UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Object-Oriented Programming (SIN.06021)
Project Report: reflective work

Cedric Membrez, cedric.membrez@unifr.ch

Spring 2022

May 15, 2022

Contents

1 Introduction

In the project of this course, I implemented a Space Invaders[1] like game in Java. Below is an overview of the classes[2] used in the final version.

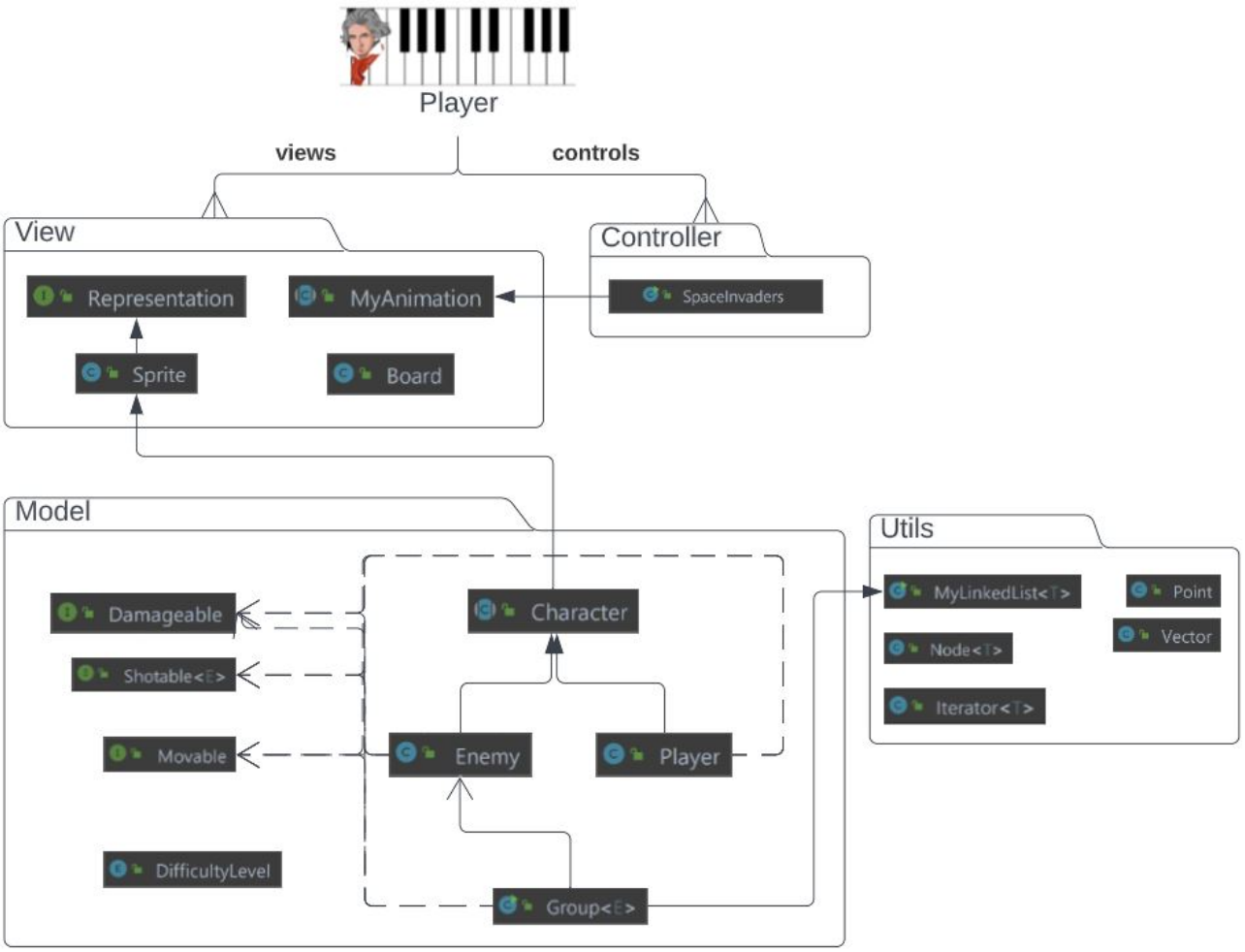


Figure 1: Overview of the Model-View-Controller with classes used in my final project. The player uses the Controller and sees the View. Dashed arrows are for "implements" and normal arrows are for "extends".

It uses the Model-View-Controller (MVC) pattern and has an extra package, *Utils*, for the basic data structures.

Comparing to the given instructions in the Series, the main differences to the are:

- the way *Enemy* and *Player* moves
- distinguishing between the *Projectile* of an *Enemy* and *Player*
- the *enum* for difficulty levels instead of directions

[1] Wikipedia page: https://en.wikipedia.org/wiki/Space_Invaders

[2] For visibility purposes, only the class names are used. The reader can find detail of Player and Enemy classes on next page, and other important classes in the Appendix A.

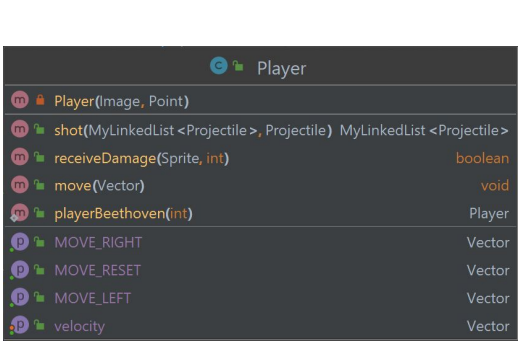
- a possibility for the player to restart or continue the game (i.e. the *MyAnimation* class has a new *reset()* method)
- the GUI is improved with, (1) an inner panel for health, score and number of enemies remaining, as well as a menu bar to choose the difficulty level, and (2) a menu bar.

Each point is detailed in the following sections.

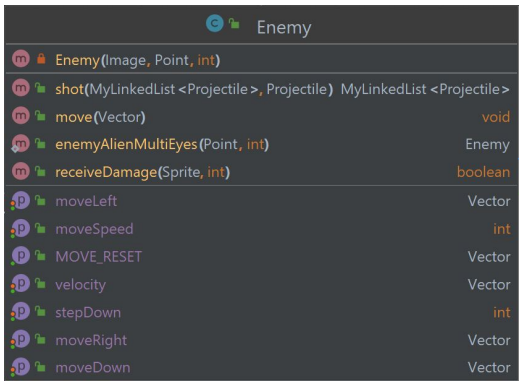
2 Movement Implementation

For some reason, regarding the movement of *Enemy* and *Player*, I did not exactly understood the *move()* and *enum Direction* implementations asked in the Series. That is, I am not using a *switch* to check each *enum* cases.

Consequently, I started by using *static Vector*s for each two classes. One variable for each possibly direction: left, right, down. But near the end of the project, I wanted the difficulty of the game to increase for the player. Because the game runs "continuously" on a single thread, and once my classes are instantiated, I cannot change my static classes anymore. Therefore, I turned these into instance variables. In effect, I am now able to adapt the speed of the enemies at run-time when a new game is restarted (cf. next section for details on restart).



(a) Player:



(b) Enemy:

Figure 2: Classes using the movement implementation

I believe the instance variables and the many properties add some confusion in my code, and might not be as elegant as it could be. Nonetheless, I am still able to use the *Group* and *MyLinkedList* classes to run the game, and overall, I would say it is a satisfying implementation for my project.

3 Projectiles

I created a class *Note* for the projectile of the *Player*, and *Flat* for the *Enemy*; the player being a pianist that shoot semi-quaver note, and the enemy shooting flat (an alteration in music to lower in pitch).

Initially, I had in mind that different behaviors could be implemented for how the player and enemy shot at each others. But in the end, I have an implementation of two projectiles which are relatively similar, not to say identical. The naming of the classes is also really confusing, "playerProjectile", or "enemyProjectile" would have make things clearer.

With time constraint, I decided to leave it as is and focus on being able to restart the game (cf. next section).

4 Game Manager

4.1 Restart

I wanted to give the player the possibility to continue after killing one set of enemies.

To the best of my understanding, the game runs on a single thread and to "restart" it, we would actually need to let this thread terminate thread gracefully and create a new one. Other methods seem to be undesirable, as mentionned in the Javadoc, and as per the *Thread.stop()* method being deprecated. I guess that I would have needed to call the *SpaceInvaders* class from a new one that manage the threads, an overkill I believe.

In the final implementation, I find an easier and simpler way. In *MyAnimation* class, I created a *reset* function which calls a modified function *init*: the later now takes a parameter *boolean restart* which is set to true when the player wants to restart or continue the game. For the player to make her choice, I created a *Board.setEndMessage()* method which is itself called by *SpaceInvaders.checkWinLoseStatus()*. In setEndMessage(), I make use of a *JOptionPane.showConfirmDialog* that pops-up based conditionally on boolean.

At the very first launch of *SpaceInvaders.java*'s main function, the *init(false)* is called and a *new Board* instance is created. If the player loses, we restart by using the same instance of the SpaceInvaders and Board, but clear all the and GUI messages (e.g. score). If the player wins and want to continue, we keep the score, reset the health to max, and increase the *int SpaceInvaders.gameLevel*: a variable that is increased by one each time a player kills all the enemies, and this variable will impact the horizontal and vertical speed of the enemies (i.e. see Enemy constructor).

As such, the player can continue his party with minimal changes in the code base, and can enjoy an increasing challenge. The side effect is that I start to have quite a few variables in my code, and I would have preferred to keep them (gameLevel, difficultyLevel, playerScore, ENEMY_PER_LINE, NUMBER_OF_LINE, POINT_PER_ENEMY_KILL, etc.) somewhere properly.

4.2 Difficulty Levels

In the final phase of fine tuning my game, I wanted to effectively use the *enum DifficultyLevel* that I implemented: BEGINNER, NORMAL, and EXPERT.

Currently, I believe the changes in speed both horizontal and vertical with respect to the different levels of difficulty and game levels are acceptable. Not perfectly smooth, but it gives a good challenge to the player.

5 GUI

My implementation of the class *Board* uses a BorderLayout and an inner panel called *LocalStatisticPanel*. It displays with JLabel the remaining life of the player, her current score, and number of enemies remaining.

In addition, my abstract class *MyAnimation* uses a JMenuBar which is effectively set in the *SpaceInvaders* class. I created an inner class *MenuListener* which iterates through the Enum *DifficultyLevel* to create each JMenuItem.

It made me realize that an Enum is more than just some constant String. I made use of their ordinal values as well to adapt the difficult level and impact my enemies' speed.

6 Conclusion

To conclude this report[3], I realized there are still glitches to fix. Among others, I can think of:

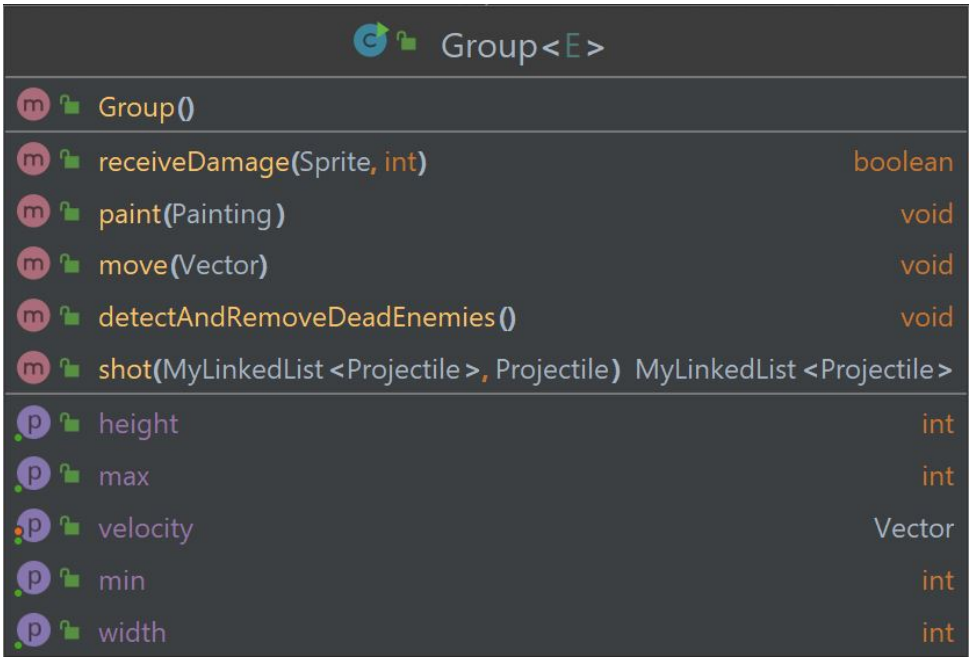
- End game: the last enemy Sprite is visible, and count is not set to zero on the GUI.
- Pop-up messages: unfortunately listen to the keyboard and the SPACE bar will confirm the Yes button which create undesirable effect. But I could not find a fix to remove the listener of a JOptionPane (despite several methods available).
- Collision: once a shot of the enemy is below the player, the player can still get hit from unreasonable distance (i.e., there is a clear gap on screen, but Sprites collide nonetheless) despite reformatting the Sprite as best as possible to avoid transparent surrounding/border.
- Enemy hit and life: the player should need to hit an enemy twice to kill it, but at the moment (after re-implementing Group and MyLinkedList), a glitch exists because only one hit is required.

I believe these are minor glitches part of a continuous improvement of a game. Also, I would have enjoyed implemented a few other improvements (e.g. keeping track of high scores, adding some protection for the player that can be destroyed by the enemy, etc.) but exams period is approaching way too fast. Overall, I enjoyed this project very much and it was fun to implement the theoretical concepts in this way (SpongeBob was right!). Finally, I give you my thanks for the continuous comments and help during the Series!

[3]Overleaf word count: 1183.

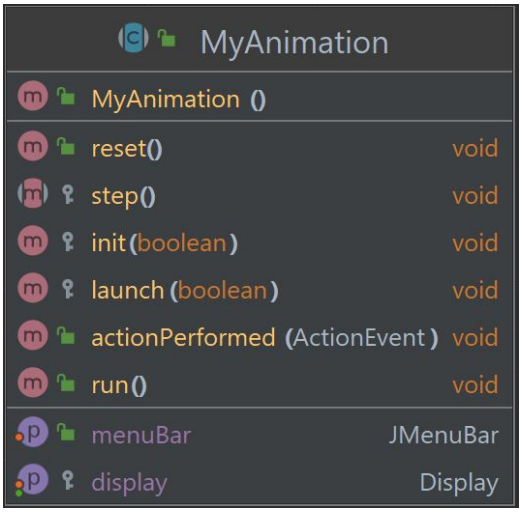
A Appendix

(a) Character abstract class

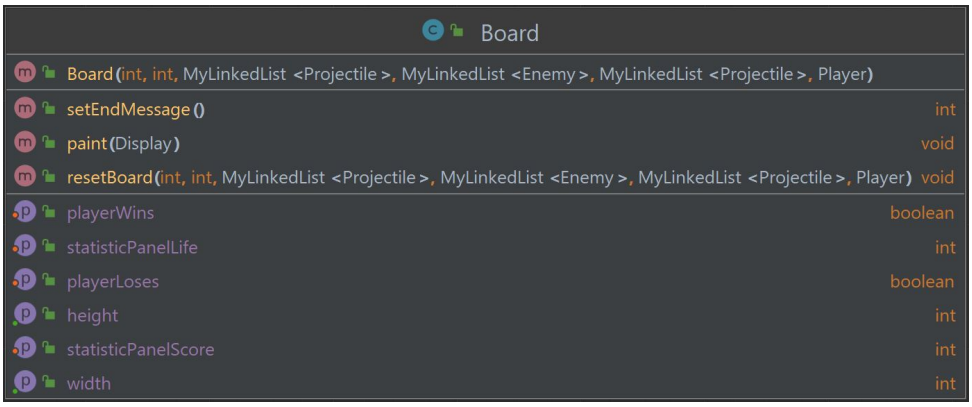


(b) Group concrete class

Figure 3: Model Classes



(a) MyAnimation abstract class



(b) Board concrete class

Figure 4: View Classes