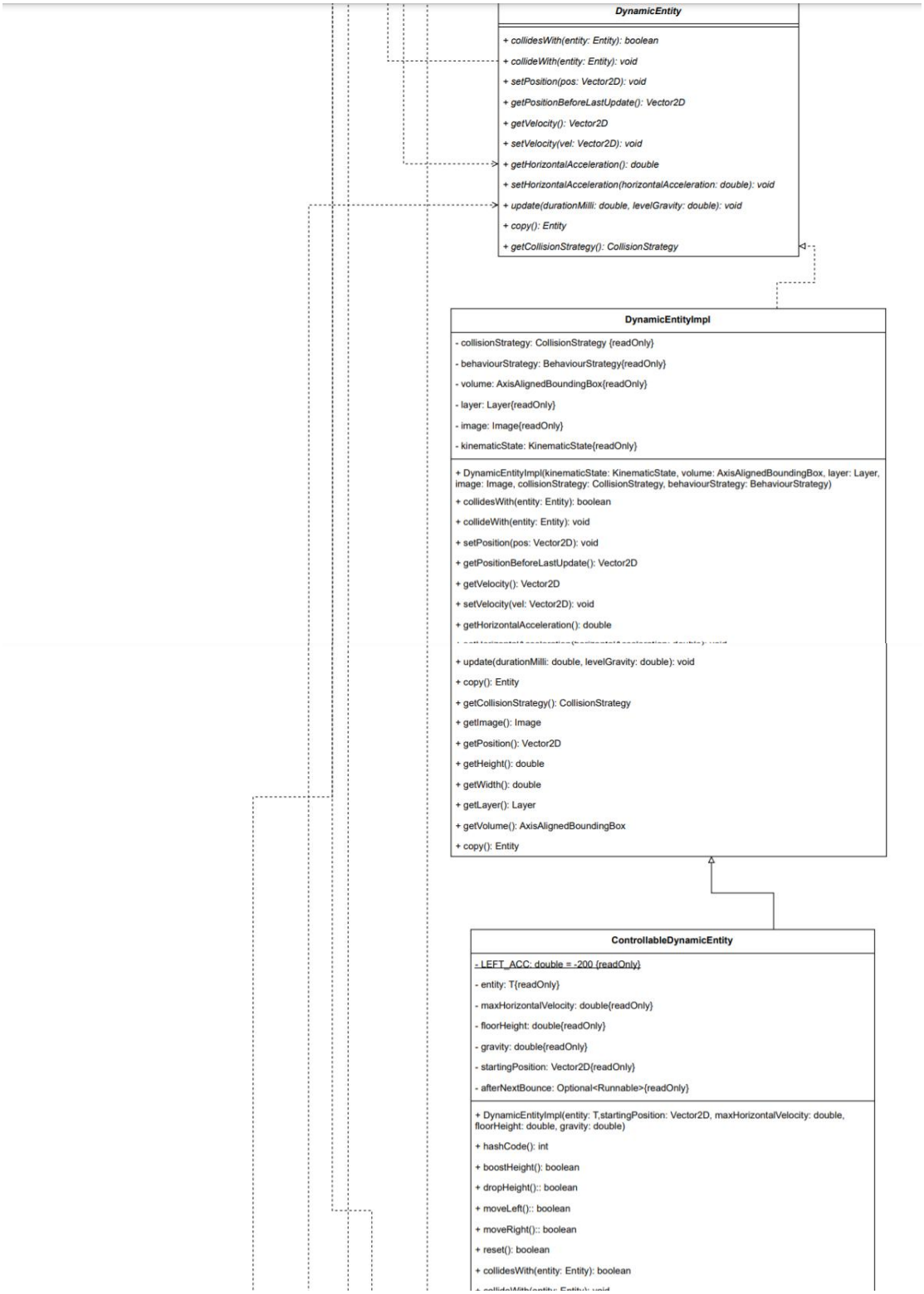
SID: 500025673

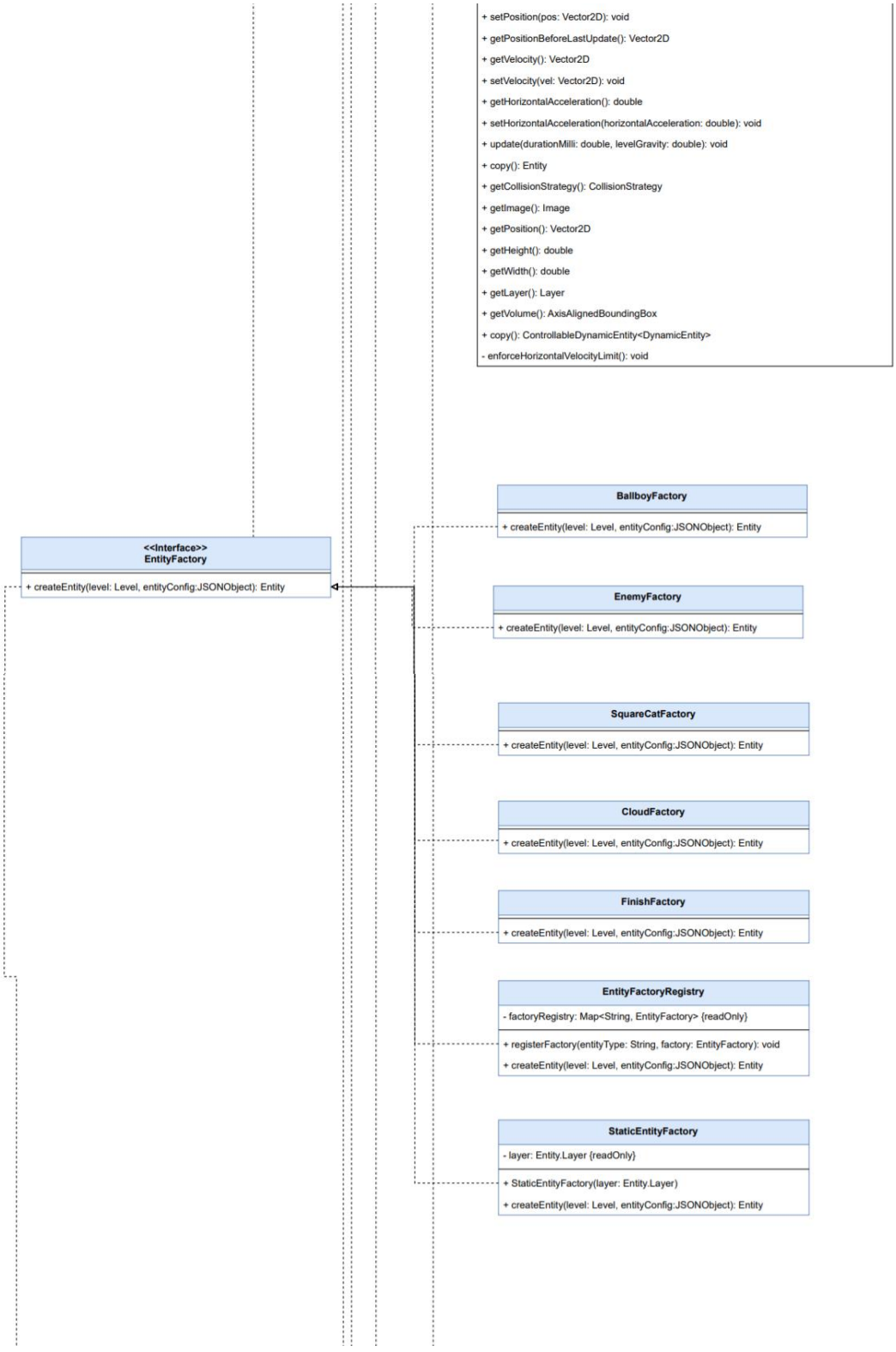
UML:

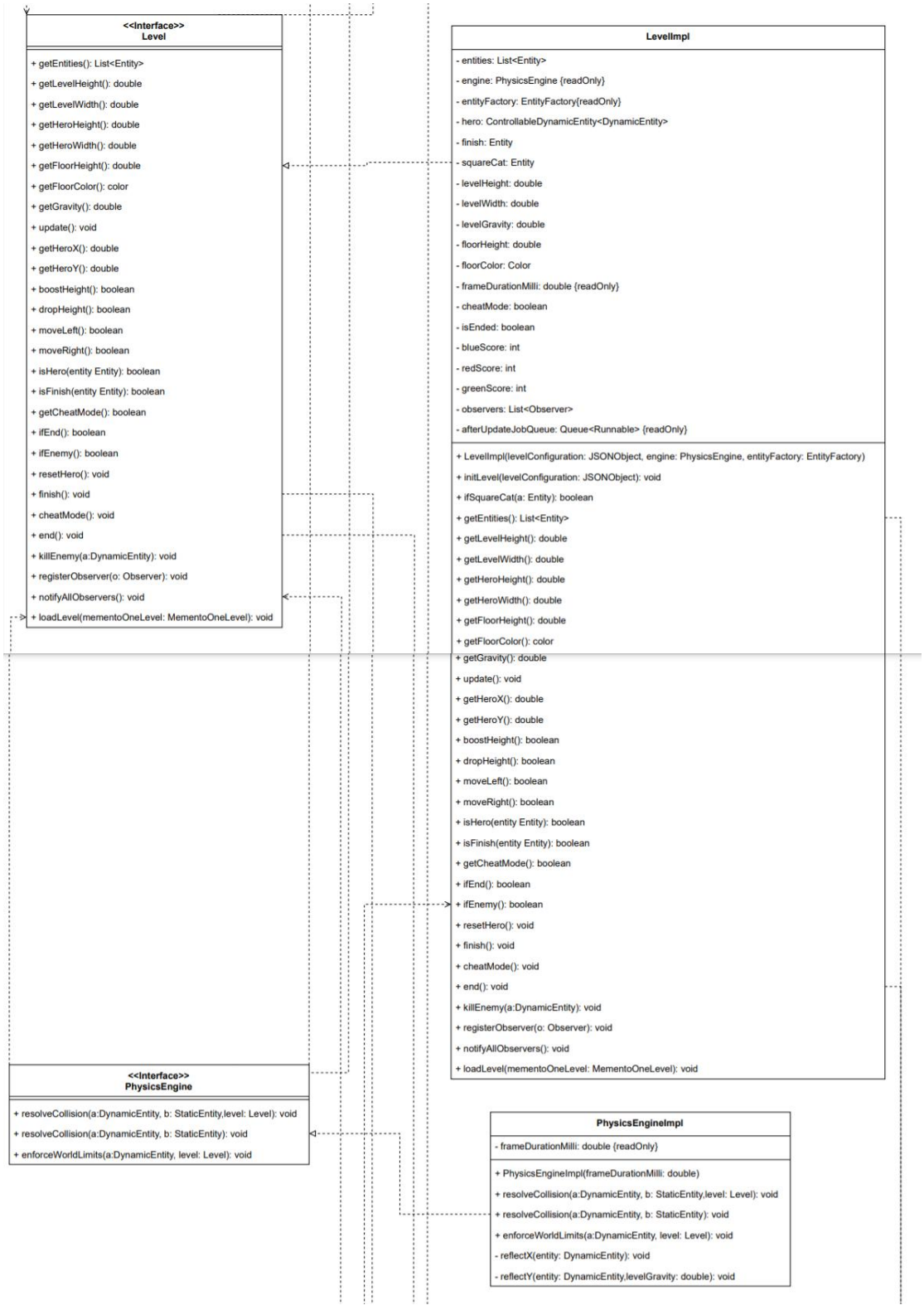


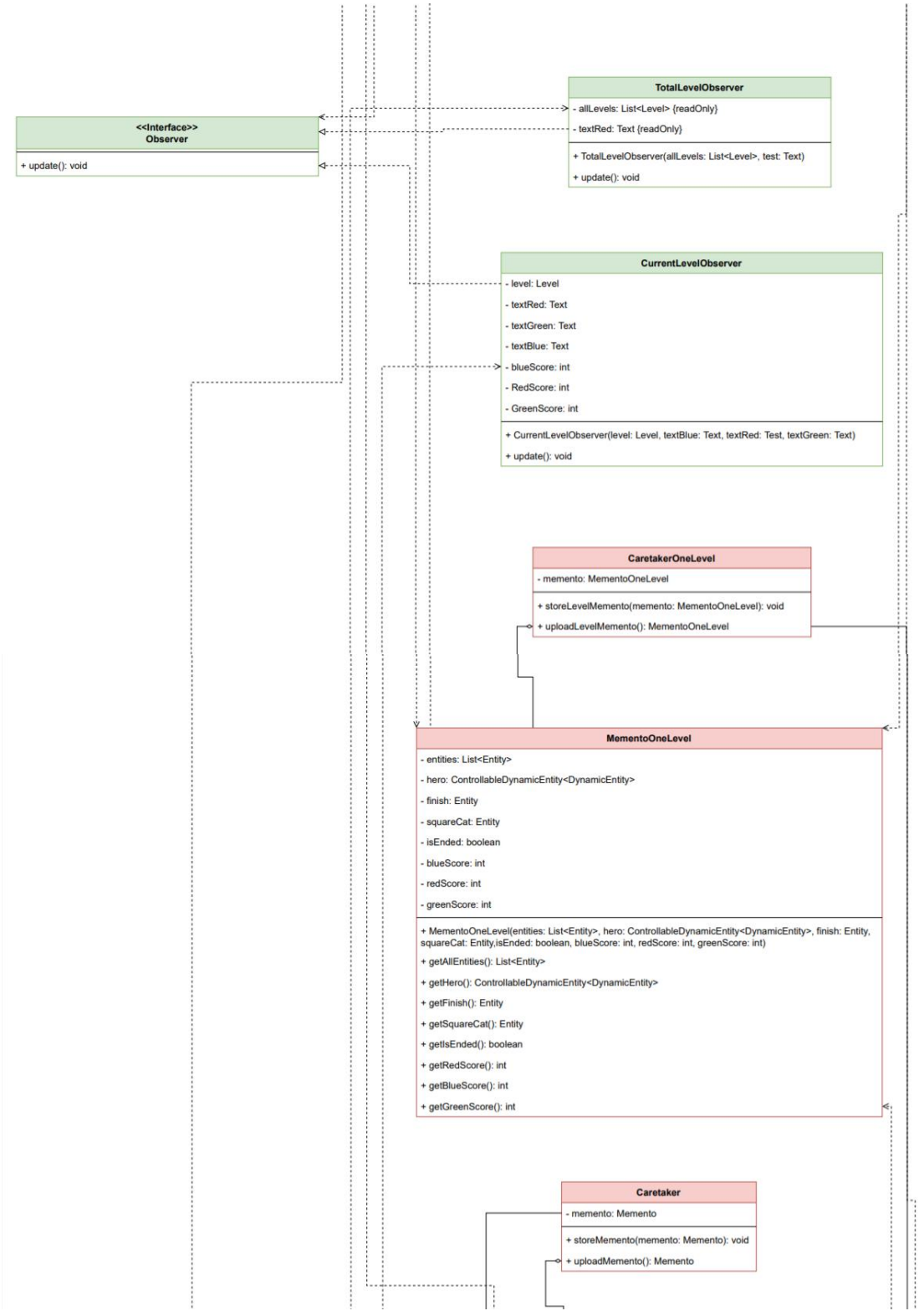

PassiveCollisionStrategy
+ collideWith(ballboy: Entity, hitEntity: Entity): void
BallboyCollisionStrategy
- level: Level {readOnly}
+ BallboyCollisionStrategy(level: Level)
+ collideWith(ballboy: Entity, hitEntity: Entity): void
<<Interface>>
AxisAlignedBoundingBox
+ getWidth(): double
+ getHeight(): double
+ collidesWith(AxisAlignedBoundingBox: box): boolean
+ containsPoint(point: Vector2D): boolean
+ getLeftX(): double
+ getRightX(): double
+ getTopY(): double
+ getBottomY(): double
+ setTopLeft(topLeft: Vector2D): void
+ copy(): AxisAlignedBoundingBox
AxisAlignedBoundingBox
- width: double {readOnly}
- height: double {readOnly}
- topLeft: Vector2D
+ AxisAlignedBoundingBoxImpl(topLeft: Vector2D, height: double, width: double)
+ getWidth(): double
+ getHeight(): double
+ collidesWith(AxisAlignedBoundingBox: box): boolean
+ containsPoint(point: Vector2D): boolean
+ getLeftX(): double
+ getRightX(): double
+ getTopY(): double
+ getBottomY(): double
+ setTopLeft(topLeft: Vector2D): void
+ copy(): AxisAlignedBoundingBox
<<Interface>>
KinematicState
+ getPreviousPosition(): Vector2D
+ getPosition(): Vector2D
+ getVelocity(): Vector2D
+ setPosition(position: Vector2D): void
+ setVelocity(velocity: Vector2D): void
+ getHorizontalAcceleration(): double
+ setHorizontalAcceleration(acceleration: double): void
+ update(): void
+ copy(): KinematicState
KinematicState
- position: Vector2D
- previousPosition: Vector2D
- velocity: Vector2D
- horizontalAcceleration: double
+ KinematicState(position:Vector2D, velocity: Vector2D,horizontalAcceleration: double)
+ KinematicState(position:Vector2D,previousPosition: Vector2D, velocity: Vector2D,horizontalAcceleration: double)
+ getPreviousPosition(): Vector2D
+ getPosition(): Vector2D
+ getVelocity(): Vector2D
+ setPosition(position: Vector2D): void
+ setVelocity(velocity: Vector2D): void
+ getHorizontalAcceleration(): double
+ setHorizontalAcceleration(acceleration: double): void
+ update(): void
+ copy(): KinematicState
+ item: attribute
<<Static>>

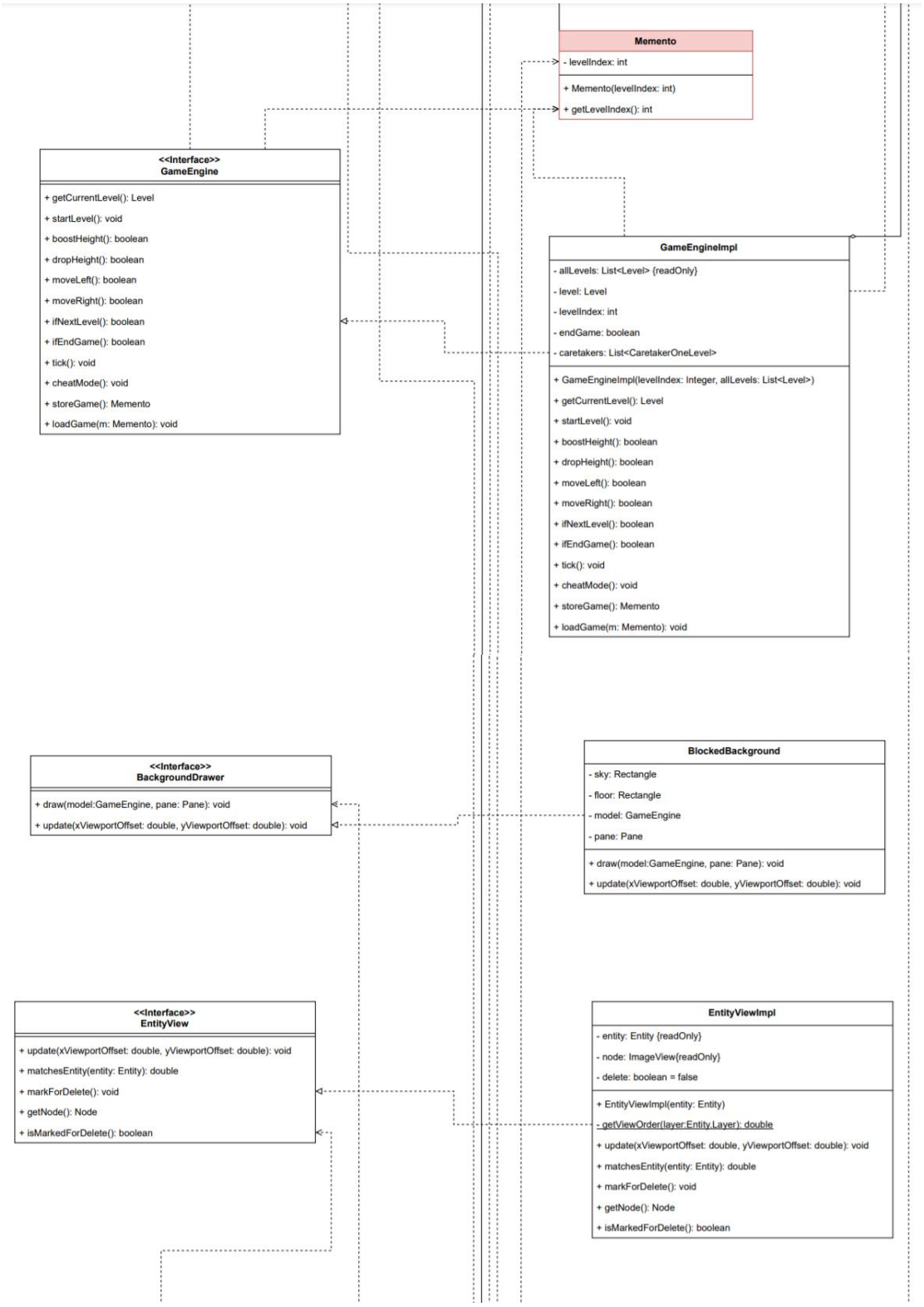

+ item: attribute
<<Static>>
KinematicStateBuilder
- position: Vector2D = Vector2D.ZERO
- velocity: Vector2D = Vector2D.ZERO
+ setPosition(position: Vector2D): KinematicStateBuilder
+ setHorizontalVelocity(velocityX: Vector2D): KinematicStateBuilder
+ build(): KinematicStateBuilder
Vector2D
+ ZERO: Vector2D = new Vector2D(0, 0) {readOnly}
- x: double {readOnly}
- y: double {readOnly}
+ Vector2D(x: double, y: double)
+ copy(): Vector2D
+ getX: double
+ getY: double
+ setX(x: double): Vector2D
+ setY(y: double): Vector2D
+ reflectY(): Vector2D
+ reflectX(): Vector2D
+ add(v: Vector2D): Vector2D
+ addX(x: double): Vector2D
+ addY(y: double): Vector2D
+ scale(scale: double): Vector2D
+ isLeftOf(x: double): boolean
+ isRightOf(x: double): boolean
+ isAbove(y: double): boolean
+ isBelow(y: double): boolean
ProjectileKinematicsUtil
+ getDeltaToMaxHeight(currentYVel: double, gravity: double): double
+ getCurrentVelocityForMaxHeight(height: double, gravity: double): double
<<enumeration>>
Layer
BACKGROUND
FOREGROUND
EFFECT
StaticEntity
+ copy(): Entity
<<Interface>>
Entity
+ getImage(): Image
+ getPosition(): Vector2D
+ getHeight(): double
+ getWidth(): double
+ getLayer(): Layer
+ getVolume(): AxisAlignedBoundingBox
+ copy(): Entity
StaticEntityImpl
- volume: AxisAlignedBoundingBox{readOnly}
- layer: Entity.Layer{readOnly}
- image: Image{readOnly}
+ StaticEntityImpl(volume: AxisAlignedBoundingBox, layer: Layer, image: Image)
+ copy(): Entity
+ getImage(): Image
+ getPosition(): Vector2D
+ getHeight(): double
+ getWidth(): double
+ getLayer(): Layer
+ getVolume(): AxisAlignedBoundingBox
+ copy(): Entity

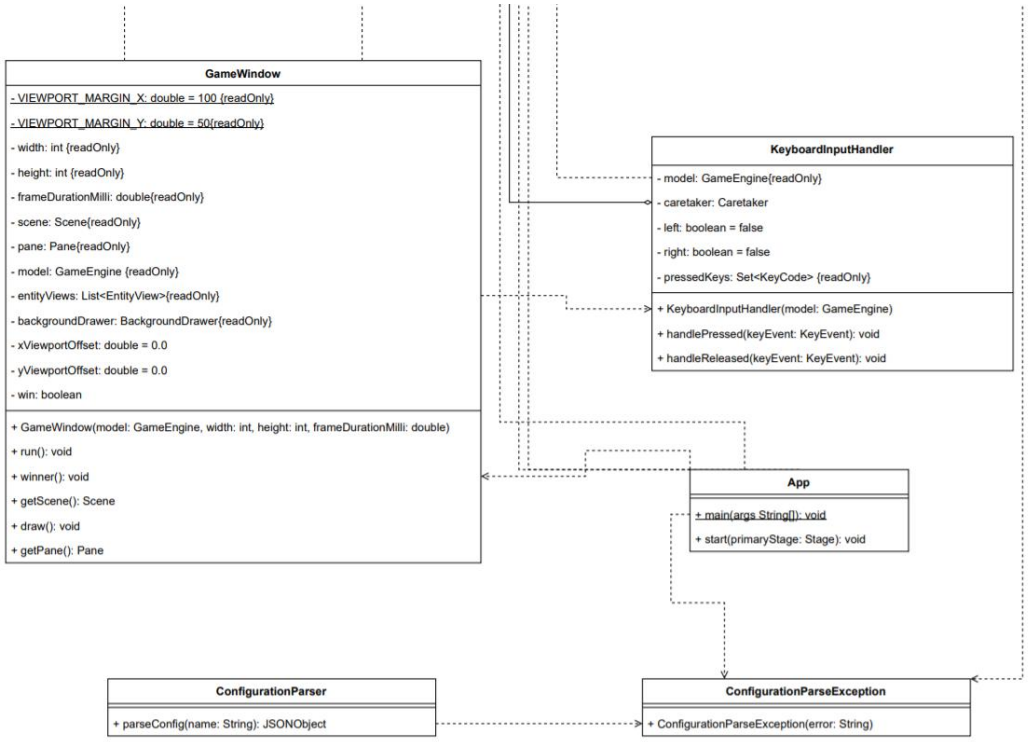

DynamicEntity
+ collidesWith(entity: Entity): boolean
+ collideWith(entity: Entity): void
+ setPosition(pos: Vector2D): void
+ getPositionBeforeLastUpdate(): Vector2D
+ getVelocity(): Vector2D
+ setVelocity(vel: Vector2D): void
+ getHorizontalAcceleration(): double
+ setHorizontalAcceleration(horizontalAcceleration: double): void
+ update(durationMilli: double, levelGravity: double): void
+ copy(): Entity
+ getCollisionStrategy(): CollisionStrategy
DynamicEntityImpl
- collisionStrategy: CollisionStrategy {readOnly}
- behaviourStrategy: BehaviourStrategy{readOnly}
- volume: AxisAlignedBoundingBox{readOnly}
- layer: Layer{readOnly}
- image: Image{readOnly}
- kinematicState: KinematicState{readOnly}
+ DynamicEntityImpl(kinematicState: KinematicState, volume: AxisAlignedBoundingBox, layer: Layer, image: Image, collisionStrategy: CollisionStrategy, behaviourStrategy: BehaviourStrategy)
+ collidesWith(entity: Entity): boolean
+ collideWith(entity: Entity): void
+ setPosition(pos: Vector2D): void
+ getPositionBeforeLastUpdate(): Vector2D
+ getVelocity(): Vector2D
+ setVelocity(vel: Vector2D): void
+ getHorizontalAcceleration(): double
+ update(durationMilli: double, levelGravity: double): void
+ copy(): Entity
+ getCollisionStrategy(): CollisionStrategy
+ getImage(): Image
+ getPosition(): Vector2D
+ getHeight(): double
+ getWidth(): double
+ getLayer(): Layer
+ getVolume(): AxisAlignedBoundingBox
+ copy(): Entity
ControllableDynamicEntity
- LEFT_ACC: double = -200 {readOnly}
- entity: T{readOnly}
- maxHorizontalVelocity: double{readOnly}
- floorHeight: double{readOnly}
- gravity: double{readOnly}
- startingPosition: Vector2D{readOnly}
- afterNextBounce: Optional<Runnable>{readOnly}
+ DynamicEntityImpl(entity: T,startingPosition: Vector2D, maxHorizontalVelocity: double, floorHeight: double, gravity: double)
+ hashCode(): int
+ boostHeight(): boolean
+ dropHeight():: boolean
+ moveLeft():: boolean
+ moveRight():: boolean
+ reset(): boolean
+ collidesWith(entity: Entity): boolean


+ setPosition(pos: Vector2D): void
+ getPositionBeforeLastUpdate(): Vector2D
+ getVelocity(): Vector2D
+ setVelocity(vel: Vector2D): void
+ getHorizontalAcceleration(): double
+ setHorizontalAcceleration(horizontalAcceleration: double): void
+ update(durationMilli: double, levelGravity: double): void
+ copy(): Entity
+ getCollisionStrategy(): CollisionStrategy
+ getImage(): Image
+ getPosition(): Vector2D
+ getHeight(): double
+ getWidth(): double
+ getLayer(): Layer
+ getVolume(): AxisAlignedBoundingBox
+ copy(): ControllableDynamicEntity<DynamicEntity>
- enforceHorizontalVelocityLimit(): void
<<Interface>>
EntityFactory
+ createEntity(level: Level, entityConfig:JSONObject): Entity
BallboyFactory
+ createEntity(level: Level, entityConfig:JSONObject): Entity
EnemyFactory
+ createEntity(level: Level, entityConfig:JSONObject): Entity
SquareCatFactory
+ createEntity(level: Level, entityConfig:JSONObject): Entity
CloudFactory
+ createEntity(level: Level, entityConfig:JSONObject): Entity
FinishFactory
+ createEntity(level: Level, entityConfig:JSONObject): Entity
EntityFactoryRegistry
- factoryRegistry: Map<String, EntityFactory> {readOnly}
+ registerFactory(entityType: String, factory: EntityFactory): void
+ createEntity(level: Level, entityConfig:JSONObject): Entity
StaticEntityFactory
- layer: Entity.Layer {readOnly}
+ StaticEntityFactory(layer: Entity.Layer)
+ createEntity(level: Level, entityConfig:JSONObject): Entity


<<Interface>>
Level
+ getEntities(): List<Entity>
+ getLevelHeight(): double
+ getLevelWidth(): double
+ getHeroHeight(): double
+ getHeroWidth(): double
+ getFloorHeight(): double
+ getFloorColor(): color
+ getGravity(): double
+ update(): void
+ getHeroX(): double
+ getHeroY(): double
+ boostHeight(): boolean
+ dropHeight(): boolean
+ moveLeft(): boolean
+ moveRight(): boolean
+ isHero(entity Entity): boolean
+ isFinish(entity Entity): boolean
+ getCheatMode(): boolean
+ ifEnd(): boolean
+ ifEnemy(): boolean
+ resetHero(): void
+ finish(): void
+ cheatMode(): void
+ end(): void
+ killEnemy(a:DynamicEntity): void
+ registerObserver(o: Observer): void
+ notifyAllObservers(): void
+ loadLevel(mementoOneLevel: MementoOneLevel): void
LevelImpl
- entities: List<Entity>
- engine: PhysicsEngine {readOnly}
- entityFactory: EntityFactory{readOnly}
- hero: ControllableDynamicEntity<DynamicEntity>
- finish: Entity
- squareCat: Entity
- levelHeight: double
- levelWidth: double
- levelGravity: double
- floorHeight: double
- floorColor: Color
- frameDurationMilli: double {readOnly}
- cheatMode: boolean
- isEnded: boolean
- blueScore: int
- redScore: int
- greenScore: int
- observers: List<Observer>
- afterUpdateJobQueue: Queue<Runnable> {readOnly}
+ LevelImpl(levelConfiguration: JSONObject, engine: PhysicsEngine, entityFactory: EntityFactory)
+ initLevel(levelConfiguration: JSONObject): void
+ ifSquareCat(a: Entity): boolean
+ getEntities(): List<Entity>
+ getLevelHeight(): double
+ getLevelWidth(): double
+ getHeroHeight(): double
+ getHeroWidth(): double
+ getFloorHeight(): double
+ getFloorColor(): color
+ getGravity(): double
+ update(): void
+ getHeroX(): double
+ getHeroY(): double
+ boostHeight(): boolean
+ dropHeight(): boolean
+ moveLeft(): boolean
+ moveRight(): boolean
+ isHero(entity Entity): boolean
+ isFinish(entity Entity): boolean
+ getCheatMode(): boolean
+ ifEnd(): boolean
+ ifEnemy(): boolean
+ resetHero(): void
+ finish(): void
+ cheatMode(): void
+ end(): void
+ killEnemy(a:DynamicEntity): void
+ registerObserver(o: Observer): void
+ notifyAllObservers(): void
+ loadLevel(mementoOneLevel: MementoOneLevel): void
<<Interface>>
PhysicsEngine
+ resolveCollision(a:DynamicEntity, b: StaticEntity,level: Level): void
+ resolveCollision(a:DynamicEntity, b: StaticEntity): void
+ enforceWorldLimits(a:DynamicEntity, level: Level): void
PhysicsEngineImpl
- frameDurationMilli: double {readOnly}
+ PhysicsEngineImpl(frameDurationMilli: double)
+ resolveCollision(a:DynamicEntity, b: StaticEntity,level: Level): void
+ resolveCollision(a:DynamicEntity, b: StaticEntity): void
+ enforceWorldLimits(a:DynamicEntity, level: Level): void
- reflectX(entity: DynamicEntity): void
- reflectY(entity: DynamicEntity,levelGravity: double): void


<<Interface>>
Observer
+ update(): void
TotalLevelObserver
- allLevels: List<Level> {readOnly}
- textRed: Text {readOnly}
+ TotalLevelObserver(allLevels: List<Level>, test: Text)
+ update(): void
CurrentLevelObserver
- level: Level
- textRed: Text
- textGreen: Text
- textBlue: Text
- blueScore: int
- RedScore: int
- GreenScore: int
+ CurrentLevelObserver(level: Level, textBlue: Text, textRed: Test, textGreen: Text)
+ update(): void
CaretakerOneLevel
- memento: MementoOneLevel
+ storeLevelMemento(memento: MementoOneLevel): void
+ uploadLevelMemento(): MementoOneLevel
MementoOneLevel
- entities: List<Entity>
- hero: ControllableDynamicEntity<DynamicEntity>
- finish: Entity
- squareCat: Entity
- isEnded: boolean
- blueScore: int
- redScore: int
- greenScore: int
+ MementoOneLevel(entities: List<Entity>, hero: ControllableDynamicEntity<DynamicEntity>, finish: Entity, squareCat: Entity,isEnded: boolean, blueScore: int, redScore: int, greenScore: int)
+ getAllEntities(): List<Entity>
+ getHero(): ControllableDynamicEntity<DynamicEntity>
+ getFinish(): Entity
+ getSquareCat(): Entity
+ getIsEnded(): boolean
+ getRedScore(): int
+ getBlueScore(): int
+ getGreenScore(): int
Caretaker
- memento: Memento
+ storeMemento(memento: Memento): void
+ uploadMemento(): Memento


Memento
- levelIndex: int
+ Memento(levelIndex: int)
+ getLevelIndex(): int
<<Interface>>
GameEngine
+ getCurrentLevel(): Level
+ startLevel(): void
+ boostHeight(): boolean
+ dropHeight(): boolean
+ moveLeft(): boolean
+ moveRight(): boolean
+ ifNextLevel(): boolean
+ ifEndGame(): boolean
+ tick(): void
+ cheatMode(): void
+ storeGame(): Memento
+ loadGame(m: Memento): void
GameEngineImpl
- allLevels: List<Level> {readOnly}
- level: Level
- levelIndex: int
- endGame: boolean
- caretakers: List<CaretakerOneLevel>
+ GameEngineImpl(levelIndex: Integer, allLevels: List<Level>)
+ getCurrentLevel(): Level
+ startLevel(): void
+ boostHeight(): boolean
+ dropHeight(): boolean
+ moveLeft(): boolean
+ moveRight(): boolean
+ ifNextLevel(): boolean
+ ifEndGame(): boolean
+ tick(): void
+ cheatMode(): void
+ storeGame(): Memento
+ loadGame(m: Memento): void
<<Interface>>
BackgroundDrawer
+ draw(model:GameEngine, pane: Pane): void
+ update(xViewportOffset: double, yViewportOffset: double): void
BlockedBackground
- sky: Rectangle
- floor: Rectangle
- model: GameEngine
- pane: Pane
+ draw(model:GameEngine, pane: Pane): void
+ update(xViewportOffset: double, yViewportOffset: double): void
<<Interface>>
EntityView
+ update(xViewportOffset: double, yViewportOffset: double): void
+ matchesEntity(entity: Entity): double
+ markForDelete(): void
+ getNode(): Node
+ isMarkedForDelete(): boolean
EntityViewImpl
- entity: Entity {readOnly}
- node: ImageView{readOnly}
- delete: boolean = false
+ EntityViewImpl(entity: Entity)
- getViewOrder(layer:Entity.Layer): double
+ update(xViewportOffset: double, yViewportOffset: double): void
+ matchesEntity(entity: Entity): double
+ markForDelete(): void
+ getNode(): Node
+ isMarkedForDelete(): boolean



Strategy pattern: yellow
Factory method pattern: blue
Observer pattern: green
Memento pattern: red

Code Review:

OOP design principles:

1. Single Responsibility Principle:
A class should have only one reason to change. In the codebase, all the factory classes only have the responsibility of creating the entities based on the configuration file. For all the behaviour strategies and collision strategies, each behaviour strategy will only build the behaviour of one kind of entity and each collision strategy will be only changed by the entity that collides. Hence, those factory, behaviour, and collision classes achieve the Single Responsibility Principle of OOP.

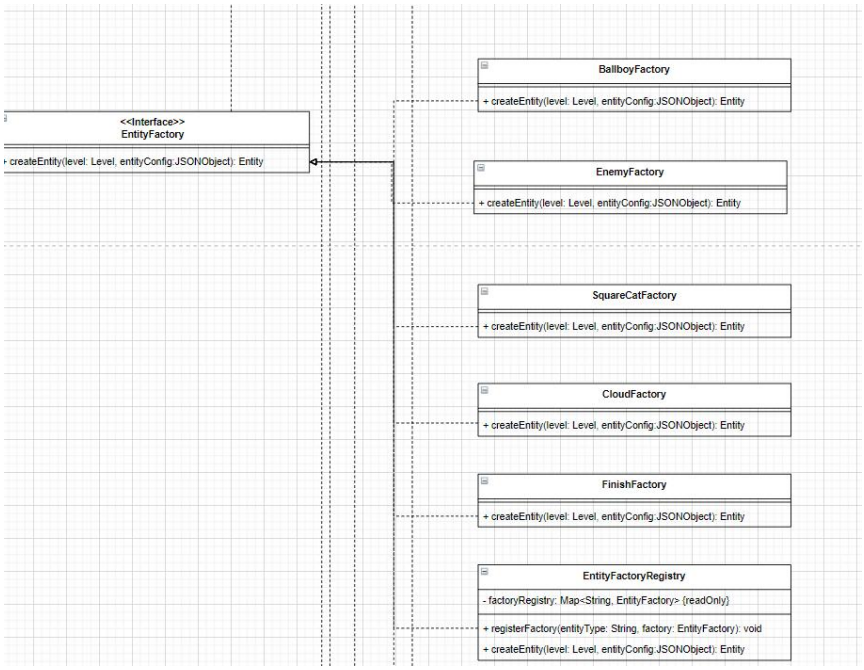


Figure1: factory

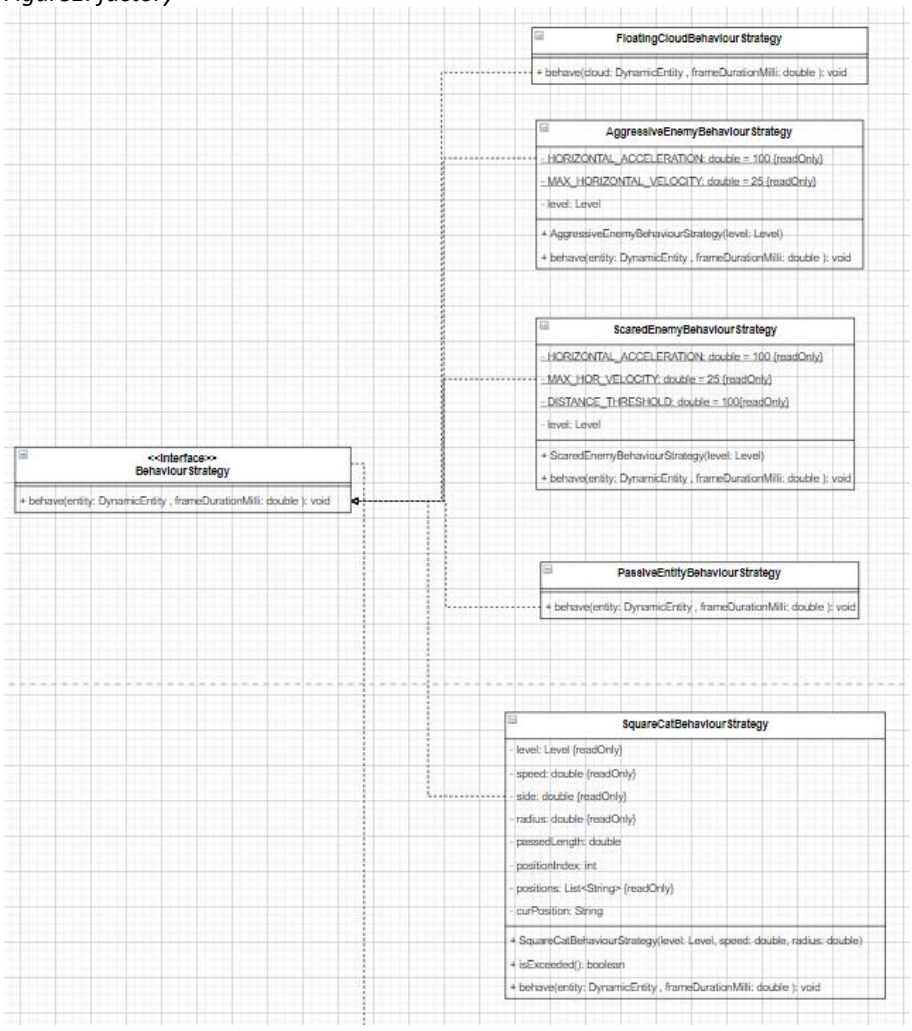


Figure2: behaviour

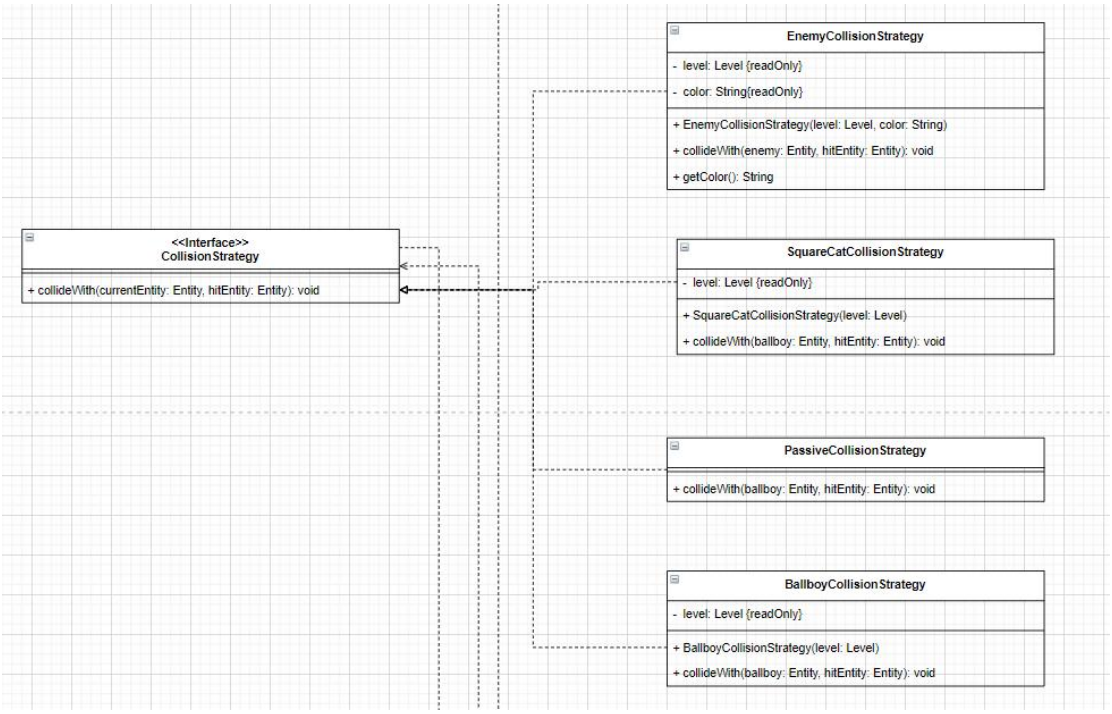


Figure3: collision

2. Open Closed Principle:

In the factory package, it has an EnittyFactory interface, all the entity factories will implement the EnittyFactory interface to create the corresponding entity and be initialized in the initLevel() of the LevelImpl class. When any new entity needs to be added, only need to build the new corresponding factory class that implements the EnittyFactory interface to create an entity without affecting the building of other entities which achieve the Open Closed Principle of OOP (Figure1). In addition, the same reason for the collision package and behaviour package, they also have interfaces that will be used in other classes, any new behaviour strategy and collision strategy only need to implement the corresponding strategy interface without changing other strategies.(Figure2 & Figure3)

3. Liskov Substitution Principle:

For the Entity interface, the two abstract classes DynamicEntity and StaticEntity will implement it and they have the corresponding concrete class to extend and achieve all the functions. Also, a ControllableDynamicEntity will extend from DynamicEntity. For most methods, they will get an Entity parameter and they will check what exact kind of entity it is in the method. In addition, all the Entity could be replaced by the child class which achieve the Liskov Substitution Principle of OOP

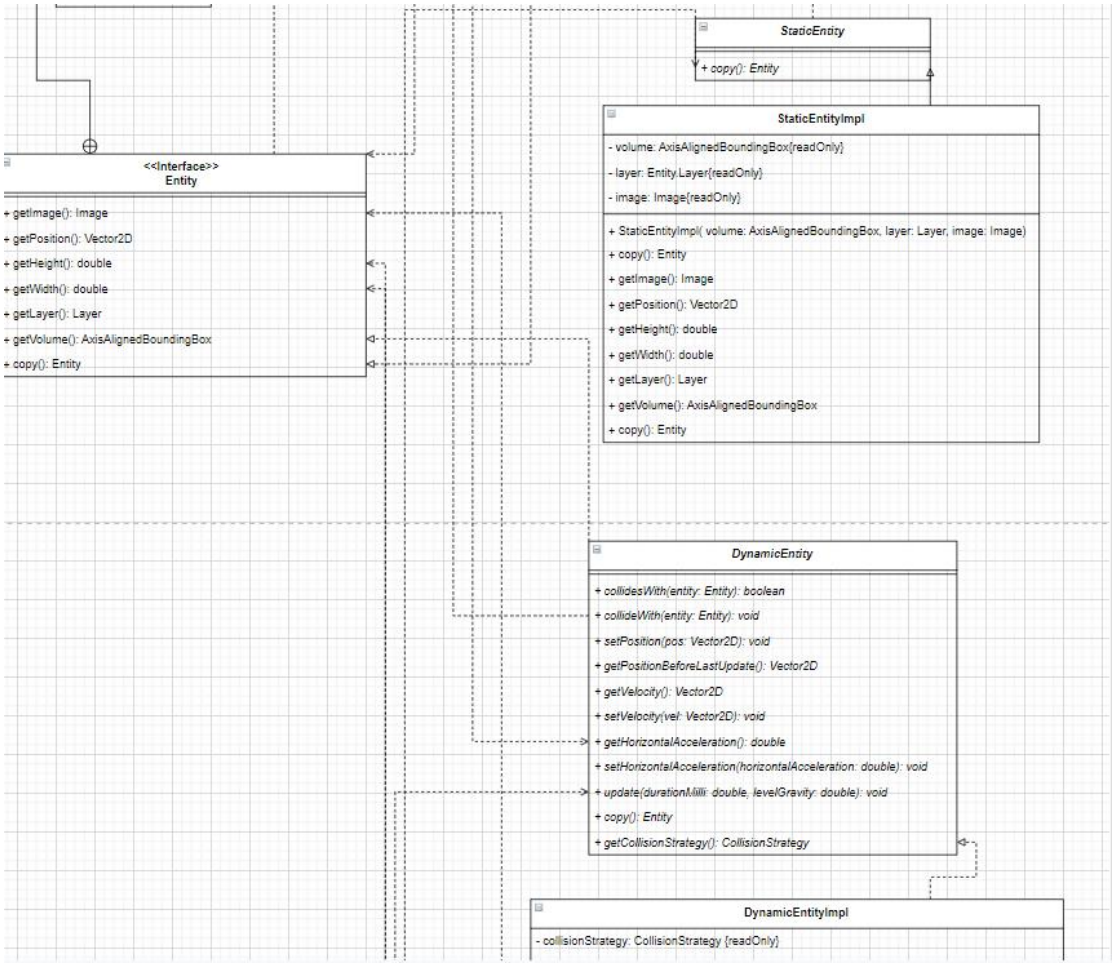


Figure4: entity

4. Law of Demeter:

For the collision and behavior strategy, they do not have any connection with each other even though they are the strategy for the same entity. They will be connected to the entity in the building of DynamicEntity. They only achieve one kind of function of one type of entity without connection of other things which achieve the Law of Demeter of OOP. (Figure1 & Figure2)

5. Dependency Inversion Principle:

In the codebase, most of the dependence relies on abstraction rather than concrete. For example, the Level interface relies on the Entity interface and the DynamicEntityImpl relies on the BehaviourStrategy interface and CollisionStrategy interface which make the detail depend upon abstractions. Hence, the codebase achieve the Dependency Inversion Principle(Figure5 & 6)

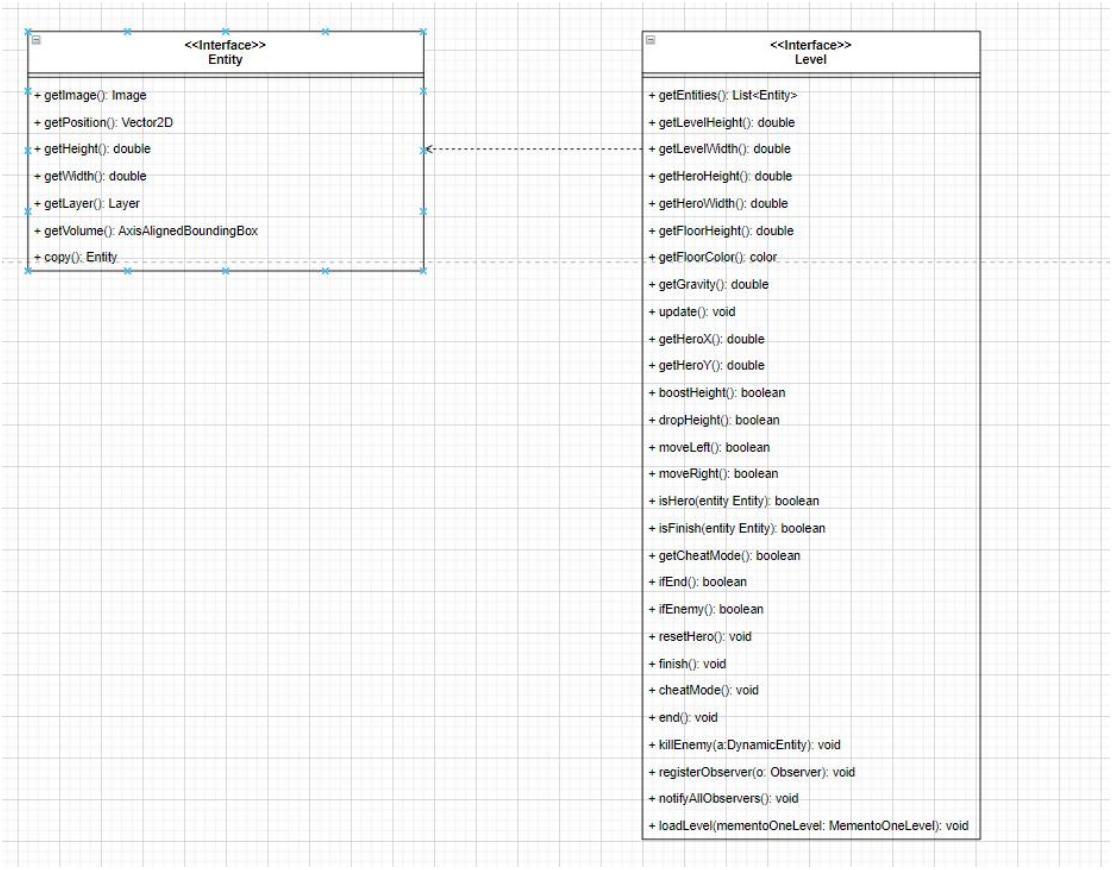


Figure5: level and entity

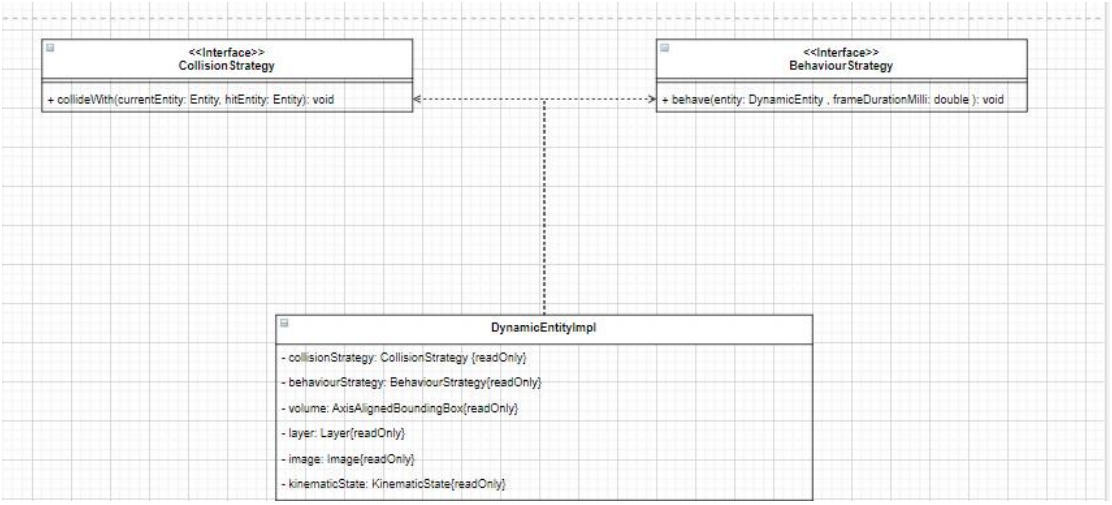


Figure6: dynamic entity relies on strategy

6. Interface Segregation Principle:

In the codebase, it builds many interfaces to refine the interface as much as possible and puts each method to complete a specific function in a special interface. For example, it separate strategy into collision and behaviour with different interfaces to make each method achieve a more specific function which consistent with the Interface Segregation Principle of OOP(Figure1 & 2)

Design Patterns:

1. Factory method pattern:

For the factory method pattern, it uses this pattern to create different types of entities based on the configuration file. In the package, it builds an EntityFactory interface with a create() method and the

separate concrete entity to create each kind of concrete entity which achieves the factory method pattern. (Figure3).

2. Strategy pattern:
For strategy pattern, it uses this pattern to realize the different behaviors of entities in the game and different reactions when entities collide. It creates two strategies which are collision strategy and behaviour strategy. For the collision strategy, it creates a CollisionStrategy interface with a collideWith() method and creates different concrete strategies for the corresponding entity that may be collided. Then, the collideWith() method will be used in other methods that will check the collision, such as the update() method in the LevelImpl class. For the behaviour strategy, it creates a BehaviourStrategy interface with a behave() method and creates the corresponding concrete strategy class for different entities. These behaviours will be used when creating the entities. (Figure1&2)

Documentation:

1. Readme:
This file first describes how to start and run the game with the Gradle command and how to operate the game by using the arrow button. Secondly, it interprets the game data source file which is the configuration file, illustrates the meaning of each type of data and the sub-data in it by giving examples Then, it lists the strategies that are used in the code and list out the classes in the code related to the strategies. Finally, it lists the source of the picture.

2. Comments:
The comments in the code provide the interpretation and tips for the functions of some classes or methods. At the same time, the comments of some code provide me with design ideas.

3. Configuration File:
This file provides the data that will be used in the game and help us to change the basic data in the game.

Benefit:
The above makes me achieve the functionality easier.
Firstly, for the OOP principle, the implementation of the Single Responsibility Principle can make the function of the class clearer and easier to use and can give me a clearer understanding. The implementation of the Open Closed Principle can make it more convenient for me to add new functions or classes without affecting other designed classes. The implementation of the Liskov Substitution Principle and Dependency Inversion Principle allows me to design child classes easier so that new child classes could also be called in the original method because they all depend on abstraction. The implementation of the interface isolation principle could clearly reflect the function of each interface and the dependence between implementation abstractions, so as to facilitate the implementation of the Dependency Inversion Principle. The implementation of Demeter's law can reduce the coupling between classes and avoid my design errors affecting other classes.

Secondly, for the strategy, the Factory method pattern could help me create new characters from the configuration file and add them to the game. Moreover, each entity factory could create an entity according to the attributes owned by itself, which is clear and convenient to use. The implementation of the strategy pattern avoids repeated code. At the same time, it allows me to flexibly add new algorithms without modifying the original code, provides different implementations of the same behavior, and provides clear design ideas to me.

Finally, the readme file gives me a clear introduction of the codebase which helps me to understand it. The comment could make the functionality more understandable and provide me with design ideas. Moreover, the configuration could help me define the basic data of the game.

Feature Extension:

1. Actual extension:

For the level transition, I create a boolean variable in the Level to check whether one level is finished and a boolean variable in the GameEngine to check whether all the levels are finished. In the tick() method of GameEngine, it has all the levels and if the current level is finished, the level index will add one to the next level until all the levels are finished, the GameWindow will get it and display the winner picture.

For the squarecat, I use the same way as other entities to add the “cat” information in the configuration file first. Then, build the SquareCatFactory to create the square cat entity in the game， and create corresponding SquareCatCollisionStrategy and SquareCatBehaviourStrategy based on the requirement.

For the score, I use the Observers design pattern to build an Observer interface with an update() method to form the output text first. Then, create two concrete observers to implement it which are the CurrentLevelObserver to observe the scores of three colors in a single level and TotalLevelObserver to observe all the scores of all the levels and add them up. The Level contains all observers and it will notify all the observers once the score is updated. In the App, I create those texts and observers and add them to the pane. Also, I add enemy color information to divide the enemies into three colors corresponding to the color score.

For the save and load, I use the Memento design pattern. I build a MementoOneLevel class to save the changed elements in a level and a CaretakerOneLevel class to save the MementoOneLevel. In the Level, I create saveLevel() method to save the copy of all the changed elements to the MementoOneLevel and loadLevel() method to set the save elements back. Furthermore, I also create a Memento to save the level index in the GameEngine and a Caretaker to save Memento. In the GameEngineImpl, it has a list of levels and a list of CaretakerOneLevel. Also, I create a storeGame() method to save every level and level index to the Memento and a loadGame() method to load the level index and every copied level information out. Finally, the Caretaker will be created in the KeyboardInputHandler class, if the "S" is pressed, it will use the storeGame() method in the GameEngine and save it in the Caretaker; if the "Q" is pressed, it will use load the Memento out.

2. Application of OO design principles:

I used the Open Closed Principle when designing the square cat to help me build the squrecat entity and the corresponding strategies. (Figure1&2) I used the Liskov Substitution Principle and Dependency Inversion Principle in the copy of the entities information which could help me to achieve the copy of entities from the parent class to the child class due to I could just use the copy() method of the Entity without checking the exact entity except for the special entity. (Figure4) I use the Interface Isolation Principle to build the Observer interface to let all the observers implement it. (Figure7)

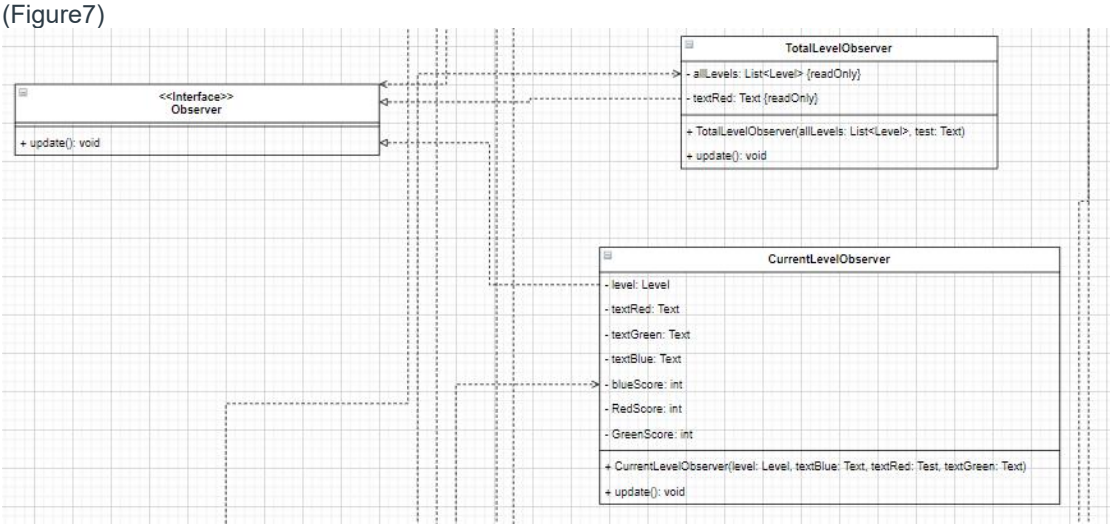


Figure7: observer

3. Document any design patterns used

Observer pattern:

Building an Observer interface, and creating a CurrentLevelObserver class and a TotalLevelObserver class to implement Observer. They will observe the scores of three colors in one level and the total score of all levels respectively. (detail usage in the Feature Extension)

It reduces the coupling relationship between the level and the observer, which is an abstract coupling relationship. Comply with the Dependency Inversion Principle of SOLID and the low coupling of GRASP. (Figure7)

Memento pattern:
Build a MementoOneLevel to record the information of one level and a CaretakerOneLevel to record MementoOneLevel. Also, build a Memento to record level index and all level information and a Caretaker to record Memento. (detail usage in the Feature Extension)

The encapsulation of the internal state is realized. All state information of a level is saved in the MementoOneLevel and managed by the CaretakerOneLevel; all the level and level index is saved in the Memento and managed by Caretaker which Complies with the Single Responsibility Principle of SOLID and Infomation Expert of GRASP. (Figure8)

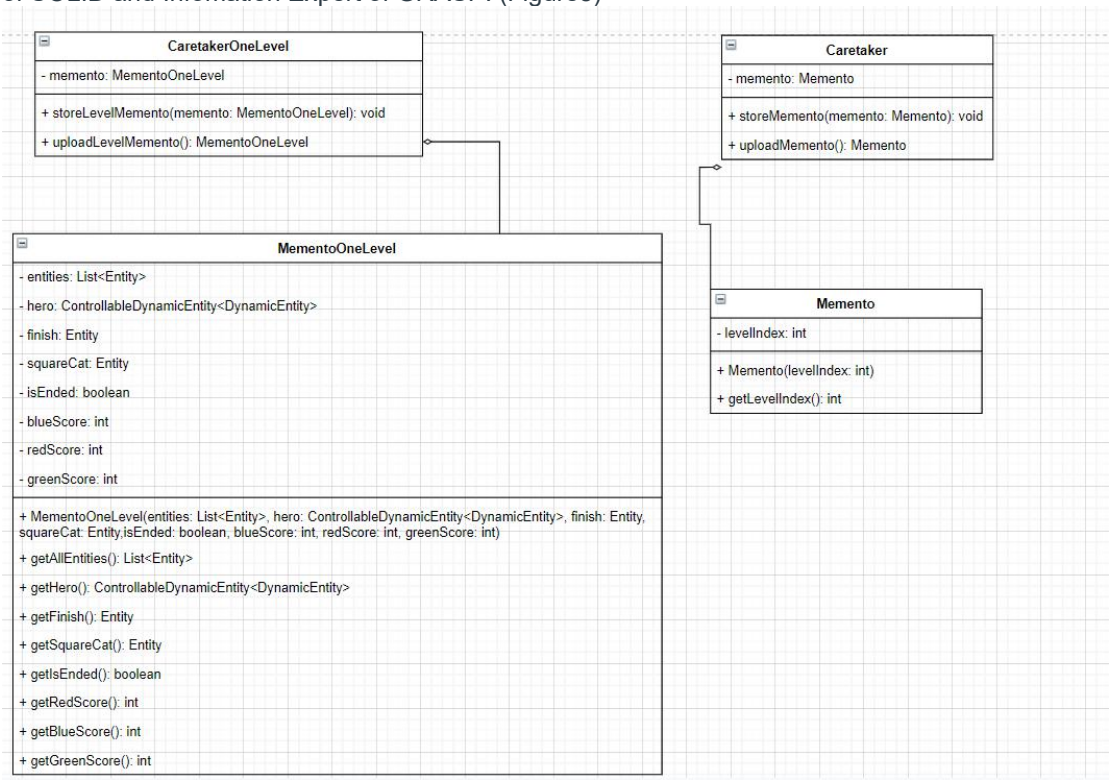


Figure8: memento

Factory method pattern:
Add new SquareCatFactory to implement the EntityFactory interface and create the square cat entity based on the configuration file.

We only need to know that the square cat belongs to Entity and do not need to know other implementation entities which achieve the Liskov Substitution Principle and Dependency Inversion Principle of SOLID. Also, the Factory method pattern allows us to add any new factory and entity without affecting others which also achieves the Open Closed Principle of SOLID. (Figure3)

Strategy pattern:
Add new SquareCatCollisionStrategy and SquareCatBehaviourStrategy to implement CollisionStrategy and BehaviourStrategy respectively.

We could add any new strategy under the CollisionStrategy interface or the BehaviourStrategy interface which achieves Open Closed Principle in SOLID. Also, two kinds of interfaces achieve the Interface Isolation Principle of SOLID. (Figure1&2)

4. Reflect on extension design

First of all, for enemies with different colors, I only inserted red, blue, and green, so I only designed the scores and observations of the three colors and did not consider the addition of other colors. If there are enemies of new colors, I need to add data corresponding to colors in multiple classes in the code again. It is very troublesome. I can't automatically identify and add data of new colors, and not achieve the Open Closed Principle.

Secondly, in the memo, because my design will delete the last memo every time I load the game, I have to save it again in the method of loading the game to ensure that it can be loaded multiple times. This algorithm may increase the running time.

In conclusion, even though there are small issues in my design such as not considering the further color of the enemy and the design may increase the running time, I achieve all the required goals and most of the design achieve the OOP.