Department of Informatics

Professor: R. Ingold
Teaching Assistants: M. Alberti, L. Vögtlin
Assistant: R. Geissbühler

S2019 Object Oriented Programming

Assignment 07

1 Theory

In this exercise you have to answer the following theoretical questions. Keep your answers short and precise.

(a) What are the different strategies to handle the position of graphic elements on the screen? Which one is the best and why?

(b) What is the purpose of a layout manager?

(c) Mention 3 default layout manager implemented in AWT/Swing.

(d) Is it possible to implement your own layout manager ? If yes, when would you do that? If not, why?

(e) Which methods should and graphic element implement for AWT and Swing to display it properly?

(f) How can one refresh the view after modifying some elements in AWT/Swing? How is this different in JavaFX?

(g) When should one use `java.awt.Graphics2D` instead of the regular `java.awt.Graphics`?

(h) What informations are held inside an event?

(i) Can a single event be treated by multiple listeners? Why?

(j) What should one pay attention to when using `getX()`/`getY()` from `MouseEvent`?

(k) What is the difference between `keyPressed` and `keyTyped`?

(l) What is the focus in the context of Java GUI and why is it important not to forget about it while developing an interface?

2 Implementation

For this exercise you are required to submit one file: MouseEventDemo.java, implemented as described below.

(a) Implement a `MouseEventDemo` class which implements `MouseListener` and extends `JPanel`. This class has to show the user a GUI *similar* to the one shown in Figure 1 where at the top of the window is a blank area (in the image is painted blue) and each time a mouse event occurs (mouse enters the area, clicks, leaves, ...) a descriptive message is displayed under the blank area. Be creative!

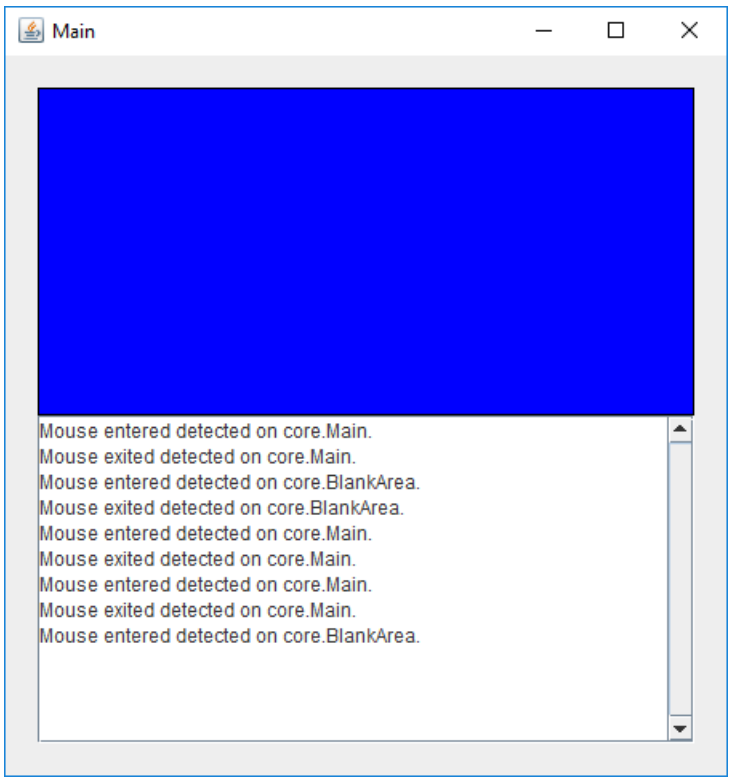


Figure 1: Example of the GUI for the implementation exercise after a couple of mouse interaction have been done.

3 Debugging

In this exercise we want to practice the use of the debugger and familiarize with the error messages of the compiler. Refer to the files `KeyEventDemo.java` attached to this series for this exercise.

(a) Take a look at the code and try to understand what it does. Eventually, run it and find out how it works.

(b) This small snippet is supposed to show the keyboard events (is a very similar fashion of the implementation exercises for the mouse). Notice that printing regular character works fine (see Figure 2a).

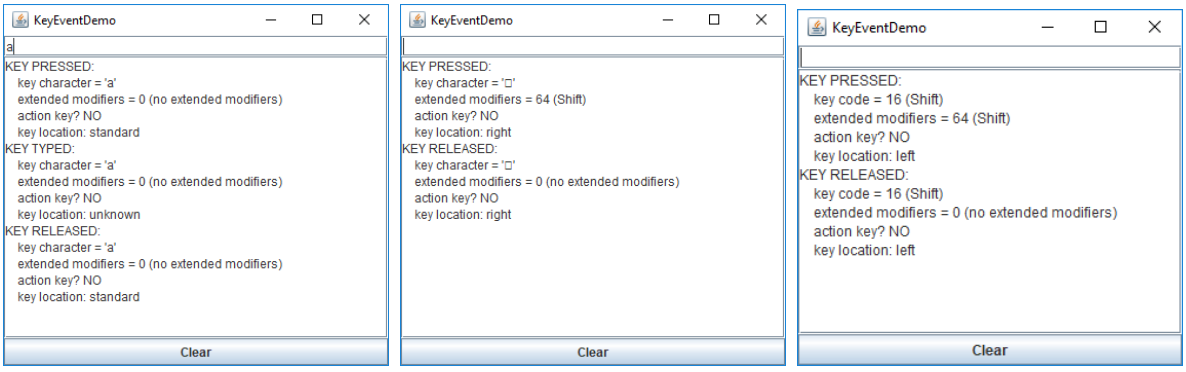
(c) Now clear the area and try pressing the key `SHIFT`. Notice that is does not work properly as they output is a weird character (see Figure 2b) instead of the proper name (see Figure 2c).

(d) What is this happening? Why ? How could you fix it?

(e) Go with the debugger and inspect the value of `keyString` inside `displayInfo()` when you are pressing the `SHIFT` button. Is it fine? If not, what is wrong with it?

(f) Try to fix the problem on your own.

(g) If you have no clue how to fix it, here:
Hint 1: You should only rely on the key char if the event is a key typed event.
Hint 2: Call `e.getKeyChar()` *only when the event is of* `KeyEvent.KEY_TYPED` *type.*
Hint 3: Solution lies at `https://docs.oracle.com/javase/tutorial/uiswing/events/keylistener.html`. *Copying it is not going help you learn, understand what is going on and why if you want to learn something.*



(a) Correct output for "a" (b) Wrong output for "SHIFT" (c) Correct output for "SHIFT"

Figure 2: Examples of output for this exercise GUI.

4 Bonus exercise

For this exercise you are required to submit as many files as needed, implemented as described below.

(a) Implement the application presented in the debugging exercise making use of JavaFX instead of Swing/AWT.

(b) Make sure you used no Swing/AWT components by checking the imports in the upper part of your files. You should not read anything related to them there but only JavaFX related imports.

(c) Now that you have the same software written in two versions: which one do you prefer and why? Elaborate.

5 Project

This week we will recap what has been done so far and the introduced the next step of the project, which is moving the enemies with a timer.

5.1 What You Should Have So Far

The design part, i.e. taking the decision of which class to implement and how, is the hard part of programming; anyone can blindly follow a list of steps to reach a certain goal like a cooking recipe. Getting to build a nice looking project without understanding the underlining design choices and the reasoning behind them is not so helpful from the learning point of view. This is why in the last weeks the guidelines were unconstrained (by choice). However, we got the feedback that some of you are a bit lost in the design part. Thus, this Section will recap the status of the project in order to help clarify certain aspects.

In figure 3 there is the UML diagram (see last weeks series for details about UML diagrams) of our implementation at this stage. Keep in mind we are in the MVC[1] architecture paradigm: model, view and controller. The first observation that one can do is that there are clearly 4 groups of classes clustered together. Three of them are the model, view, controller and the fort one are support classes which does not really belong anywhere in the pattern.

5.1.1 The Model

In the top left corner of figure 3 there are the classes that represent the **model** of our project. All these classes are described in a more detailed UML diagram in figure 4 where also the methods are shown. In general, the role of these model classes is indeed, as the name says, to model the project. Concretely, they are in charge of describing *what* is in the project and how it is defined. Therefore, in our Space Invades scenario, we'd expect to find all the classes relative to the player, the enemies, the shots and virtually anything which can be visible on screen (you might have some other components such as obstacles and the like). Let's have a deeper look:

- `Sprite`: this class implemented the basic representation of an element which can be painted to the canvas of the game. Other elements will extend from this (it is abstract!), because we want to leverage polymorphism and reduce the amount of code we write. In fact, things like the x,y coordinates and the images that represent the entity are common elements and thus we can put them here without having every sub-classes to re-implement them. Since this class is, de facto, the representation of an entity, it implements `Representation`.
- `Representation`: this is a very simple interface which is in charge of exposing an API of something which can be represented. In our scenario it is only Sprite but its good practice to have a solid and robust class hierarchy s.t. if we were to extend the project afterwards we would not have to re-factor a lot. Concretely, this interface is implementing `Paintable` interface, and exposes the method `getBoundingBox()`.

[1] https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

This method is abstract in this class, and its implementations is responsible of returning the cooridnates of a the bonding box around the entity we are painting. This bounding box will be used for the collision computation.

- `Paintable`: this is a `oop.lib` class which you receive and its core function is to expose the method `Paint()`.

- `Shot`: this class has the same nature of `Sprite` by being the base - but abstract - implementation of the shots entities we will have in our game. In our project we have implemented only two concrete versions of shots, namely `Bomb` and `Bullet`. As you can see, this class is responsible for holding the information about the direction and the amount of damage a shot does.

- `Bomb`: this class is the concrete implementation of a `Shot` and we use it to represent what the aliens shoot at the player (so top-down direction).

- `Bullet`: this class is the concrete implementation of a `Shot` and we use it to represent what the player shoot at the aliens (so bottom-up direction).

- `Enemy`: Once again, we get the same design choice of abstract class exposing base functionality that we don't want to repeat on the top and all other concrete implementations extending from it. In this case we have `Enemy` holding information about how fast is the descent, how many health points and a random generator to determine whether or not they shoot towards the player.

- `AlienShip`: in the same way that `Bomb` and `Bullet` were the concrete implementations of `Shot`, `AlienShip` is the concrete implementation of `Enemy`. Here new information has to be held regarding this specific enemy, which is not shared with other implementations. These include the initial x,y coordinates and the firing options.

- `Player`: as we would expect, this class extends `Sprite` and bring some more functionally peculiar to its function. Specifically, it keep tracks of its own health points and exposes methods to interact with it such as `gotHit()` (which should lower the health points) and `move()` which enables the Controller classes to interact with its position.

- `Attacker`: this interface core functionality is to expose the method `attack()`. Intuitively, any entity which want to perfrom an attack in our game (so the player and all enemies concrete implementations) needs to implement this interface. Here you can see the potential of the abstract class as `Enemy` *does not* implement the method from `Attacker`. Thus, classes who extends from it will be responsible to do so if they wish to be concrete (e.g. `AlienShip` does).

- `Hittable`: this interface is designed in the same fashion as `Attacker`, by expose methods which enables handling entities who can be hit i.e. `gotHit()` and `isDead()`.

- `Moveable`: same thing again, this time for objects who needs to be moved around.

5.1.2 The View

In the top bottom middle part of figure 3 there are the classes that represent the **view** of our project. All these classes are described in a more detailed UML diagram in figure 5 where also the methods are shown. In general, the role of these view classes is indeed, as the name says, to visualize the project. Concretely, they are in charge of handling the process of *show* what is in the project and how it looks. Therefore, in our Space Invades scenario, we'd expect to find all the classes relative to the panting routine and the graphics components. Let's have a deeper look:

- `Board`: this class is the container where all the **model** classes are referenced such that they can be painted whenever **controller** classes require to do so. In fact, this is the "artist" in our project which paints all the components on the canvas. As you can see, it has the references to `enemies`, `shots` and `player` because it needs to know what to paint and where. All these operations are - for now - taken care by the provided class `Display` which board extends from.

- `Display`: this is a provided class which implements the concrete method that actually paint the elements on the canvas. It is responsible of handling the graphics components at the Swing level.

- `Painting`: this is a provided interface which exposes some method which a class in charge of painting should have. When in the next weeks you will be required to implement your own version of `Display` you will need to implement this interface and fulfill all the abstract methods it exposes. Thanks to this interfaces, the functionality of `Board` will be unaffected by the swapping of the two different versions of `Display`.

- `JPanel`: this is a base Swing Java class and we just make use of it in the `Display` class to have access to a canvas to paint on.

5.1.3 The Controller

In the top bottom left part of figure 3 there are the classes that represent the **controller** of our project. All these classes are described in a more detailed UML diagram in figure 6 where also the methods are shown. In general, the role of these controller classes is indeed, as the name says, to control the project. Concretely, they are in charge of handling the process of *executing* the project. Therefore, in our Space Invades scenario, we'd expect to find the main class which starts the whole thing, and the classes relative to clock (or timer) of the game which takes care of the time flowing. Let's have a deeper look:

- `SpaceInvaders`: this is the main class which is responsible to handling the routine of the game. Specifically it will have the game loop.

- `MyAnmination`: the description of this class is provided in Section 5.3 "The Clock is Ticking!" below

- `Runnable`: this is the base interface to run another thread in the application and has been covered extensively in the course slides.

- `ActionListener`: this is the base interface to handle user inputs and has been covered extensively in the course slides.

5.1.4 The Utils

In the top bottom right part of figure 3 there are the classes that represent the **utils** of our project. All these classes are described in a more detailed UML diagram in figure 7 where also the methods are shown. These classes do now have a specific role in the project and are mostly a support class. In our Space Invades scenario, we have the list for our entities. These classes are not explained in details as they are self-documented and/or parts of previous series (or the current one, for `IntergerList`. Their core functionality is handling collection of elements that we have in our projects.

5.2 Moving The Enemies

Now we will move into the next segment of the project development. This week we will be implementing the moving operation for the enemies and the game clock. In this section we describe the moving enemies part. The enemies move in a predefined pattern in Space Invaders. They start at the left border and move to the right. When the most right enemy touches the right border all enemies move down and then start heading to the left. If they hit the left border they do the same. Like this the enemies are closing the gap between them and the player. In figure 8 you can see the described behaviour.

5.2.1 Implement a IntegerList

To move the enemies we need a structure to hold a list of integers. As a template you can take either `ShotList` or `EnemyList`. Change the name of the class (also the file) to `IntegerList` and adapt it as needed to accept `int`.
Additionally we need to be able go the the maximum and the minimum from this list. So you need to implement two functions:

getMax() When this function is called you have to iterate over all elements in the array (until size) and return the maximum of it.

getMin() When this function is called you have to iterate over all elements in the array (until size) and return the minimum of it.

5.2.2 Implement the Moving Operation

As your enemies can already move we just need to implement the pattern as shown in figure 8. To achieve this you have to add a new private method in the `SpaceInvaders` class called `moveEnemies()`. The body of the method is described in the listing 1 (this is psydocode). To keep track of the direction create a new field of type `String` called `direction`.

Listing 1: Psydocode of the enemy movement

```
// Get X coordinates of all enemies
IntegerList xCoordinates
foreach enemy in enemies:
    xCoordinates.append(enemy.getX())

// Step for the enemies
foreach enemy in enemies:
    enemy.move(direction)

// If the border is met, everybody goes down and direction is changed
if xCoordinates.getMax() >= BOARD_WIDTH - enemies.get(0).getWidth():
    direction = "LEFT";
    foreach enemy in enemies:
        enemy.move("DOWN");

else if xCoordinates.getMin() <= 0:
    direction = "RIGHT";
    foreach enemy in enemies:
        enemy.move("DOWN");
```

Do not forget to call this function in the step method.

5.3 The Clock is Ticking!

Every game need a game loop which is responsible for refreshing the board after a certain amount of time. Until today the abstract class `Animation` of the `oop.lib` library provided you with this functionality. This week you will write your own animation class. Follow the steps below to create your ofn animation class.

1. Create a new class called `MyAnimation` which implements `Runnable` and `ActionsListener`.
2. We need four private fields:
   (a) JFrame frame
   (b) Timer timer (javax.swing)
   (c) boolean autoplay
   (d) Display display
3. Override the run method from the `Runnable` interface. In there create a new JFrame and give it a title and set the default close operation.
4. Create a new `Timer` with the parameters 20 and this and assign it to the timer variable. The timer will then every 20ms create an `ActionEvent` which we have to take care of.
5. Call the `init()` method.
6. Add the display to the frame and pack the frame.
7. Call frame.setLocationRelativeTo(null), display.requestFocus() and frame.setVisible(true).
8. Now start the timer with the method `.start()` and set the inital delay to 20ms if `autoplay` is true.

9. Override the actionPerfirmed method and call in the the method `step()` and `display.repaint()`.

10. Create two empty procedure called `init()` and `step()` which are protected.

11. To launch our application create the method launch with the following header `protected synchronized void launch(boolean automatic)`. In the body assign the value of automatic to the variable autoplay and then call `SwingUtilities.invokeLater(this);`

12. As last add a setter for the field display

Now you have to change the class `SpaceInvaders` is extending to `MyAnimation`. Your game should now run over your own animator class.

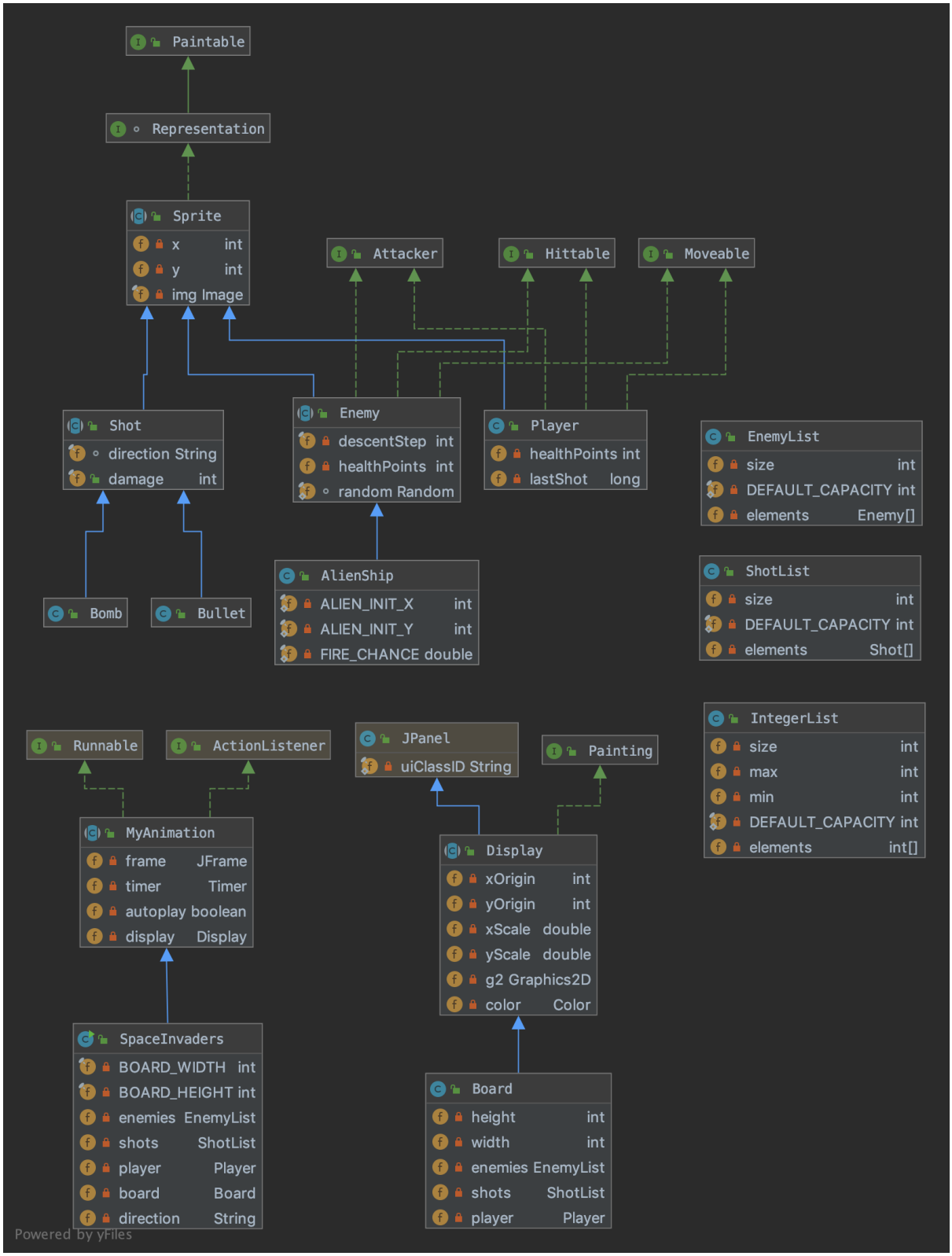


Figure 3: UML Diagram of the project structure.

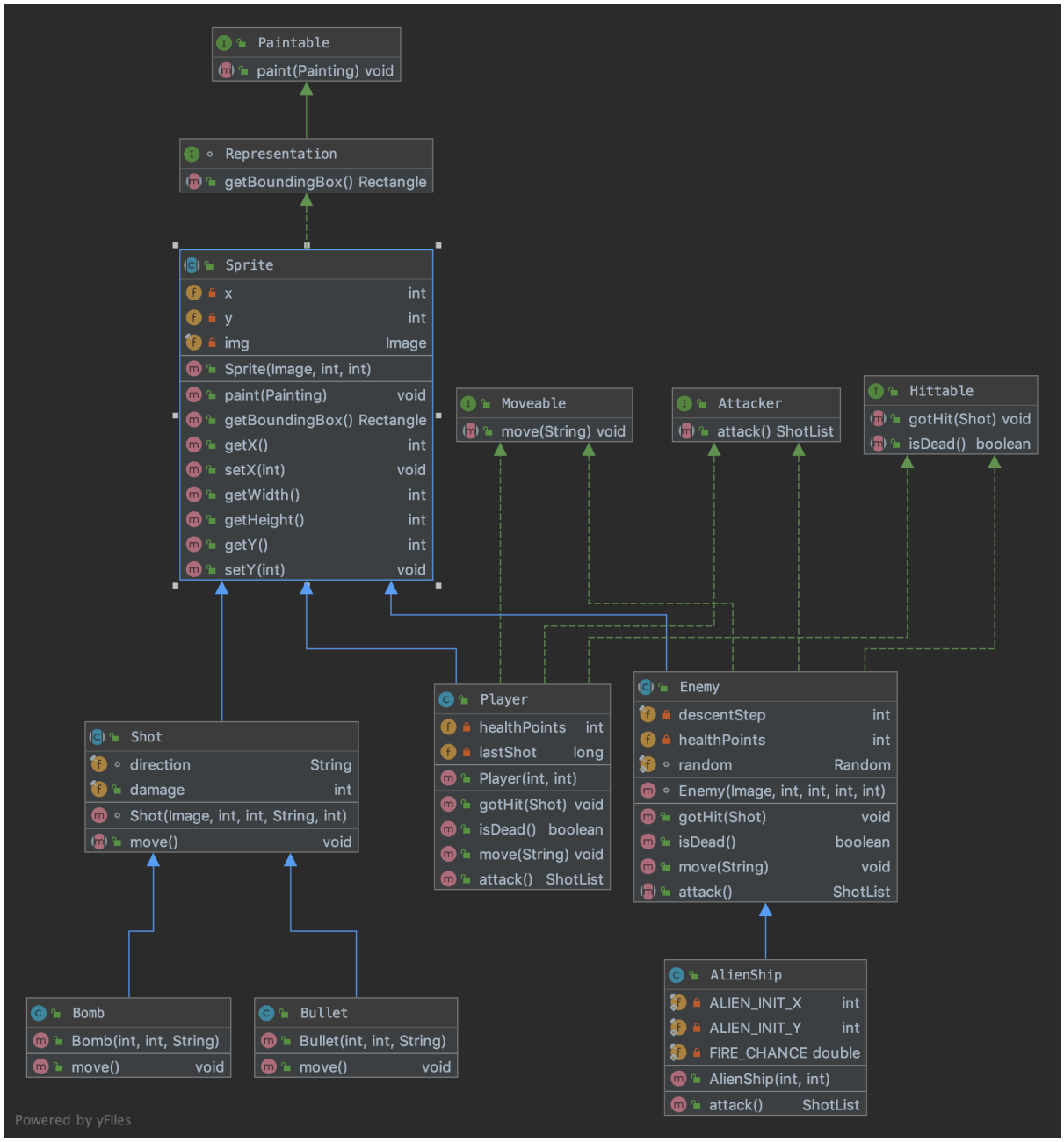


Figure 4: UML Diagram of the model part of the project.

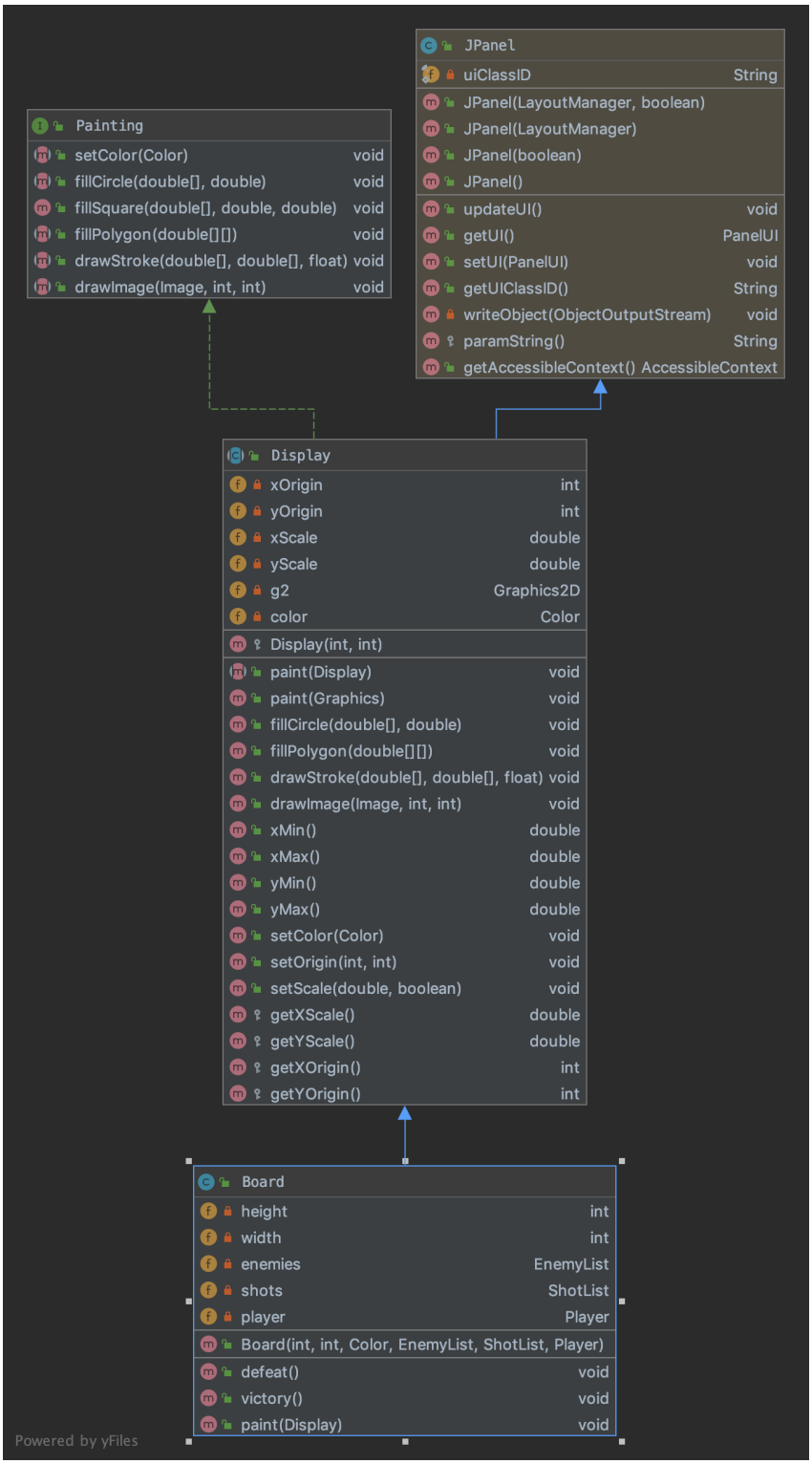


Figure 5: UML Diagram of the view part of the project.

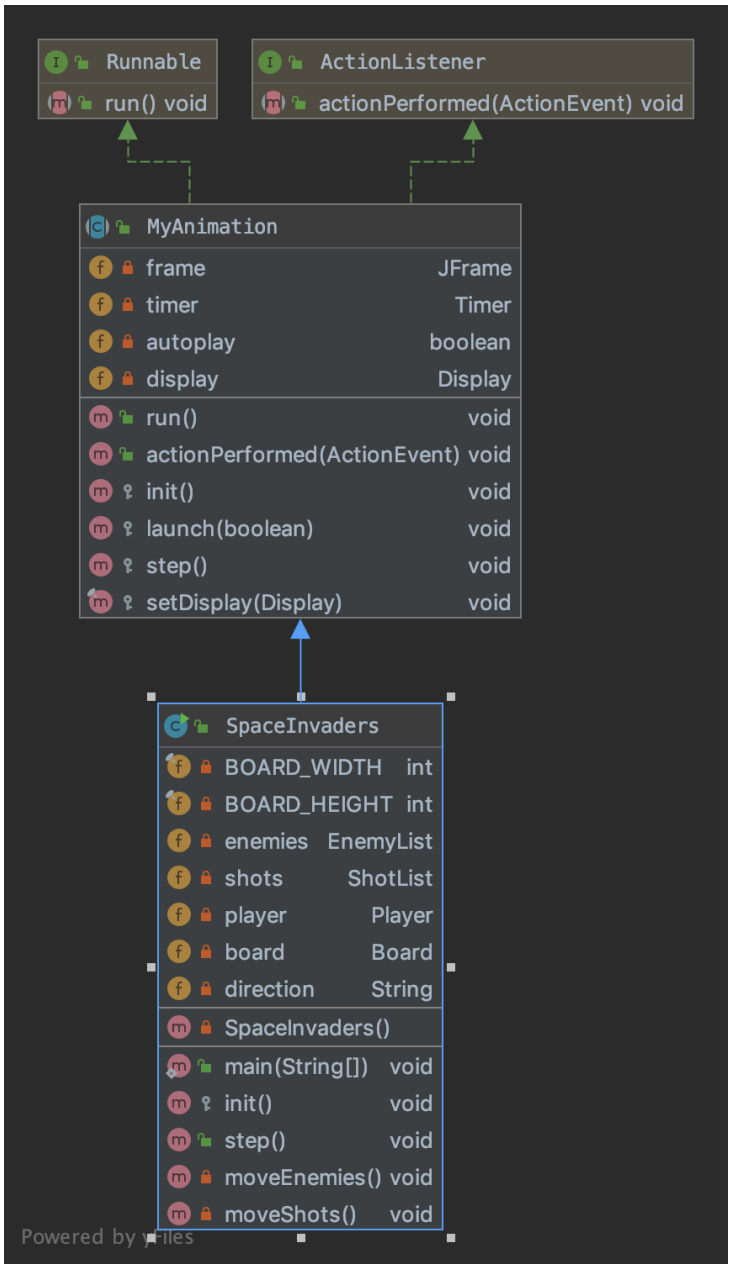


Figure 6: UML Diagram of the controller part of the project.

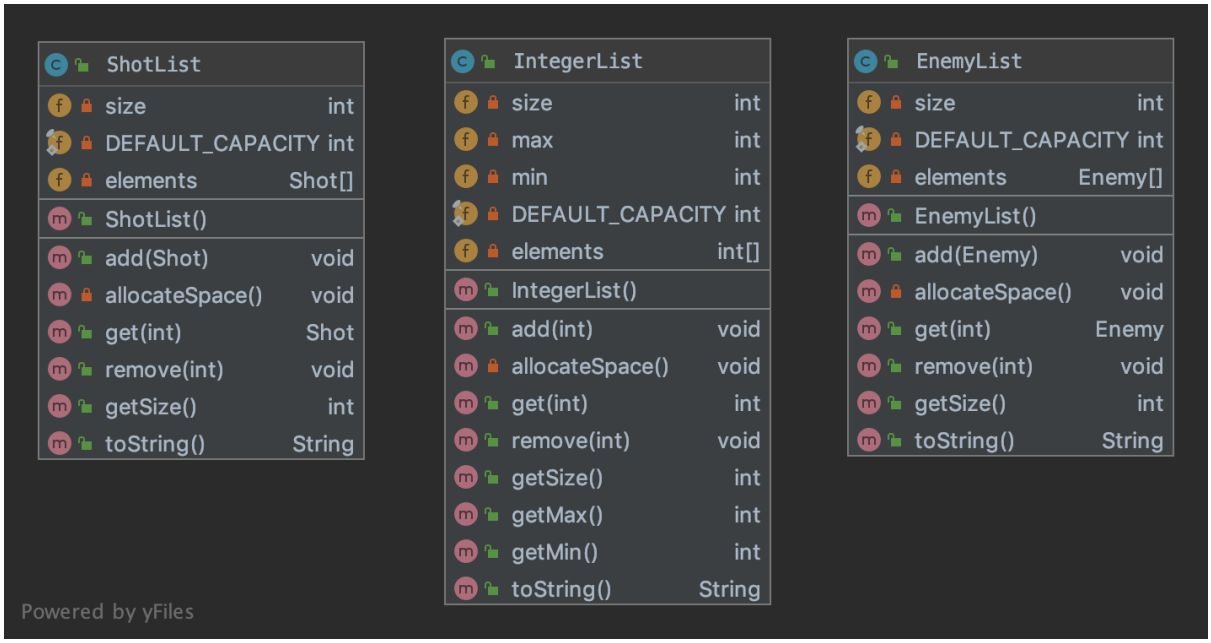


Figure 7: UML Diagram of the utils part of the project.

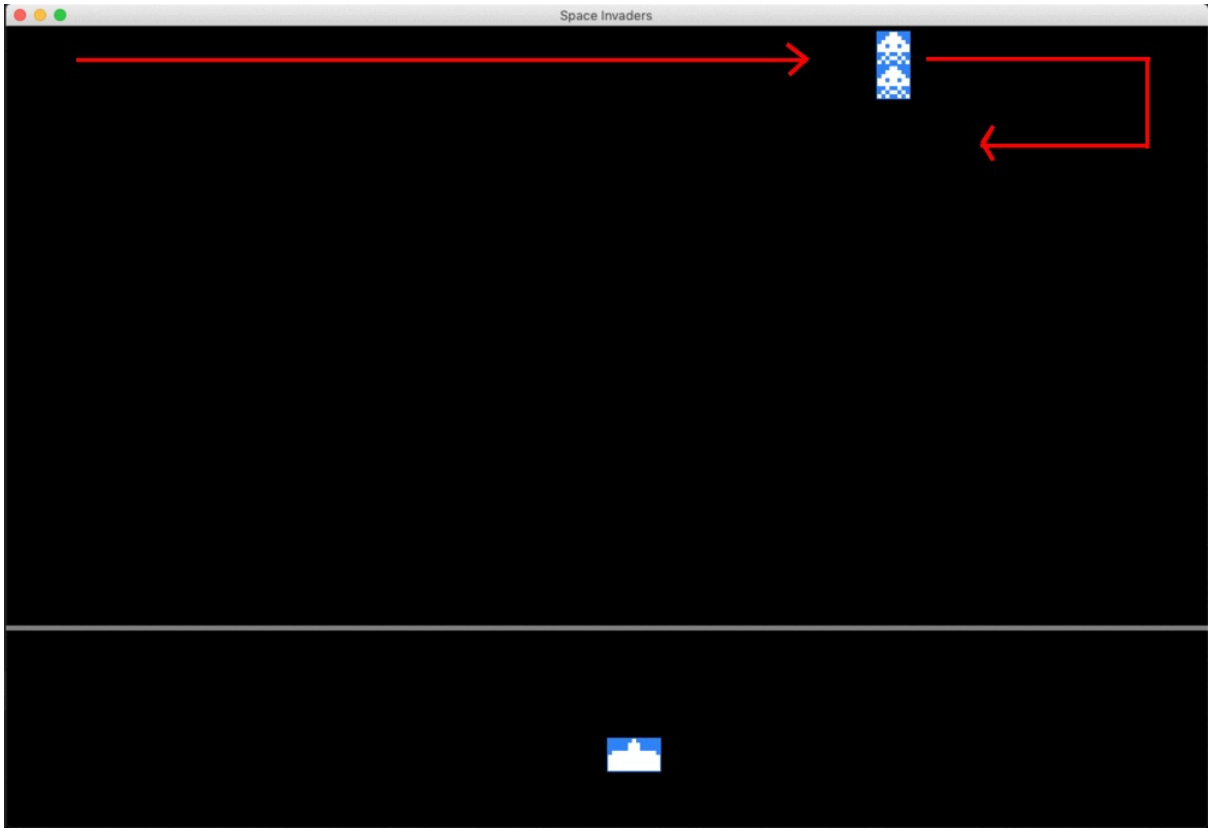


Figure 8: The movement of the enemies.