Code C

OOP DESIGN PRINCIPLE ANALYSIS & DESIGN PATTERN ANALYSIS

Many design patterns are used for this assignment and I will be talking about classes in the model package first.

One of the design patterns would be Builder Pattern. It is used for creating entities after extracting the information from the JSON file. JSONExtractor class extracts the information and LevelBuilder class takes this information and creates a new Level objects using the information. Those information from the JSON file can be changed easily by editing the numerical numbers and there would not be errors popping out from the methods in the classes. This has obeyed the Single Responsibility Principle. This is because the two classes only have single job to do, which is to extract the entities' information and set them to a new Level class respectively. This shows that they have minimise the coupling between their individual components, therefore by modifying one of the methods it would not affect other methods in the class. One suggestion for the JSONExtractor class would be handling of new array (input by user) in the JSON file. Also, if possible, JSONExtractor class only extract information from the JSON file and store in the respective arraylists and a new class is created to create new entities, with the aid of the information from the JSONExtractor class. Then, LevelBuilder class get the entities from that new class and create a new Level. This is because with a deeper studying of JSONExtractor class, it does extract information, as well as creating new entities. To lessen its coupling, a new class could be created for it.

Factory Pattern is also used for creating entities like slime and the platforms. Abstractions are used to create entities that have different behaviours, such as movable or static. Firstly, for the creation of all objects, to separate the entities, interfaces like EnemyFactoryInterface, EntityFactoryInterface and MovableEntityFactoryInterface are created. This has also obeyed the Single Responsibility Principle. This can be shown in the classes, like EnemyFactory, EntityFactory and MovableEntityFactory. The classes implement the functionalities from the interfaces. Every classes have a single responsibility which is to create new entities. Overall, they are used in JSONExtractor class for the creation of the entities. The different interfaces hold different responsibilities to create different type of entities. This could not be achieved if only one interface is presented.

Adapter Pattern is also used for setting up the entities in the game. Firstly, an Entity interface is created to set the position and behaviours of the entities. Then, more interfaces, like Controllable, Enemy and MovableEntity, are created to implement Entity interface. Beside the interfaces, abstractions like abstract classes are also implemented, such as EnemyAbstraction, EntityAbstraction and MovableEntityAbstraction. The diversity of the abstractions did not violate Interface Segregation Principle. This is because each of them has different respective responsibilities and all of the functionalities are well used. For example, when an Entity object is called, the different type of Entity objects could be called and used in the calculation. With the aid of the functionalities in the interfaces, such as Boolean methods, the specific type of Entity object would be able to perform in the ideal functions. In addition, the variety of the classes, used to set up entities, is very effective in a way to differentiate the type of entities. Hence, this has also obeyed the Open Close Design Principle. This is shown that if only Entity interface is presented, the functionalities that required specific Entity objects would be hardly to be carried out. In order to perform that, many attributes, classes and ‘if-else’ statements are required, and this is not a good or ideal coding design because when a change is amended, other classes might be affected. And this has clearly violated the Open Close Design Principle. This Adapter Pattern allows user or programmer to create more classes which extend from the abstractions for the creations of more entity types, and this will not affect the existing code. In addition, Dependency Inversion Principle can be found too. When creating new entities, each class is responsible of a specific type of entity. Any changes in the individual classes will not affect other classes that have the same abstraction. This makes the creation of entities more flexible and effective.

There are interfaces that control the movements of the entities, such as hero and slime. For example, Controllable, EnemyMovement, GravitationalState and MovableEntity. Adapter and Strategy Pattern are presented here.

For the Adapter Pattern, it can be seen from the classes that implement those interfaces, such as Hero and Slime. In these two classes, Liskov Substitution Principle can be found. This is because beside overwriting the methods from the interfaces, Hero and Slime classes have more functionalities than the respective interfaces. For example, Hero class has handleImmunity() and animate() methods and Slime class has adjustSlime(String colour) and manageTicks(). Those addition functionalities are private, and this is a good designing because those methods would not be changed by other classes so that they would not leave impact on the existing code. Thus, this is a correct way of writing the access modifier.

For the Strategy Pattern, some of the existing functionalities are presented in the run time of the application. For example, the tick() method in Hero class. It contains the functionalities that controls the horizontal movement of the hero, as well as the handling of horizontal collision with the slime. The moveHorizontally(Controllable hero, boolean leftCollision, boolean rightCollision) method in Slime class is also called in the run time. There are many more functionalities are call in the run time that has Strategy Pattern that I will talk about them later in other classes. In general, they all have single responsibility to perform ideally. This shows that they have obey the Single Responsibility Principle. Individually, those methods have minimum coupling to deal with and by modifying one will not affect other methods. Whenever a ‘bigger’ function requires to combine different functionalities from individual classes, those functionalities will always behave the same and only way. Hence, this makes the writing of the logic much easier and effective.

For the Slime class, there is an interesting behaviour. Two classes implemented from EnemyMovement interface which deal with the movement of the slimes. Both classes inherit the method move(Enemy enemy, Controllable hero, boolean leftCollision, boolean rightCollision) but perform it differently. This shows when EnemyMovement object is called, the respective slime would behave differently from other slimes. This follows Interface Segregation Principle as the functionality in the interface is well used by two implemented classes. Without this interface, in order to write different slime behaviours, much work is required in a single class which result in many attributes and methods created in that class. This has clearly violated Single Responsible Principle because that class requires much coupling between the individual components and changing one of the functionalities would impact other functionalities. With this EnemyMovement interface, user or programmer can add more behaviours to the slime by creating classes which implement this interface; therefore, this interface is able for extension at all time which satisfy the Open Close Design Principle. In addition, Dependency Inversion Principle can alse be found. As each class is responsible of a specific type of movement, any changes in the individual classes will not affect the other class. This makes the process more flexible and effective. I would suggest of adding a vertical movement functionality of the slime to the EnemyMovement interface. This is to create more interesting slime behaviours, but one needs to ensure not violating Interface Segregation Principle as it would be pointless if the functionalities in the interface is not relevant to the user.

For the controlling of the vertical movement of the hero, classes like Gravity and those which implement from the GravitationalState interface is used. Inside the Hero class, the private method setGravitationalState(GravitationalState gravitationalState) controls the GravitationalState objects and thus the vertical movement of the hero. Since the method moveVertically() is called in the run time, the direction (falling or bouncing or jumping) is controlled by the GravitationalState objects. Each of these classes are responsible with only one direction so they follow the Single Responsibility Principle. In Hero class, whenever a vertical behaviour is called, the respective GravitationalState objects is assigned to the hero to perform the correct vertical movement. Now is clearly to see that moveVertically() has the Strategy pattern. It will always call the GravitationalState interface to perform the action and it will not do other stuff. All the possible vertical behaviours are listed in the respective classes and hence any expansion is unnecessary for hero's vertical movement.

In Layman's Term, GameEngine controls Level and Level controls Entity. I will talk about Level first.

LevelImplementation class implements Level interface. It also has more functionalities than Level interface. Those addition functionalities are private hence they will not cause impact to other codes because they are unreachable from other classes. This tells that there is Liskov Substitution Pattern in there. The functionalities are correctly implemented and the respective class that implemented Level interface would be called when Level is called, without creating errors. This is essential because in other classes which only call for Level object, everything needs to run smoothly and up to expectation. If anything needs to modify, LevelImplementation class can be amended at any time, without affecting the existing code. Hence, it follows Open Close Design Principle.

In general, LevelImplementation class controls all of the entities' behaviours and are assigned to the tick() function. This function will always produce the same outcome unless the internal methods are modified. Hence, it follows the Strategy Pattern. This tick() function will be called in the run time of the program. Those internal methods are individual entity's behaviour and all of them have private access modifier. This shows that no class is able to change individual entity's behaviour and hence the tick() function would be able to run nicely. All in all, LevelImplementation class obeys Single Responsibility Principle. This is because it only has one responsibility to carry out, which is to control the entities, include their movements and positions. It acts as a storeroom for all of the entities. For this assignment, its role is significant. Imagine if there is no collection of the entities, how messy the project would become.

As for the GameEngine interface, a class called GameEngineImplementation is implemented from it. Alike LevelImplementation class, GameEngineImplementation class also has more functionalities than its abstraction. I have added my own features in those two classes, but I will talk about them later. A new method, createLevels() is created in the GameEngineImplementation class. It is used to create new levels associated with the JSON file. However, its access modifier is public. I would suggest making it private to prevent modification from somewhere else. If this method is anyhow called in the run time, the program would be messed up. Thus, it is always good to maintain a proper encapsulated and organised code. Actually, GameEngineImplementation class follows Interface Segregation Principle. This can be proved in all of the classes that required GameEngine object to perform. All of the expectations are achieved, and no errors are found. Hence, GameEngine interface is essential for the project and GameEngineImplementation class has correctly implemented it. Beside managing the right level to be shown, GameEngineImplementation class also responsible for the lives of the hero, as while as the game duration. Here, it can be seen that GameEngineImplementation class has several responsibilities to take care. This do violate Single Responsibility Principle and GameEngineImplementation class should reduce coupling between its individual components. For example, lives of the hero are also required in the Level. If anything is wrongly edited, the game could have ended before the hero loses all of his lives. I would suggest creating new classes to take care of the lives of the hero, as well as the game duration. And then combine them together in GameEngineImplementation class or a new class. This can follow the Creational Design Pattern and therefore a well organised work would be produced. This also make the work tidier.

I will now talk about classes in the view package.

EntityView and BackgroundDrawer objects are created for the images appears in the game. In my opinion, the both interfaces could be abandoned because there are only one class implements each of them. I feel that even they do not violate the Interface Segregation Principle, Runner class can just call ParallaxBackground class and EntityViewImpl class (which implemented the EntityView and BackgroundDrawer interfaces), no errors would be popped out. A programmer should always program for the abstractions, not for the implementations. However, to me, the two interfaces are useless in this project and they can be replaced by two classes. The present of them is not really necessary as they do not need any expansion or diversion.

KeyboardInputHandler class controls the keys pressed by the user. When a user pressed an expected key, the program will then call the functionalities in the GameEngine object to perform. This class obeys the Single Responsibility Principle. This is because it only has one responsibility which is to intake the key pressed by the user and signal the GameEngine object to perform the expected behaviour. As a reader, this class could be comprehended easily due to low coupling between its individual components.

Runner class has many responsibilities to take care, such as manage the lives of the hero and the user’s view on the game window. It has a method called draw(), which is performed in the run time. Although it has a private access modifier, when another class, such as GameWindow class, called its run() method, draw() will be called then hence the GameEngine object’s tick() function. Thus, the Strategy Pattern is shown. As run() is public, whenever the user wants to start the program, it will be called, and it will always output the same and only outcome, unless the internal functionalities are edited. The starting, ending and all of the behaviours of the entities are inside the draw(), which is called by run() in the Runner class. Beside GameEngine object’s tick() function, many of the conditions are also presented in the draw() method. However, I would suggest to use methods or classes for the conditions in the draw() method. Not only making draw() looks neater, but also a good designing. For example, inside the draw() method, some conditions like managing the lives of the hero and the finishing of the game, could be rebuild with a Strategy Method. A single method is called only and inside that method, all of the conditions are being taken care. New classes could be created to assist it. This is also to lessen the coupling in the Runner class.

GameWindow class is always called by the main class, App.java, to start the program. It is also used to set the dimension of the game window. The main class would then start the whole application while setting up the dimension of the game window, as well as passing a specific JSON file to a new GameEngine class.

CODE STYLE & DOCUMENTATION ANALYSIS

After looking through https://oracle.com/technetwork/java/codeconventions-150003.pdf, the given codebase do follow the given style guild. Nothing beside Java language is used. There are also comments and Javadoc provided in the code for me to comprehend better. Methods and attributes are labelled correctly too.

If it does not follow the given style guide, the original programmer should take the codebase back first and correct the style guide immediately. This is because as a reader, it is very difficult to differentiate whether the codebase or the given style guide is wrong.

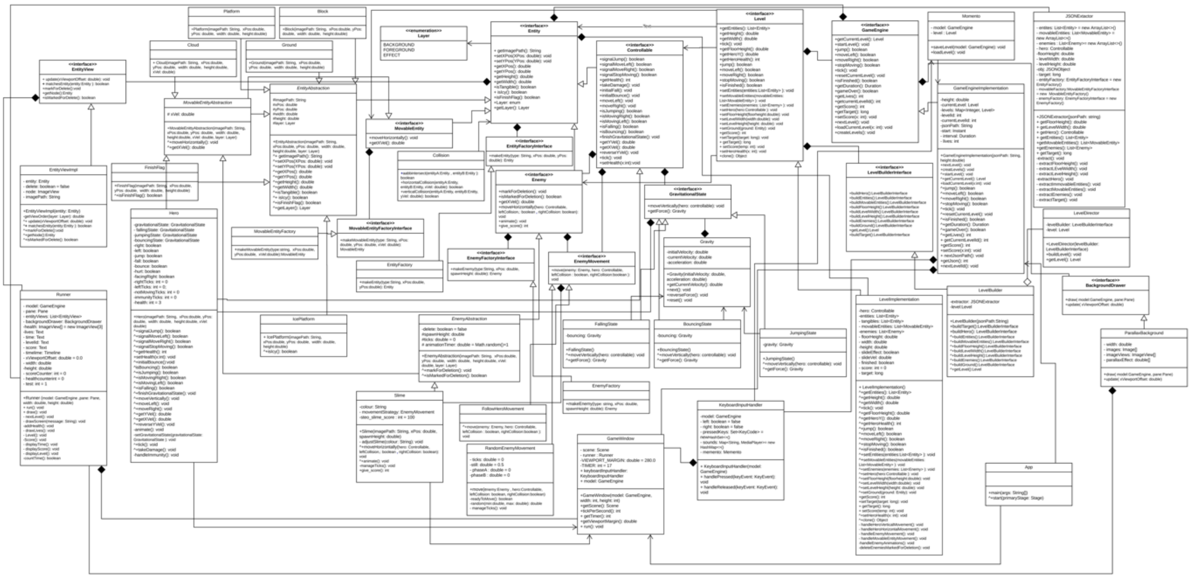
Actually, when the codebase is differing from the given style guild, it is very troublesome for the reader, especially when the reader is eager to learn from it. For example, when there is a difference between the codebase and style guild, the reader is unable to determine which is correct. Hence this may deter the reader from learning, or even preventing the reader to learn the expected ideal knowledge.

To me, the given codebase is well documented. It is well organized, and all of the abstractions are easy to understand because of the name of the files and the way the writer wrote it. I would suggestion of creating more packages for a better organization of the abstractions of the class and interfaces, and thus a better comprehending of the whole project.

OVERALL IMPACT ANALYSIS

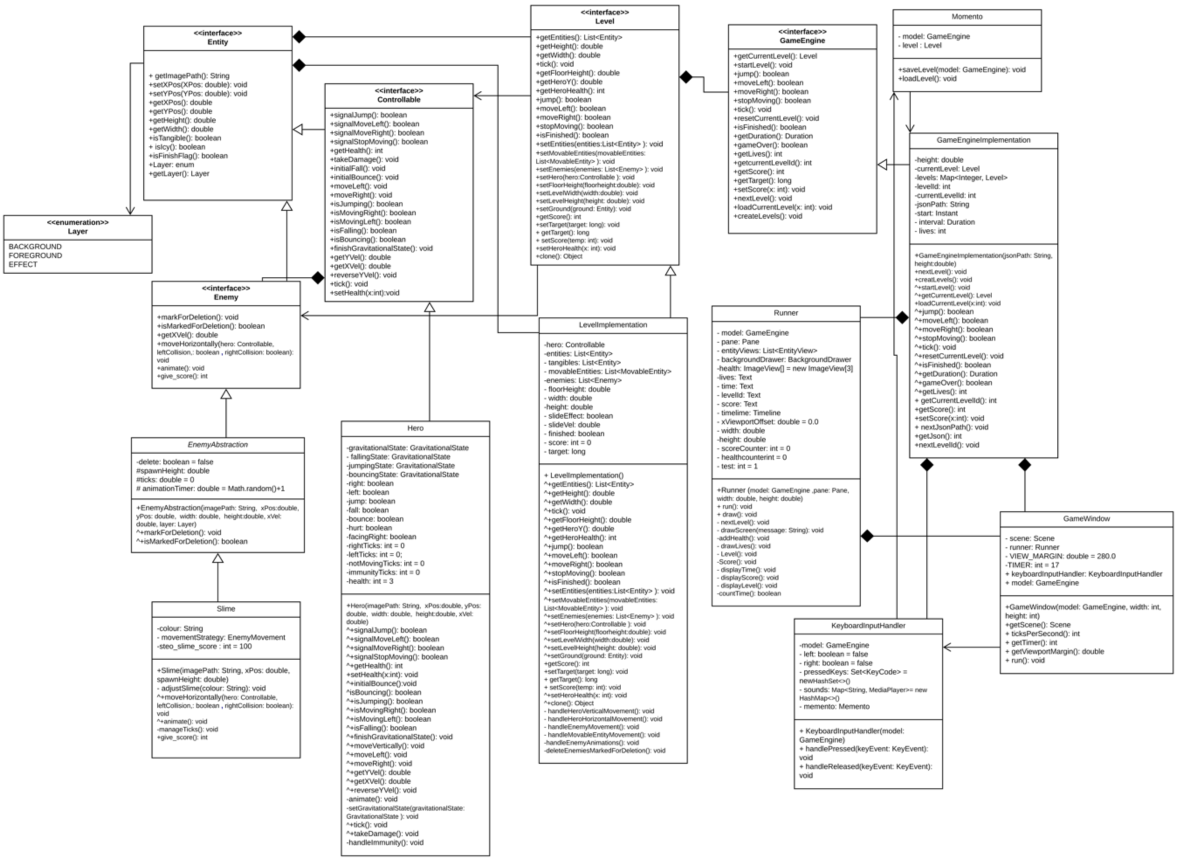
It is actually quite challenging to achieve the required extensions. This is because this is my first time studying such a designed code. Many abstractions, the uses of the interfaces, make me very confusing initially. Although the individual functionalities are moderate to understand, but I took two days to comprehend most of the design patterns and principles behind this project. By studying through the past tutorials and lectures, I started to like this codebase. It does help me better learning in design patterns and principles. For example, I have never used Builder Pattern before in any of the assignment. However, after studying the LevelBuilder.java and JSONExtractor.java, I started to know how the Builder Pattern actually works.

The UML for all of the classes and interfaces


Platform
+Platform(imagePath: String, xPos:double, yPos: double, width: double, height:double)
Block
+Block(imagePath: String, xPos:double, yPos: double, width: double, height:double)
<<enumeration>>
Layer
BACKGROUND
FOREGROUND
EFFECT
Cloud
+ Cloud(imagePath: String, xPos:double, yPos: double, width: double, height:double, xVel: double)
Ground
+Ground(imagePath: String, xPos:double, yPos: double, width: double, height:double)
<<interface>>
EntityView
+ update(xViewportOffset: double): void
+ matchesEntity(entity:Entity): boolean
+markForDelete():void
+getNode():Entity
+isMarkedForDelete(): boolean
MovableEntityAbstraction
# xVel: double
+MovableEntityAbstraction(imagePath: String, xPos:double, yPos: double, width: double, height:double, xVel: double, layer: Layer)
^+moveHorizontally(): void
^+getXVel(): double
EntityAbstraction
#imagePath: String
#xPos: double
#yPos: double
#width: double
#height: double
#layer: Layer
+EntityAbstraction(imagePath: String, xPos:double, yPos: double, width: double, height:double, layer: Layer)
^+ getImagePath(): String
^+setXPos(XPos: double): void
^+setYPos(YPos: double): void
^+getXPos(): double
^+getYPos(): double
^+getHeight(): double
^+getWidth(): double
^+isTangible(): boolean
^+ isIcy(): boolean
^+isFinishFlag(): boolean
^+getLayer(): Layer
<<interface>>
MovableEntity
+moveHorizontally(): void
+getXVel(): double
<<interface>>
Entity
+ getImagePath(): String
+setXPos(XPos: double): void
+setYPos(YPos: double): void
+getXPos(): double
+getYPos(): double
+getHeight(): double
+getWidth(): double
+isTangible(): boolean
+ isIcy(): boolean
+isFinishFlag(): boolean
+Layer: enum
+getLayer(): Layer
Text
<<interface>>
Controllable
+signalJump(): boolean
+signalMoveLeft(): boolean
+signalMoveRight(): boolean
+signalStopMoving(): boolean
+getHealth(): int
+takeDamage(): void
+initialFall(): void
+initialBounce(): void
+moveLeft(): void
+moveRight(): void
+isJumping(): boolean
+isMovingRight(): boolean
+isMovingLeft(): boolean
+isFalling(): boolean
+isBouncing(): boolean
+finishGravitationalState(): void
+getYVel(): double
+getXVel(): double
+reverseYVel(): void
+tick(): void
+setHealth(x:int):void
<<interface>>
Level
+getEntities(): List<Entity>
+getHeight(): double
+getWidth(): double
+tick(): void
+getFloorHeight(): double
+getHeroY(): double
+getHeroHealth(): int
+jump(): boolean
+moveLeft(): boolean
+moveRight(): boolean
+stopMoving(): boolean
+isFinished(): boolean
+setEntities(entities:List<Entity>): void
+setMovableEntities(movableEntities: List<MovableEntity>): void
+setEnemies(enemies: List<Enemy>): void
+setHero(hero:Controllable): void
+setFloorHeight(floorheight:double): void
+setLevelWidth(width:double): void
+setLevelHeight(height: double): void
+setGround(ground: Entity): void
+getScore(): int
+setTarget(target: long): void
+ getTarget(): long
+ setScore(temp: int): void
+setHeroHealth(x: int): void
+clone(): Object
<<interface>>
GameEngine
+getCurrentLevel(): Level
+startLevel(): void
+jump(): boolean
+moveLeft(): boolean
+moveRight(): boolean
+stopMoving(): boolean
+tick(): void
+resetCurrentLevel(): void
+isFinished(): boolean
+getDuration(): Duration
+gameOver(): boolean
+getLives(): int
+getcurrentLevelId(): int
+getScore(): int
+getTarget(): long
+setScore(x: int): void
+nextLevel(): void
+loadCurrentLevel(x: int): void
+createLevels(): void
Momento
- model: GameEngine
- level : Level
+saveLevel(model: GameEngine): void
+loadLevel(): void
JSONExtractor
- entities: List<Entity> = new ArrayList<>()
- movableEntities: List<MovableEntity> = new ArrayList<>()
- enemies : List<Enemy>= new ArrayList<>()
- hero: Controllable
-floorHeight: double
- levelWidth: double
- levelHeight: double
-obj: JSONObject
- target: long
- entityFactory: EntityFactoryInterface = new EntityFactory()
- movableFactory:MovableEntityFactoryInterface = new MovableEntityFactory()
- enemyFactory: EnemyFactoryInterface = new EnemyFactory()
+JSONExtractor(jsonPath: string)
+ getFloorHeight(): double
+ getLevelWidth(): double
+ getHero(): Controllable
+ getEntities(): List<Entity>
+getMovableEntities(): List<MovableEntity>
+getEnemies(): List<Enemy>
+ getTarget(): long
-extract(): void
- extractFloorHeight(): void
- extractLEvelWidth(): void
- extractLevelHeight(): void
-extractHero(): void
- extractImmovableEntities(): void
- extractMovableEntities(): void
- extractEnemies(): void
- extractTarget(): void
GameEngineImplementation
-height: double
-currentLevel: Level
-levels: Map<Integer, Level>
-levelId: int
-currentLevelId: int
-jsonPath: String
-start: Instant
- interval: Duration
- lives: int
+GameEngineImplementation(jsonPath: String, height:double)
+nextLevel(): void
+creatLevels(): void
^+startLevel(): void
^+getCurrentLevel(): Level
+loadCurrentLevel(x:int): void
^+jump(): boolean
^+moveLeft(): boolean
^+moveRight(): boolean
^+stopMoving(): boolean
^+tick(): void
^+resetCurrentLevel(): void
^+isFinished(): boolean
^+getDuration(): Duration
^+gameOver(): boolean
^+getLives(): int
+ getCurrentLevelId(): int
+getScore(): int
+setScore(x:int): void
+ nextJsonPath(): void
+getJson(): int
+nextLevelId(): void
EntityViewImpl
- entity: Entity
- delete: boolean = false
- node: ImageView
- imagePath: String
+EntityViewImpl(entity: Entity)
-getViewOrder(layer: Layer): double
^+ update(xViewportOffset: double): void
^+ matchesEntity(entity:Entity): boolean
^+markForDelete():void
^+getNode():Entity
^+isMarkedForDelete(): boolean
FinishFlag
+FinishFlag(imagePath: String, xPos:double, yPos: double, width: double, height:double)
^+isFinishFlag(): boolean
Collision
+aabbIntersect(entityA:Entity , entityB:Entity): boolean
+horizontalCollision(entityA:Entity, entityB:Entity, xVel: double): boolean
+verticalCollision(entityA:Entity, entityB:Entity, yVel: double): boolean
<<interface>>
EntityFactoryInterface
+makeEntity(type: String, xPos: double, yPos: double): Entity
<<interface>>
Enemy
+markForDeletion(): void
+isMarkedForDeletion(): boolean
+getXVel(): double
+moveHorizontally(hero: Controllable, leftCollision,: boolean , rightCollision: boolean): void
+animate(): void
+give_score(): int
<<interface>>
GravitationalState
+moveVertically(hero: controllable): void
+getForce(): Gravity
<<interface>>
LevelBuilderInterface
+buildHero():LevelBuilderInterface
+buildEntities():LevelBuilderInterface
+buildMovableEntities():LevelBuilderInterface
+buildFloorHeight():LevelBuilderInterface
+buildLevelWidth():LevelBuilderInterface
+buildLevelHeight():LevelBuilderInterface
+buildEnemies():LevelBuilderInterface
+buildGround():LevelBuilderInterface
+getLevel():Level
+buildTarget():LevelBuilderInterface
LevelDirector
-levelBuilder: LevelBuilderInterface
-level: Level
+LevelDirector(levelBuilder: LevelBuilderInterface)
+buildLevel(): void
+getLevel(): Level
Hero
-gravitationalState: GravitationalState
- fallingState: GravitationalState
-jumpingState: GravitationalState
-bouncingState: GravitationalState
-right: boolean
-left: boolean
-jump: boolean
-fall: boolean
-bounce: boolean
-hurt: boolean
-facingRight: boolean
-rightTicks: int = 0
-leftTicks: int = 0;
-notMovingTicks: int = 0
-immunityTicks: int = 0
-health: int = 3
+Hero(imagePath: String, xPos:double, yPos: double, width: double, height:double, xVel: double)
^+signalJump(): boolean
^+signalMoveLeft(): boolean
^+signalMoveRight(): boolean
^+signalStopMoving(): boolean
^+getHealth(): int
+setHealth(x:int): void
^+initialBounce():void
^isBouncing(): boolean
^+isJumping(): boolean
^+isMovingRight(): boolean
^+isMovingLeft(): boolean
^+isFalling(): boolean
^+finishGravitationalState(): void
^+moveVertically(): void
^+moveLeft(): void
^+moveRight(): void
^+getYVel(): double
^+getXVel(): double
^+reverseYVel(): void
-animate(): void
-setGravitationalState(gravitationalState: GravitationalState): void
^+tick(): void
^+takeDamage(): void
-handleImmunity(): void
MovableEntityFactory
+makeMovableEntity(type:string, xPos:double, yPos:double, xVel:double):MovableEntity
<<interface>>
MovableEntityFactoryInterface
+makeMovableEntity(type: String, xPos: double, yPos: double, xVel: double): MovableEntity
EntityFactory
+makeEntity(type:String, xPos:double, yPos:double): Entity
<<interface>>
EnemyFactoryInterface
+makeEnemy(type:String, xPos: double, spawnHeight: double): Enemy
<<interface>>
EnemyMovement
+move(enemy: Enemy, hero: Controllable, leftCollision : boolean, rightCollision:boolean): void
Gravity
-initialVelocity: double
-currentVelocity: double
-acceleration: double
+Gravity(initialVelocity: double, acceleration: double)
+getCurrentVelocity(): double
+next(): void
+reverseForce(): void
+reset(): void
<<interface>>
BackgroundDrawer
+draw(model:GameEngine, pane:Pane)
+update(xViewportOffset: double)
Runner
- model: GameEngine
- pane: Pane
- entityViews: List<EntityView>
- backgroundDrawer: BackgroundDrawer
-health: ImageView[] = new ImageView[3]
-lives: Text
- time: Text
- levelId: Text
- score: Text
- timeline: Timeline
- xViewportOffset: double = 0.0
- width: double
-height: double
- scoreCounter: int = 0
- healthcounterint = 0
- test: int = 1
+Runner (model: GameEngine ,pane: Pane, width: double, height: double)
+ run(): void
+ draw(): void
- nextLevel(): void
- drawScreen(message: String): void
-addHealth(): void
- drawLives(): void
- Level(): void
-Score(): void
- displayTime(): void
- displayScore(): void
- displayLevel(): void
-countTime(): boolean
IcePlatform
+ IcePlatform(imagePath: String, xPos:double, yPos: double, width: double, height:double)
^+isIcy(): boolean
EnemyAbstraction
-delete: boolean = false
#spawnHeight: double
#ticks: double = 0
# animationTimer: double = Math.random()+1
+EnemyAbstraction(imagePath: String, xPos:double, yPos: double, width: double, height:double, xVel: double, layer: Layer)
^+markForDeletion(): void
^+isMarkedForDeletion(): boolean
FallingState
-bouncing: Gravity
+FallingState()
^+moveVertically(hero: controllable): void
^+getForce(): Gravity
BouncingState
-bouncing: Gravity
+BouncingState()
^+moveVertically(hero: controllable): void
^+getForce(): Gravity
JumpingState
-gravity: Gravity
+JumpingState()
^+moveVertically(hero: controllable): void
^+getForce(): Gravity
LevelImplementation
-hero: Controllable
-entities: List<Entity>
- tangibles: List<Entity>
- movableEntities: List<MovableEntity>
-enemies: List<Enemy>
- floorHeight: double
- width: double
-height: double
- slideEffect: boolean
- slideVel: double
- finished: boolean
- score: int = 0
- target: long
+ LevelImplementation()
^+getEntities(): List<Entity>
^+getHeight(): double
^+getWidth(): double
^+tick(): void
^+getFloorHeight(): double
^+getHeroY(): double
^+getHeroHealth(): int
^+jump(): boolean
^+moveLeft(): boolean
^+moveRight(): boolean
^+stopMoving(): boolean
^+isFinished(): boolean
^+setEntities(entities:List<Entity>): void
^+setMovableEntities(movableEntities: List<MovableEntity>): void
^+setEnemies(enemies: List<Enemy>): void
^+setHero(hero:Controllable): void
^+setFloorHeight(floorheight:double): void
^+setLevelWidth(width:double): void
^+setLevelHeight(height: double): void
^+setGround(ground: Entity): void
+getScore(): int
+setTarget(target: long): void
+ getTarget(): long
+ setScore(temp: int): void
^+setHeroHealth(x: int): void
^+clone(): Object
- handleHeroVerticalMovement(): void
- handleHeroHorizontalMovement(): void
- handleEnemyMovement(): void
- handleMovableEntityMovement(): void
-handleEnemyAnimations(): void
-deleteEnemiesMarkedForDeletion(): void
LevelBuilder
-extractor: JSONExtractor
-level:Level
+LevelBuilder(jsonPath:String)
+buildTarget():LevelBuilderInterface
+buildHero(): LevelBuilderInterface
^+buildEntities():LevelBuilderInterface
^+buildMovableEntities():LevelBuilderInterface
^+buildFloorHeight():LevelBuilderInterface
^+buildLevelWidth():LevelBuilderInterface
^+buildLevelHeight():LevelBuilderInterface
^+buildEnemies():LevelBuilderInterface
^+buildGround():LevelBuilderInterface
^+getLevel():Level
ParallaxBackground
- width: double
- images: Image[]
- imageViews: ImageView[]
- parallaxEffect: double[]
+draw(model:GameEngine, pane:Pane)
+update(xViewportOffset: double)
EnemyFactory
+makeEnemy(type: string, xPos: double, spawnHeight: double): Enemy
Slime
-colour: String
- movementStrategy: EnemyMovement
-sleo_slime_score : int = 100
+Slime(imagePath: String, xPos: double, spawnHeight: double)
- adjustSlime(colour: String): void
^+moveHorizontally(hero: Controllable, leftCollision,: boolean , rightCollision: boolean): void
^+animate(): void
-manageTicks(): void
+give_score(): int
FollowHeroMovement
^+move(enemy: Enemy, hero: Controllable, leftCollision : boolean, rightCollision:boolean): void
RandomEnemyMovement
- ticks: double = 0
-still: double = 0.5
- phaseA : double = 0
-phaseB : double = 0
+move(enemy:Enemy , hero:Controllable, leftCollision: boolean, rightCollision:boolean)
-readyToMove(): boolean
-random(min:double, max: double): double
- manageTicks(): void
KeyboardInputHandler
-model: GameEngine
- left: boolean = false
- right: boolean = false
- pressedKeys: Set<KeyCode> = newHashSet<>()
- sounds: Map<String, MediaPlayer>= new HashMap<>()
- memento: Memento
+ KeyboardInputHandler(model: GameEngine)
+ handlePressed(keyEvent: KeyEvent): void
+ handleReleased(keyEvent: KeyEvent): void
GameWindow
- scene: Scene
- runner : Runner
-VIEWPORT_MARGIN: double = 280.0
-TIMER: int = 17
+ keyboardInputHandler: KeyboardInputHandler
+ model: GameEngine
+GameWindow(model: GameEngine, width: int, height: int)
+getScene(): Scene
+tickPerSecond(): int
+ getTimer(): int
+getViewportMargin(): double
+ run(): void
App
+main(args: String[])
^+start(primaryStage: Stage)

CODE REVIEW OF OWN CHANGES

The UML for all of the classes and interfaces I used to implement the new features



Level Transition

Since there is a total of three levels, I need to make sure when the hero reaches the finish flag in level 1 and level 2, hero must proceed to next level instead of showing ‘Winner’. Also, when hero loses lives and requires a restart at the starting line, the hero will return to the current level’s starting point. Whenever a hero loses all of his lives, ‘Game Over’ will be shown in the game window.
In order to perform this feature, I need to add functionalities in GameEngineImplementation.java, LevelImplementation.java and Runner.java. These are the key classes that control the level of the game. In fact, I need to create two more JSON files for the next two levels.

I used Interface Segregation Principle to achieve this feature.

In the GameEngine interface, I add two new methods called getCurrenLevelId() and nextLevel(). Since there is a hashmap in GameEngineImplementation.java to store the levels, I need to method to return an integer that tells me which level is showing now. Also, nextLevel() helps me to move the current level to next level. Since GameEngineImplementation.java do implemented the added methods and performed them well, it follows the Interface Segregation Principle.

I understand that only GameEngine object is called and if I only add functionalities in GameEngineImplementation.java, there will be errors shown. This leaves an impact that makes me be alert that I should create in the interface first at all time.

In Runner.java, I created displayLevel() method, to show the current level id in the game window.

There is Strategy Pattern in this. My nextLevel() is always called in the run time and it will always produce the expected output, unless the current level is three. However, in Runner.java's draw(), there is condition for the nextLevel() to perform. Ideally, it should not. This is because it violates the pattern and I should add the conditions in the nextLevel() instead. However, I feel that nextLevel() is effective here. This is because it do give me the expected output.

Although there is no issue, I feel that I can make improvements to make nextLevel() better. As mentioned earlier, if I added all of the expected conditions in my nextLevel(), and only let model.nextLevel() stays in the draw() of the Runner.java, it would be much better as this follows the Strategy Pattern.

Score

Eliminating a slime will gain one hundred points. Each level has its own target time, which is written in the respective JSON files. Before passing the target time, the hero will gain the target bonus (one second equals to one point) and lose one point per second after the game time passed the target time. To achieve this, I need to modify classes like LevelImplementation.java, GameEngineImplementation.java and Runner.java, as well as JSONExtractor.java

I used Single Responsibility Principle to achieve this feature.

Firstly, I added a target time in all the JSON file. Then I created a method buildTarget() in BuilderLevel.java which JSONExtractor.java extracts the target time and pass it to Level object through BuilderLevel.java. I also created setter and getter functions for the score for Level and GameEngine objects. Whenever a slime is marked for delete, the current score would be added one hundred. This is because I created a method in Slime.java called give_score() which will return a hundred integer to the Level object.

In addition, I created setter and getter functions for the target time for Level and GameEngine objects too.
In Runner.java, I created displayScore() method, to be shown in the game window of the current and total score.

This principle alerts me at all time which I must make sure all the methods that I created should be have a single responsibility. This is to ensure minimum coupling and it will not affect other methods or classes.

I used Strategy Pattern and I feel I have implemented it effectively. This is because in the Runner.java's draw(), there is no condition for displayScore() and it is continuously running in the run time. This is indeed a strategy skill and I believe I have implemented it correctly. It is also effective as the current and total score is displayed correctly in all circumstances.

No issue for this feature and I don't think there is any room for improvement.

Save & Load (did not demo)

Although I did not demo this feature to the tutor, I would wish to say my thoughts on this.

To me, when a user pressed a save key, the game is saved. When the user pressed a load key, the game is load, no matter where the hero is or what the level is. To achieve this, several classes are required. For example, KeyboadInputHandler.java, LevelImplementation.java, GameEngineImplementation.java and entity classes.

Dependency Principle could be used for this feature.

Firstly, I need to clone the current level and all of the entities. This can be achieved by implementing clone() from Clonable to Level and Entity objects. Although they shared same implemented method, the contents in the clone() are different. For Level object, new Level object is needed to copy the existing level. For Entity object, new Entity object is needed to copy the existing entity. When both of them are cloned and assigned together, GameEngineImplementation.java is required. Since there is a map of levels in the class, the cloned class can replace the respective level and load the cloned level to the game window.

Memento Pattern is needed for this feature because it clones the internal objects within the classes. A new class can be created to store cloned Level objects and Entity objects and pass to GameEngine object to achieve the feature. Hence, when a save or load key is pressed, there will be a condition in KeyboadInputHandler.java and will signal GameEngineImplementation.java to perform. Then, GameEngineImplementation.java will call Memento.java for save or load, and Memento.java will call Entity classes and Level class to clone themselves.

I did not really work on this because I had trouble in clone().