

# SOFT2201 Stage 3 Report

11.11.2019

---

Participant

Calvin (Kang Min) Seo 480347192

## Report Outline

<b>Report Outline</b>	<b>1</b>
<b>1. Stage 3 General implementation and Requirements</b>	<b>2</b>
<b>2. CODE REVIEW OF GIVEN CODEBASE</b>	<b>4</b>
2.1 OOP DESIGN PRINCIPLE ANALYSIS	5
2.1.1 SOLID Design Principles	5
2.1.2 Grasp Designing Object with Responsibilities	8
2.2 DESIGN PATTERN ANALYSIS	8
2.2.1 Creational Patterns	8
2.2.2 Behavioral Patterns	10
2.2.3 Structural Design Patterns	11
2.3 CODE STYLE & DOCUMENTATION ANALYSIS	11
2.4 OVERALL IMPACT ANALYSIS	14
<b>3.CODE REVIEW OF OWN CHANGES</b>	<b>15</b>
3.1 EXTENSION DESCRIPTION	16
3.1.1 LEVEL TRANSITION	16
3.1.2 SCORE	19
3.1.3 SAVE & LOAD	22
3.2 OO DESIGN PRINCIPLE FOR EXTENTION	24
3.2.1 LEVEL TRANSITION	24
3.2.2 SCORE	25
3.2.2 SAVE & LOAD	25
3.3 DESIGN PATTERN DOCUMENTATION	25
3.3.1 Memento Structural Pattern	26
3.4 UML Diagram for updated Code	26
3.5 DEVELOPMENT FOR FUTURE	27

Word count = 3678

## 1. Stage 3 General implementation and Requirements

For stage 3 of stickman I have received some additional features to add, by following OO design principles with Design Patterns. These additional features were;

- First, having a level transition from level 1 to level 3. When the level is done it should print out winner.
- Second, You have to keep track of the score when enemy is killed. It should also deduct the point after the target time has passed. Score is displayed for a different level as well.
- Lastly, you have to have a method that will save current level and the state and be able to load.

Here is the brief evidence of successfully implementing all three features.



*Figure 1.1.1 Displaying Current Level's Score*

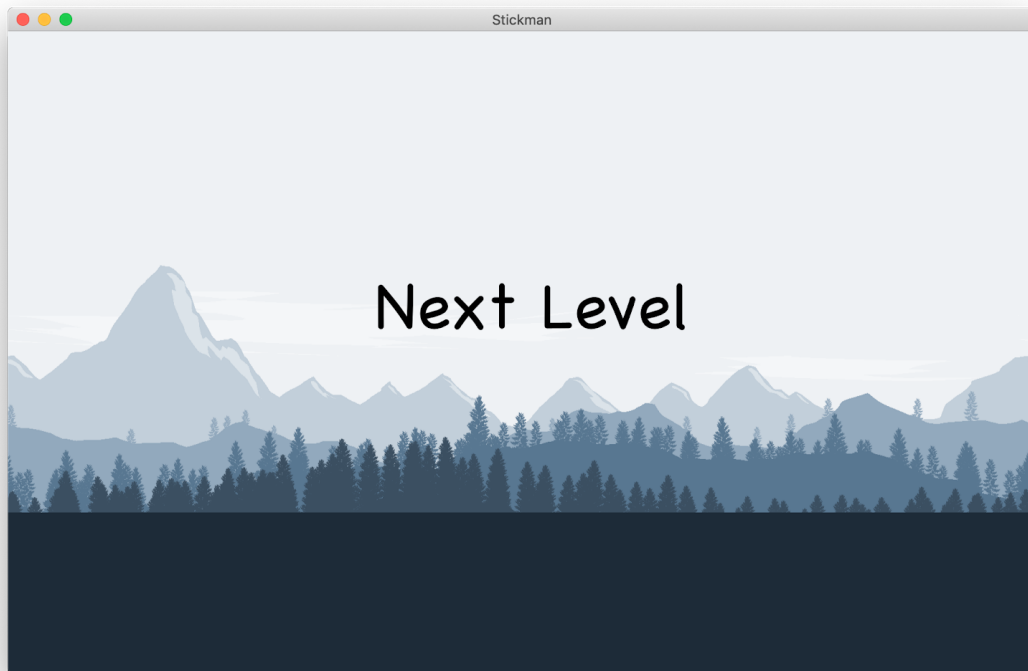


Figure 1.1.2 Moving To next Level



Figure 1.1.3 Displaying Score of the Current Level and Previous With Saving



## **2. CODE REVIEW OF GIVEN CODEBASE**

## 2.1 OOP DESIGN PRINCIPLE ANALYSIS

- CodeBase C has followed OO Design Principles with very little to none design Smell. Here is a detailed Analysis of the CodeBase following design principles.

### 2.1.1 SOLID Design Principles

#### 1. Single Responsibility

- Classes in the CodeBase C contained single responsibility. For example when the **Level** is configured from the **GameEngineImplementation** class it will first call the **LevelBuilderImplemntation** class that implements form **LevelBuilder** interface which will use **JsonExtractor** class the get each object information that is stored in the .json file. Then **LevelBuilderImplemntation** will be put into **LevelDirector** class as a parameter. LevelDirector is going to use buildLevel() method to direct **LevelBuilderImplementation** class to create a level and return the level to **LevelDirector**.
- Not only for the level, but CodeBase C has also followed this principal when configuring each entity, hero, enemy and etc.

```
/**
 * Creates the levels associated with the json file
 */
public void createLevels() {
    for(int i = 1 ; i <= maxLevelId ; i ++){
        JSONLevelString(levelId);
        LevelBuilder levelBuilder = new LevelBuilder(this.jsonPath);
        LevelDirector levelDirector = new LevelDirector(levelBuilder);
        levelDirector.buildLevel();
        this.levels.put(this.levelId, levelDirector.getLevel());
        score.initialize(levelId);
        levelId += 1;
    }
}
```

Figure 2.1.1 Constructing Level with Single ResponsiBilities

#### 2. Open/Closed

- CodeBase C will allow developers to put additional enemies to the code. Most of the moving objects, still object in CodeBase C will implement **Entity**

Interface. If you want to add additional feature that can implement either Entity with different abstraction. But since most of the class has implements Entity you can't modify **Entity** interface but to utilise its function.

```
public interface Entity extends Serializable {

    /**
     * Returns the image path of the entity
     * @return The image path of the entity
     */
    String getImagePath();

    /**
     * Sets the xPos of the entity
     * @param xPos The new xPos of the entity
     */
    void setXPos(double xPos);

    /**
     * Sets the yPos of the entity
     * @param yPos The new yPos of the entity
     */
    void setYPos(double yPos);

    /**
     * Returns the xPos of the entity
     * @return The xPos of the entity
     */
}
```

Figure 2.1.2 Entity Interface opening/closing functionalities

### 3. Sustainability

- The design of the CodeBaseC helped to increase the sustainability of the system. **MoveableEntity** interface will implement from the **Entity** interface. Inside the **LevelImplementation** Class, it has several Lists. It can be a list that contains *Entities*, *tangible* entities that will interact with hero, *Movable* Entities and *Enemies*. To each lists, since *Movable entity* also implements Entity you can put these types in to the list without causing a problem. The well used interfaces, abstract class of CodeBaseC will allow developers to add or delete different entities with different algorithm but it will still be able to articulate with the whole system.



```

public interface Controllable extends Entity {
    /**
     * Sends signal to the controllable entity to jump
     * @return True if the controllable entity can jump, else false
     */
    boolean signalJump();

    /**
     * Sends signal to the controllable entity to move left
     * @return True if the controllable entity can move left, else false
     */
    boolean signalMoveLeft();

    /**
     * Sends signal to the controllable entity to move right
     * @return True if the controllable entity can move right, else false
     */
    boolean signalMoveRight();

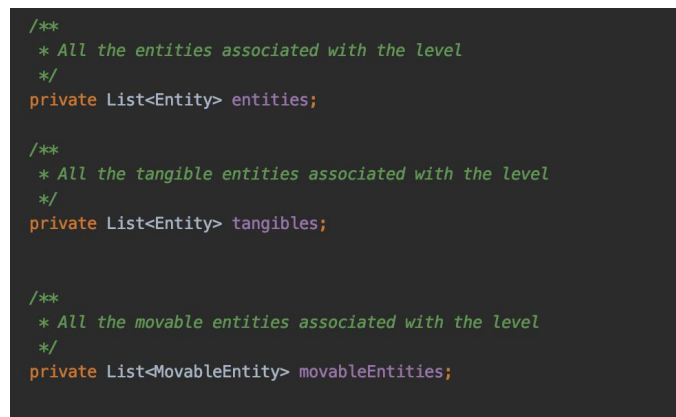
    /**
     * Sends signal to the controllable entity to stop moving
     */
}

public interface MovableEntity extends Entity {
    /**
     * Moves the movable entity horizontally
     */
    void moveHorizontally();

    /**
     * Returns the x velocity of the movable entity
     * @return The x velocity of the movable entity
     */
    double getXVel();
}

```

Figure 2.1.3 Interfaces extending Entity



```

/**
 * All the entities associated with the level
 */
private List<Entity> entities;

/**
 * All the tangible entities associated with the level
 */
private List<Entity> tangibles;

/**
 * All the movable entities associated with the level
 */
private List<MovableEntity> movableEntities;

```

Figure 2.1.4 LevelImplementation class's List

#### 4. Interface Segregation

- If you take a look at figure 2.1.3 and figure 2.1.4 you can see that **Controllable** interface and **MovableEntity** interface has been separated from the **Entity** interface. It is because if the functionality of two different interface was crammed into one single **Entity** interface, then the concrete classes such as that implements entity will have to write unnecessary methods that they do not utilize. Since it has separated it's function it will remove the design smell with clear Entities with different functionalities.

#### 5. Dependency Inversion

- Dependency Inversion did not happen for the CodeBaseC. It had classes that will focus on one functionality and this helped the system as a whole by keeping zero to none dependency for the higher class. The



**LevelImplementation** class will direct each entity to do its job but it won't be affected by the existence of each entity. It can construct the game with or without the entities.

### *2.1.2 Grasp Designing Object with Responsibilities*

When designing an OO project it is important to give appropriate amount of responsibility to classes. If you give too much or too little it will create a possibility of creating a design smell. **GameEngine** class should know it's level and create an instance of it. **GameEngine** will direct (do) each level with instruction to controllable object Hero. Just like this GameEngine will know and does it's job and function. This nicely designed principle was found throughout the code.

## **2.2 DESIGN PATTERN ANALYSIS**

There were multiple Design Patterns that was found in the CodeBaseC. You can classify it in terms of Creational Patterns, Behavior Patterns and Structural Patterns. For each pattern it is possible to find more than one but I will list the major design pattern that was used heavily.

### *2.2.1 Creational Patterns*

#### 1. Builder Pattern

When Creating an entity for the system, CodeBase C also followed builder pattern. It has separated the construction of a complex object (Level implementation) from its representation (types of entity; like hero, movable entities and etc). Each of these constructional happened in **LevelBuilder** class. When it extracts information from the .json file it will then construct using different concrete classes that ends with --Factory to create different entities. Then it will pass it's information to the **LevelImplementation** class to construct level inside the **GameEngine** class. (Please refer to the figure 2.2.1) Here the client would be **GameEngine** and a Director would be **LevelDirector** with a

concreteBuilder a **LevelBuilder** that uses **JSONExtractor**. And the result GameEngine wants is a constructed **Level**. (Please refer to Figure 2.1.1 and 2.2.1)

```
private void extractHero() {
    JSONObject heroObj = (JSONObject) this.obj.get("hero");
    double xPos = (double) heroObj.get("x");
    double xVel = (double) heroObj.get("xVelocity");
    double width = 30.0;
    double height = 51.0;
    String size = (String) heroObj.get("size");
    switch (size) {
        case "tiny":
            width *= 0.75;
            height *= 0.75;
            break;
        case "normal":
            width *= 1.0;
            height *= 1.0;
            break;
        case "large":
            width *= 1.25;
            height *= 1.25;
            break;
        case "giant":
            width *= 1.5;
            height *= 1.5;
            break;
    }
    this.hero = new Hero( imagePath: "ch_stand1.png", xPos, yPos: floorHeight - height, width, height, xVel);
    this.entities.add(this.hero);
}

/**
 * Extracts the immovable entities
 */
private void extractImmovableEntities() {
    JSONArray array = (JSONArray) this.obj.get("immovableEntities");
    for (Object o : array) {
        JSONObject entityObj = (JSONObject) o;
        String type = (String) entityObj.get("type");
        double xPos = (double) entityObj.get("x");
        double yPos = (double) entityObj.get("y");
        Entity entity = entityFactory.makeEntity(type, xPos, yPos);
        this.entities.add(entity);
    }
}
```

Figure 2.2.1 JSONExtractor class building different representation to construct complex LevelImplementation Class

## 2.2.2 Behavioral Patterns

### 1. Observer Pattern

The concrete class **Runner** which implements **BackGroundDrawer** interface will call it's draw method every 17 milliseconds. And the update should only happen every 17seconds not before or later. The draw method is going to make GameEngine (model) to tick which will update the current level. This has to use the observer behavioral design pattern to check on the state of each entity. If not it will update the xPos and YPos faster than the draw method which will make the system hard to keep track of each entities movement.

```
private void draw() {
    model.tick();
    if (model.isFinished()) {
        .....
    }
}
```

This is a code from the **Runner** class and when it calls the **GameEngineImplementation** to tick, it will call the tick method in it self which will make the current level to tick.

```
this.currentLevel.tick();
```

Inside the Current Level, the tick method looks like this.

```
@Override
public void tick() {
    hero.tick();
    handleHeroVerticalMovement();
    handleHeroHorizontalMovement();
    deleteEnemiesMarkedForDeletion();
    handleEnemyMovement();
    handleMovableEntityMovement();
    handleEnemyAnimations();
}
```

This will update the enemy according to the current state of enemy. It has solved the problem where the problem of to notify a varying list of object that the event (tick) has occurred and make the updates happen to each object in the list.

```

for (Entity entity: tangibles) {
    if (verticalCollision(hero, entity, hero.getYVel())) {
        if (entity.isFinishFlag()) {
            finished = true;
            return;
        } else if (hero.getYVel() > 0) {
            boolean conserveMomentum = ((hero.getYPos() + hero.getHeight() != entity.g
            hero.setYPos(entity.getYPos() - hero.getHeight());
            hero.finishGravitationalState();
            if (entity.isIcy()) {
                double slideAcc = 0.1;
                if (!slideEffect) {
                    if (hero.isMovingRight()) {
                        slideEffect = true;
                        if (conserveMomentum) {
                            slideVel = hero.getXVel();
                        } else {
                            slideVel = slideAcc;
                        }
                    } else if (hero.isMovingLeft()) {
                        slideEffect = true;
                        if (conserveMomentum) {
                            slideVel = -1 * hero.getXVel();

```

Figure 2.2.2 tick() in *LevelImplementation* interacting with “tangible” list and updating “tangible” it

### 2.2.3 Structural Design Patterns

#### 1. Adaptor Pattern

For the structural part of this system, it seems that it has used Adaptor pattern. Basic concept of the adaptor is that it converts the interface of another interface that clients expects. Here in the CodeBase C it has classified different types of Entity objects. Some of them were either movable, tangible, controllable and etc. As mentioned (please refer to Figure 2.1.3 and 2.1.4) in the Agenda 2.1 the use of OOP Design principles, keeping an interface organized like this will keep the system from creating a design smell.

## 2.3 CODE STYLE & DOCUMENTATION ANALYSIS

The code style and documentation of CodeBase C is very well designed with very little design smell. Since it has followed OO design principles, and implemented several Design patterns, it has achieved high cohesion (responsibility) and low coupling. To be specific, it has the following advantages.



### 1. It has no needless complexity

When writing an OO based design code, it is important for the developer to keep the class simple. CodeBase C utilized appropriate interfaces/abstract classes and different concrete class that implements different appropriate superclasses. Each class know and does it's job with purpose so it won't contain unnecessary complexity. (no Viscosity)

### 2. It has No Rigidity nor Fragility

Since it has followed Open/Closed principle with very nice sustainability form the SOLID design principle it will allow to system or user to add new objects and modify the code for additional requirements. This means that additional new objects won't break the system which also means that the code is not fragile.

### 3. Codebase C has no needless reputation

Since system has multiple interface that contains Adequate methods for each concrete type to use, each concrete class can decide which interface to implements. This will allow the concrete class not to repeat some needless reputation. (please refer to Figure 2.1.3 and Figure 2.1.4)

### 4. It clear and precise Documentation

Each Class, methods and instance in the CodeBase contains many comments with the purpose and the functionality of each object. When another developer modifies the code it is very helpful when the documentation is provided. It will allow developers to utilize it's method and understand class relationship which will reduce the time to understand the code. Take a look at the figure for one - by - one documentation for each part of the code.

```

/**
 * Represents the hero in the game
 */
public class Hero extends MovableEntityAbstraction implements Controllable {

    /**
     * Keeps track of the hero's current gravitational state
     */
    private GravitationalState gravitationalState;

    /**
     * The hero's falling state
     */
    private GravitationalState fallingState;

    /**
     * The hero's jumping state
     */
    private GravitationalState jumpingState;
}

```

Figure 2.3.1 Each instance declaration and class with Documentation

```

/**
 * Determines whether two entities are intersecting with each other
 * @param entityA The first entity
 * @param entityB The second entity
 * @return True if the entities intersect, else false
 */
public static boolean aabbintersect(Entity entityA, Entity entityB) {
    return (entityA.getXPos() < (entityB.getXPos() + entityB.getWidth()) &&
            (entityA.getXPos() + entityA.getWidth() > entityB.getXPos()) &&
            (entityA.getYPos() < (entityB.getYPos() + entityB.getHeight()) &&
            (entityA.getYPos() + entityA.getHeight() > entityB.getYPos()));
}

/**
 * Determines whether an entity is going to intersect with another
 * entity based on its current horizontal velocity
 * @param entityA The moving entity
 * @param entityB The entity that the moving entity may potentially collide with
 * @param xVel The current horizontal velocity of the moving entity
 * @return True if the moving entity is going to hit the other entity, else false
 */
public static boolean horizontalCollision(Entity entityA, Entity entityB, double xVel) {
    if (entityA == entityB) {
        return false;
    }
}

```

Figure 2.3.1 Each method with Documentation

## **2.4 OVERALL IMPACT ANALYSIS**

Implementing additional three features wasn't hard. Each of the classes implemented or inherited right interface and its superclass. Addition to all the documentation for the CodeBaseC it really helped me to figure out each class's functionality with which information it should contain. Since each class had adequate cohesion and adequate coupling it, it was easy to find a method which was required to modify, but without affecting the whole system. Just by looking at the class name and its documentation for more information I was able to figure out which design pattern it has used to construct objects, and control its behavior without causing major changes to the system. When the functionality seems not adequate to any existing classes I just had to add another ConcreteClass. For more detailed information please check for the Agenda 3.1



### **3.CODE REVIEW OF OWN CHANGES**



### 3.1 EXTENSION DESCRIPTION

This section records how I have implemented each requirement with explanation and screenshots of the code.

#### *3.1.1 LEVEL TRANSITION*

##### 1. GameEngine Modification

**GameEngine** contained information about different levels. Is had it create level function of creating different levels, with updating each entity by calling `currentLevel.tick()` method.

**GameEngine** was an adequate place to initialize different levels and update the `currentLevelID` instance the level is done where hero passes the finish flag.

```
public void JSONLevelString(int index){
    String level1 = "level_1.json";
    String level2 = "level_2.json";
    String level3 = "level_3.json";

    if(index ==1){
        this.jsonPath = level1;
    }else if(index ==2){
        this.jsonPath = level2;
    }else if(index ==3){
        this.jsonPath = level3;
    }
}
```

*Figure 3.1.1 Changing jsonPath in GameEngine based on the Index*

```

public void createLevels() {
    for(int i = 1 ; i <= maxLevelId ; i++){
        JSONLevelString(levelId);
        LevelBuilder levelBuilder = new LevelBuilder(this.jsonPath);
        LevelDirector levelDirector = new LevelDirector(levelBuilder);
        levelDirector.buildLevel();
        this.levels.put(this.levelId, levelDirector.getLevel());
        score.initialize(levelId);
        levelId += 1;
    }
}

```

Figure 3.1.2 Game Engine initialize each level to hashMap levels

## 2. Runner modification

The class that will direct model to change its level is **Runner** Class. Runner class will call draw() method where and it will check if the GameEngine is finished which will check it's current level to check if the hero has passed the finish flag.

```

if (model.isFinished()) {
    model.movingNext( b: true);
    if(model.hasNextLevel()){
        removeScore();
        t = drawScreen("Congratulations!\nYou Won in " + model.getDuration() + "s and\nhad "
            + model.getLives() + " lives remaining!");
        secondTimeLine();
        return;
    }else{
        t = drawScreen("You Are A Winner!\n Congratulations!\nYou Won in " + model.getDuration() + "s and\nhad "
            + model.getLives() + " lives remaining!");
        removeScore();
        thirdTimeLine();
        return;
    }
} else if(model.gameOver()) {
    removeScore();
    drawScreen("You lose!");
    return;
}

```

Figure 3.1.3 **Runner** Class Checking if the currentLevel has finished Through **GameEngine**

When the **GameEngine** returns true that the level has finished than it will draw on screen that current duration and it will stop the current timeline which was calling from run() method in runner class but change the timeline to SecondTimeline.

```

void secondTimeLine(){
    start = Instant.now();
    timeline = new Timeline(new KeyFrame(Duration.millis(17),
        t -> this.secondTimeLineDraw()));

    timeline.setCycleCount(Timeline.INDEFINITE);
    timeline.play();
}

on extends Object
ount, Comparable<Duration>, Serializable
    count = 500;
    interval = java.time.Duration.between(start, Instant.now());
    if((int) interval.getSeconds() == 3){
        pane.getChildren().remove(t);
        t = drawScreen("Your Score was: " + model.getCurrentScore(model.getCurrentLevelId()));
        timeline.play();

    }else if((int) interval.getSeconds() == 6){
        pane.getChildren().remove(t);
        t = drawScreen("Next Level");
        timeline.play();

    }else if((int) interval.getSeconds() == 8){
        pane.getChildren().remove(t);
        timeline.stop();
        run();
        model.movingNext(b: false);
        model.updateCurrentLevelId();
        model.startLevel();
        drawLives();
        addHealth();
        pane.getChildren().add(lives);
        return;
    }
}

```

Figure 3.1.4 SecondTimeLine and it's own draw() method

When the second timeline is called than it will have its own draw method to display it's current score and it's timeline. Before the timeline finishes, it will call model to update the currentLevelId which and start the level. However When the model does not have next level but finishes hero passes it's finish flag it will call the the third time line in the **Runner** class.

```

void thirdTimeLine(){
    start = Instant.now();
    timeline = new Timeline(new KeyFrame(Duration.millis(17),
        t -> this.thirdTimeLineDraw()));

    timeline.setCycleCount(Timeline.INDEFINITE);
    timeline.play();
}

void thirdTimeLineDraw(){
    pane.getChildren().remove(saveLoadText[0]);
    pane.getChildren().remove(LoadText[0]);
    tickCount = 300;
    interval = java.time.Duration.between(start, Instant.now());
    if((int) interval.getSeconds() == 3){
        pane.getChildren().remove(t);
        String s = endStringGen();
        t = drawScreen(s);
    }
}
}

```

Figure 3.1.5 ThirdTimeLine

When the draw() method in runner class finds out that there is no further level to transit, it will first give a message of saying Congratulation and you are a Winner. Then it will call the third timeline which will print out the recorded score for each level. However different from the second timeline it will not return back to the main timeline by calling run() in **Runner** class.

### 3.1.2 SCORE

#### - Score Class Creation

Score information is first initialized in the **GameEngine** when the level is created. Therefore when you create or add new level it will also initialize new level it will also initialize the score for that additional level as well. If you look at **GameEngine's** (Figure 3.1.2)

createLevels() method it will also give information of the level index which will pass it's information to the **Score** class. Score class is therefore dependent on the existence of the Level game Engine has.

```
/**
 * This class will keep of the score of the current number of level added to the game engine
 */
public class Score {

    /**
     * This score is the score when the hero kills an enemy
     */
    private static int KILL_ENEMY_SCORE = 100;

    /**
     * This is the map for the (Key)currentLevel ID and the (Value)score
     */
    private Map<Integer, Integer> score;

    /**
     * This will construct the score class
     */
    public Score() { score = new HashMap<>(); }

    /**
     * This initialize will happen in the GameEngine Class
     */
    public void initialize(int levelIndex) { score.put(levelIndex, 0); }

    /**
     * When this is called it will add point to the current level
     */
    public void enemyIsKilled(int levelIndex){
        int temp = score.get(levelIndex);
        score.put(levelIndex, temp + KILL_ENEMY_SCORE);
    }
}
```

Figure 3.1.6 ScoreClass Creation

Then from the GameEngine Score class will receive an information from the tick() method it has.

```
@Override
public void tick() {
    this.currentLevel.tick();
    interval = Duration.between(start, Instant.now());
    if(this.currentLevel.enemyDead()){
        score.enemyIsKilled(this.currentLevelId);
    }
}
```

Figure 3.1.7 GameEngine tick() updating Score class

When it calls the method that score class will update the current Levels score to its own hashmap containing levelId and score. Then the runner class will utilize GameEngines scoreMethods to display the current level and the previous level's scores to display on the screen.

**updateScore();**

updateScore() method is called inside the **Runner** class's draw() method.

```
private void updateScore(){
    this.pane.getChildren().remove(scoreText[0]);
    this.pane.getChildren().remove(scoreText[1]);
    this.pane.getChildren().remove(scoreText[2]);
    this.pane.getChildren().remove(scoreText[3]);
    < 11.0.3 >] javafx.scene.text
    public class Text extends Shape
    scoreText[i] = new Text( x: 10, y: 30, text: "Level" + (model.getCurrentLevelId() - i) + ": " + model.getCurrentScore( currentLevelId: m
    scoreText[i].setFont(Font.font ( family: "Chalkboard SE", FontPosture.ITALIC, size: 20));
    scoreText[i].setFill(Paint.valueOf("BLACK"));
    scoreText[i].setX(30);
    scoreText[i].setY(30 + ((20) * i));
    this.pane.getChildren().add(scoreText[i]);
}
scoreText[0] = new Text( x: 10, y: 30, text: "Level" + model.getCurrentLevelId() + ": " + model.getCurrentScore(model.getCurrentLevelId()));
scoreText[0].setFont(Font.font ( family: "Chalkboard SE", FontPosture.ITALIC, size: 20));
scoreText[0].setFill(Paint.valueOf("BLACK"));
scoreText[0].setX(30);
scoreText[0].setY(30);
this.pane.getChildren().add(scoreText[0]);
}
```

Figure 3.1.8 **Runner** class's updateScore() method

This will receive information from the GameEngine Class and draw currentScore and the previous scores if they exist.

Inside the .json file for each configuration of different levels, I have also added it's level's target time. This will be interpreted when the level is constructed. When the target time has passed while playing it will deduct 1 point every one second but not going to drop below 0.

```
public double getTargetTime(){return this.targetTime;}

/**
```

Figure 3.1.9 **JSONExtrator's** get target time method



```
{
  "_comment": "This is the level1 JSON file. You must use this file to save the level",
  "_stickmanSizeComment": "Your main character can be 100 pixels wide and 100 pixels high",
  "_targetfinishTiem" : "This is a time that user has to finish the level",
  "targetFinishTime" : 8.0,
}
```

Figure 3.1.10 .json configuration file that store each level's target time

### 3.1.3 SAVE & LOAD

When the user press **S** on the keyboard it will give its information to the **GameEngine** to save its current state. When you press **L** it will load if there is any saved level. Since **GameEngine** store it's levels information when you save a level it will deepCopy the whole Level into its local instance "savedLevel".

```
@Override
public void save(){
    if(!movingNext){
        saving = true;
        lastCalledInt = 1;
        if(savedStart == null){
            SavedInterval = Duration.between(start, Instant.now());
            currentLevel.saveTime((int)SavedInterval.toSeconds());
        }
        SavedInterval = Duration.between(savedStart, Instant.now());
        int a = currentLevel.getTime() + (int)SavedInterval.toSeconds();
        currentLevel.saveTime(a);
    }
    this.savedLevel = currentLevel.save();
    this.saveLevelId = currentLevelId;
    savedLives = lives;
    saveScore = score.getCurrentScore(saveLevelId);
    duplicateSavedLevel();
    savedLivesDup = lives;
    saveScoreDup = saveScore;
}
}
```

Figure 3.1.11 save() method in **GameEngine** is called from **KeyboardInputHandler** class

Then it will get the current level to give a copy of it self to **GameEngine**

```
@Override
public Level save() {
    savedLevel = (LevelImplementation) deepClone( object: this);
    return savedLevel;
}
```

Figure 3.1.12 save() method in *LevelImplementation* class which will return

When the user presses L it will retrieve the saved level and make it to the current level and load the level again.

```
@Override
public void load(){
    if(!movingNext){
        if(savedLevel == null && duplicateSaved !=null){
            savedLevel = duplicateSaved.save();
            savedLives = savedLivesDup;
            saveScore = saveScoreDup;
        }
        if(savedLevel != null){
            loading = true;
            lastCalledInt = 2;
            this.currentLevel = savedLevel;
            currentLevelId = saveLevelId;
            lives = savedLives;
            resetNextLevels(currentLevelId);
            levels.remove(saveLevelId);
            levels.put(saveLevelId, currentLevel);
            score.loadScore(saveLevelId, saveScore);
            if(addingTime > currentLevel.getTargetTime()){
                negativeScore = 1;
                tempNegativeScore = 0;
            }else{
                negativeScore = 0;
                tempNegativeScore = 0;
            }
            saveScore = 0;
            savedLives = 0;
            savedStart = Instant.now();
            startLevel();
            stopMoving();
            addingTime = getCurrentLevel().getTime();
        }
        savedLevel = null;
    }
}
```

Figure 3.1.13 loa() method in *GameEngin* is called from *KeyboardInputHandler* class



Then it will update the **GameEngine** other attributes such as the duration that the saved level has and every other information it contained. If the user loads previous level **GameEngine** will also reset next level and score and other instances if it exists.

### 3.2 OO DESIGN PRINCIPLE FOR EXTENTION

When I have added new features to the class I have documented every single Class, Instances, and method for further explanation in case next developer might be unsure of the methods functionality.

```
/**
 * Returns the current score of the current level to the runner class to display them on the screen
 */
@Override
public int getCurrentScore(int currentLevelId) { return score.getCurrentScore(currentLevelId); }

@Override
public int getCurrentLevelId() { return this.currentLevelId; }

/**
 * The level is going to be saved using this function includeing the information that was in the current gaming
 * state such as lives, duration.
 */
```

Figure 3.2.1 load() Example documentation from additional feature in **GameEngine** class.

I have also followed most of the coding style and also followed the design patterns that was used when creating different classes from the CodeBaseC. Since it was very well designed additional features have also followed an OO design Principles very well. (Achieving little to none opacity for the additional code)

Additional features were also added to the **ReadMe.Md** file so that the next developers is also familiar with it.

#### 3.2.1 LEVEL TRANSITION

From the Index 2 when analysing the CodeBase C, it was well designed with very little design smell. I have followed the same design when adding a new level where I have created different configuration for different levels. (Please refer to figure 3.1.1 and 3.1.2) already had information about the currentLevel so it was an appropriate class to configure

different levels. This means that GameEngine class has achieved higher Cohesion while maintaining the same to slightly higher coupling.

In addition, the transition of the level is not checked in the **GameEngine** itself but it will be directed from the **Runner** Class. Since **Runner** class is getting all the information about the current Level through the **GameEngine** class, it will also be capable of knowing if the level has finished. **Runner** class is also responsible for drawing the current level so it should also be responsible for displaying if the level has finished with the score and a message saying 'Next Level'. Runner class will call second Timeline which is totally separate from the main timeline and won't affect the timeline at all. From this Runner class have also achieved higher cohesion with very little increase in the coupling.

### *3.2.2 SCORE*

I have created a separate class for the score to be stored. I could have put an instance of the score in the **LevelImplementation** class but this would have made it's coupling to high with the GameEngine class. Since score class's hashmap is initialized based on the number of level GameEngine has, the system became strong with no fragility, giving Level less responsibility to focus an updating the entity.

### *3.2.2 SAVE & LOAD*

For saving and Loading it was tricky whether I should create a separate class to store the information of the level saved. But later I realized that this would be a needless class being created because level was able to copy itself where the **GameEngine** is just storing one extra Level information. This seemed to be more appropriate because GameEngine was responsible with the levels that was created.

## **3.3 DESIGN PATTERN DOCUMENTATION**

I have implemented all the design pattern that was previously used from the CodeBase C. But there was one other additional design pattern that I have used for the requirements for Save and Load function. Without mentioning the same design pattern that I have implemented for

additional features from (Please refer to Agenda 2.2 The use of Design Patterns) CodeBase C I will just mentioned the additional pattern. that I have added.

### 3.3.1 Memento Structural Pattern

Memento pattern is where user capture an object's state and make it be able to be loaded back to its current state.

```
if(keyEvent.getCode().equals(KeyCode.S)){
    model.save();
}
if(keyEvent.getCode() == KeyCode.L){
    model.load();
}
```

When the **KeyboardInputHandler** class gets input from the user, it will call the **GameEngine** to save and load the current level. Here **KeyboardInputHandler** will become an originator, where **GameEngine** becomes the Memento. Then it will get its current level to copy with exact state and return it self to **GameEngine**. Since **GameEngine** already had its level to load, it was an adequate place to put this design pattern as mentioned.

## 3.4 UML DIAGRAM

Here is the UML class diagram for the memento design pattern.

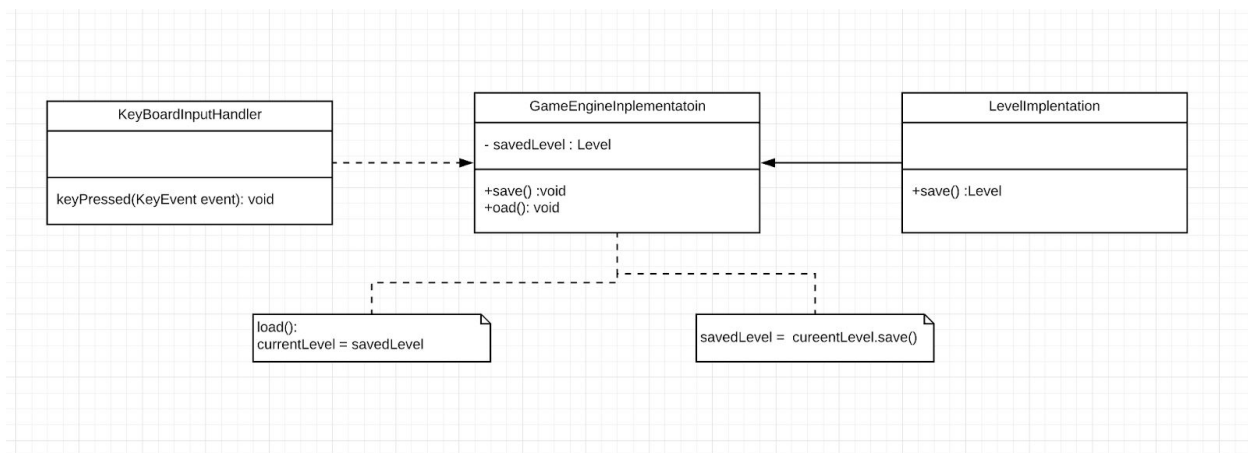



Figure 3.4.1 UML class diagram for additional design pattern

### 3.5 DEVELOPMENT FOR FUTURE

There were several Weakness to the additional code I have added. Transition to a different level, and keeping track of the score for each level worked very well and followed a design principle. But as always when designing a OO projects, it is hard to make the cohesion high while keeping a low coupling. For saving and loading feature GameEngine achieved higher cohesion where it understood and keep track of each Level's status. But it had also made the coupling higher. GameEngine had its function of handling Levels where it also interacted with the input that was received from the keyboard. It seemed for the time I was writing the code an adequate place to put its functionality due to a previous design. But next time when I have to redesign this I will try to divide it's save() and load() functionality. It is because saving a current Level's status has actually created some design smell.

```
@Override
public void load(){
    if(!movingNext){
        if(savedLevel == null && duplicateSaved !=null){
            savedLevel = duplicateSaved.save();
            savedLives = savedLivesDup;
            saveScore = saveScoreDup;
        }
        if(savedLevel != null){
            loading = true;
            lastCalledInt = 2;
            this.currentLevel = savedLevel;
            currentLevelId = saveLevelId;
            lives = savedLives;
            resetNextLevels(currentLevelId);
            levels.remove(saveLevelId);
            levels.put(saveLevelId, currentLevel);
            score.loadScore(saveLevelId,saveScore);
            if(addingTime > currentLevel.getTargetTime()){
                negativeScore = 1;
                tempNegativeScore = 0;
            }else{
                negativeScore = 0;
                tempNegativeScore = 0;
            }
            saveScore = 0;
            savedLives = 0;
            savedStart = Instant.now();
            startLevel();
            stopMoving();
            addingTime = getCurrentLevel().getTime();
        }
        savedLevel = null;
    }
}
```

Figure 3.5.1 load() Example documentation from additional feature in **GameEngine** class



When you look at the load method, GameEngine had too high coupling and it was very fragile too. If user want to add new status for the Level Implementation is should also keep track of it and update it inside GameEngine which means it is easy to break with difficulties to reuse it again.

To conclude, it was an honor for me to review the this well designed code and interpret with it. It gave me a deeper understanding of the design principles and understand design patterns better.