2 A basic Dungeon Crawler (homework assignment).

The goal of the entire lab exercise is to program the basis of a small game that displays a dungeon statically (without a camera) and sprites that can move within this dungeon while adhering to basic physics (minimum collision management).

2.1 Setup and assets

- Generate a new Java project.
- Create a 'Main' class that displays a message in the console to test your setup.
- Add a ressource directory in your repository and download the assets in it (spriteseet, sprites, level)
- Create an online git repository to share your project and synchronize.

2.2 UML presentation

This design revolves around three engines. Each of these engines runs in a specific thread timed by a timer. These timers are executed every 50 milliseconds.
Even though the study of threads falls under the advanced Java course, we will limit ourselves to the essentials here.
Here is the role of each engine:

- *The 'RenderEngine' is responsible for displaying all graphical elements. Some are static while others are animated.*
- *The 'PhysicsEngine' is tasked with modeling the game's physics. For us, this is relatively simple: we only manage constant speed and collisions.*
- *The 'GameEngine' handles user interactions (using the keyboard). In a more ambitious project, it would also load levels, manage game overs, etc.*[1]

To ensure that all three engines can be instantiated within a timer, they must implement an 'update' method. We thus define an 'Engine' interface to force the method to exists.
Logically, the main class has an instance of each engine.

[1] It is often coded as a large state machine. We will simplified it to only manage keyboard interactions.

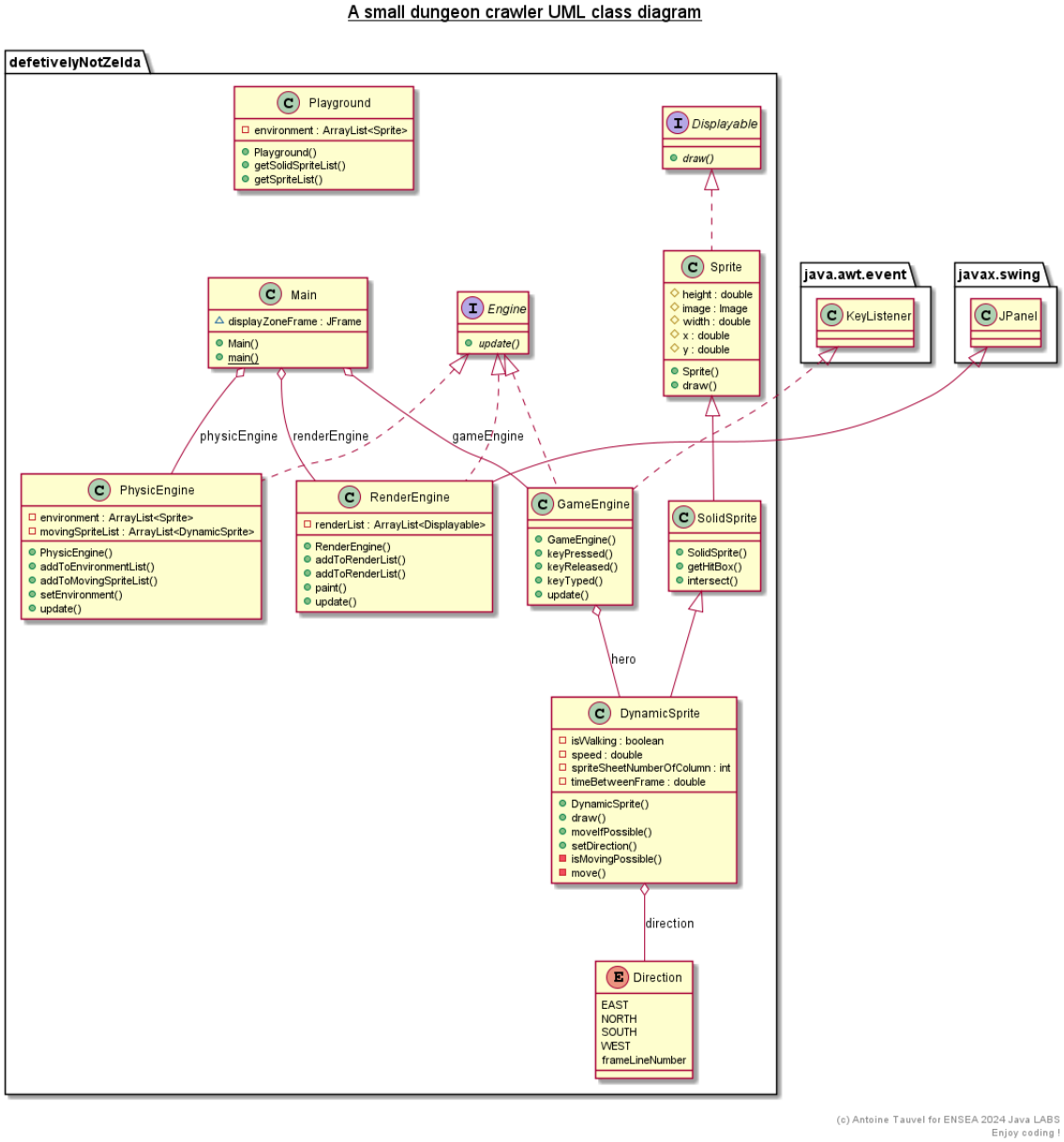

Figure 2.1: The complete UML diagram

We then have a cascade of classes inheriting from a 'Displayable' interface. This interface implements a single abstract method: 'draw', which must define the display method for each type of class.

- *The 'Sprite' class models decorative elements without physical substance. The render engine thus uses a list of 'Sprite'.*
- *The 'SolidSprite' class models decorative elements with a hitbox, such as trees or rocks. The physics engine uses a list of 'SolidSprite'.*
- *The 'DynamicSprite' class models animated elements. In our simplified version, there is only the hero. The game engine has a reference to the hero.*

Finally, to conclude this presentation, a dynamic sprite must always face a direction and potentially move in that direction. This is the role of the 'Direction' enumeration class. The four values (NORTH, SOUTH, EAST, WEST) are associated with four integers (2, 0, 3, 1) that correspond to the row number for the hero's drawing in the sprite sheet.

2.3 The Main class

2.3.1 Usage de Swing

Using code bellow, create a window application for our game.

```
1  public class Main{
2
3      JFrame displayZoneFrame;
4
5      public Main() throws Exception{
6          displayZoneFrame = new JFrame("Java Labs");
7          displayZoneFrame.setSize(400,600);
8          displayZoneFrame.setDefaultCloseOperation(EXIT_ON_CLOSE);
9          displayZoneFrame.setVisible(true);
10     }
11
12     public static void main (String[] args) throws Exception {
13         Main main = new Main();
14     }
15 }
```

Listing 2: Main class at the beginning.

As you saw in the lessons, the Swing framework allows for easy creation of GUIs[2]. Although it is quite old (2000), it is still relevant for quick results. Its main competitor, which was supposed to be its successor, is the JavaFX framework, but it requires proper management of package dependencies.
Swing displays elements in a graphical window ('JFrame'), which must itself contain one or more 'JPanel'. We will therefore use a 'JFrame' to display a window. We will use three methods: 'setSize' sets the size in pixels, 'setDefaultCloseOperation' specifies what should happen when the close button of the application is pressed (in our case, the end of the program), and 'setVisible' triggers the display.

2.3.2 The Engine Interface

Code the 'Engine' interface containing a single method `void update();`.

We will now create the skeleton of the 'RenderEngine' class.

Code the skeleton of 'RenderEngine', implementing the 'Engine' interface.
Display a message in the console when the "update" method is executed.
Once the functionality is validated, you can remove this display code.
Within the 'Main' class, use the code below to instantiate the 'RenderEngine' class within an instance of the 'Timer' class (a class from the Swing framework). Verify its proper functioning. I draw your attention to line 14 of this code, which uses a *lambda expression* to dynamically define the contents of a function. We will study this concept further in the Java course.

[2] Graphical User Interface

```java
public class Main{

    JFrame displayZoneFrame;

    RenderEngine renderEngine;

    public Main() throws Exception{
        displayZoneFrame = new JFrame("Java Labs");
        displayZoneFrame.setSize(400,600);
        displayZoneFrame.setDefaultCloseOperation(EXIT_ON_CLOSE);

        renderEngine = new RenderEngine();

        Timer renderTimer = new Timer(50,(time)-> renderEngine.update());

        renderTimer.start();
        displayZoneFrame.setVisible(true);
    }

    public static void main (String[] args) throws Exception {
        Main main = new Main();
    }
}
```

Listing 3: The instanciation of a Timer for the renderEngine update

2.4 Render Engine

As mentioned earlier, the Swing framework displays elements on a panel. It must synchronize the display; therefore, the 'RenderEngine' class must inherit from the 'JPanel' class (otherwise, the display will not be synchronized).

- Have the 'RenderEngine' class inherit from 'JPanel'.
- Add a line in the constructor of the 'Main' class to add our instance of 'RenderEngine' to the displayable elements, for example: `displayZoneFrame.getContentPane().add(renderEngine);`

2.4.1 The Sprite class

For reference, this class allows modeling any displayable element.

- Create a 'Displayable' interface with a single abstract method `public void draw(Graphics g);`.
- Create a 'Sprite' class implementing the 'Displayable' interface. This class has private attributes:
  - An image.
  - A position, modeled by two *double* values named x and y.
  - A size, modeled by two *double* values named width and height.
- Create a constructor that takes into account the five attributes.
- Override the method `public void draw(Graphics g)` to display the image at x and y. This is done with a method call on the graphics: g.drawImage (you need to find the right parameters).

2.4.2 The Render Engine

The 'RenderEngine' class currently makes a regular call to the 'update' method. We will now place the methods and attributes necessary for display.

- Add the private attribute `renderList`, which is a list of 'Displayable'.
- Write a default constructor that initializes the `renderList`.
- Create a setter for `renderList`.
- Create a method `public void addToRenderList(Displayable displayable)` that allows adding an element to the `renderList`.

We will now implement a key point of the program: polymorphism. Indeed, in the next method, even though we will only process a single 'Sprite' for now, the "correct" 'draw' method will be called regardless of the actual nature of the object (whether it's a 'Sprite', 'SolidSprite', or 'DynamicSprite').

Override the method `public void paint(Graphics g)`. This method starts by calling the paint method of the superclass (the parent class 'JPanel'), then for each element in the `renderList`, it calls the 'draw' method.
All that remains is to "paint" the component regularly. To do this, simply call the `repaint()` method within the `update` method (not paint; repaint generates the graphic parameter).
Test the whole setup using the lines below. If the result does not display a tree, please correct the issues, making sure to consult the teacher if necessary.

2.4.3 The SolidSprite class

Create the class 'SolidSprite' that inherits from the 'Sprite' class.
This class currently has only one method: a constructor equivalent to that of 'Sprite' (calling the `super` method).

2.4.4 The Direction Enumerate

Create an enum class named 'Direction' that can take the values 'NORTH', 'SOUTH', 'EAST', and 'WEST'. This class allows you to retrieve a unique integer value associated with each enumerated constant. You can use the code bellow, or try to write it yourself...

```
public enum Direction {
    NORTH(2),SOUTH(0),EAST(3),WEST(1);
    private int frameLineNumber;

    Direction(int frameLineNumber) {
        this.frameLineNumber = frameLineNumber;
    }

    public int getFrameLineNumber() {
        return frameLineNumber;
    }
}
```

Listing 4: The Direction Enum

2.4.5 The DynamicSprite class and animation rendering

Create the class 'DynamicSprite' that inherits from the 'SolidSprite' class.
This class contains the following additional attributes:

- A boolean named `isWalking` set to `true` by default.
- A `double` named `speed` set to 5 by default.
- A final attribute of type int: `spriteSheetNumberOfColumn` (set to 10 by default).
- An int attribute named `timeBetweenFrame` set to 200 by default (200 milliseconds between two frames).
- An attribute of type `Direction` named `direction`.

Code a setter for the variable `direction`.

We now will use a SpriteSheet in which every direction animation is on a single line.

Figure 2.2: La spritesheet du héros.

Override the draw method, which includes the following behaviors:

- Calculate an integer `index` representing the number of the image to display. To do this, divide the current time (`System.currentTimeMillis()`) by the time between two frames modulo the total number of frames.
- Retrieve an integer `attitude` corresponding to the numerical value of the direction.
- Display the image using `drawImage` with the correct parameters. The source starts at $(index * width, attitude * height)$ and ends at $((index + 1) * width, (attitude + 1) * height)$.
- Integrate a hero in the constructor of the `Main` class and verify its animation, as indicated below.

```
DynamicSprite hero = new DynamicSprite(200,300,
            ImageIO.read(new File("./img/heroTileSheetLowRes.png")),48,50);
renderEngine.addToRenderList(hero);
```

2.5 Game Engine

Let's take a break with the simplest class! Hang in there, we are almost there!

- Create a class `GameEngine` that implements the `Engine` and `KeyListener` interfaces.
- Create a private final reference attribute for the hero.
- Create a constructor that sets the reference to the hero.
- Within the `keyPressed` method, use a switch statement on the variable `e.getKeyCode` to set the hero's direction variable. For your information, the key codes for the arrows are in the format: `KeyEvent.VK_UP` for the up arrow.
- Add the creation of an instance of `GameEngine` in the main method.
- Inspired by the instantiation code of the `RenderEngine`, implement a `Timer` instance executing the `GameEngine` in the `Main` constructor (optional, see remark below).
- In the `Main` constructor, add a key listener as follows: `displayZoneFrame.addKeyListener(gameEngine);`
- Test your code: the hero should have an animation in the direction of the arrow indicated by the keyboard, without any movement.

You will probably be surprised that the `update` method remains empty. This is related to the simplified structure of our project, given the available time. A well-designed game engine incorporates a FSM (finite state machine) that should evolve at regular intervals.

2.6 Physic Engine

2.6.1 DynamicSprite class modification

We must now write the moving method of our dynamic sprite. We do that by adding the speed at the position every other delay.
For exemple :

```
switch (direction){
            case NORTH -> {
                this.y-=speed;
            }
```

But first, we need to check if the movement is possible. We will proceed in two steps:

- First, we calculate a hitbox offset by the intended movement.
- Then, we check for all collisions with all solid elements.

The class `Rectangle2D`, which is part of Swing, includes an `intersect` method that returns a boolean...

- Write a private method within the `DynamicSprite` class `private void move()` that moves the Sprite.
- Write a private method `private boolean isMovingPossible(ArrayList<Sprite> environment)` that creates a local variable `hitBox` of type `Rectangle2D.Double` corresponding to the sprite moved by Speed pixels in the correct direction.
- Complete the `isMovingPossible` method using a for loop that iterates through all elements of the environment parameter. If the element is a `SolidSprite` and there is an intersection, and the element is not the `DynamicSprite` itself, then return `false`.
- At the end of the for loop, return true.
- Finally, code the function `public void moveIfPossible(ArrayList<Sprite> environment)`, which intelligently calls the two previous functions.

2.6.2 The PhysicEngine class

- Create a class PhysicEngine that implements the Engine interface.
- This class has two attributes:
  - A list movingSpriteList that contains all the Sprites to be moved.
  - A list environment that contains all the solid sprites.
- Create a method to add an element to the movingSpriteList.
- Create a setter for the environment list.
- Finally, override the update method with a for loop that calls the moveIfPossible method for all elements in the movingSpriteList.
- Test using the Main code below. The character should move until it "bumps" against the rock.

```
public Main() throws Exception{
        displayZoneFrame = new JFrame("Java Labs");
        displayZoneFrame.setSize(400,600);
        displayZoneFrame.setDefaultCloseOperation(EXIT_ON_CLOSE);

        DynamicSprite hero = new DynamicSprite(0,300,
                ImageIO.read(new File("./img/heroTileSheetLowRes.png")),48,50);

        renderEngine = new RenderEngine();
        physicEngine = new PhysicEngine();
        gameEngine = new GameEngine(hero);

        Timer renderTimer = new Timer(50,(time)-> renderEngine.update());
        Timer gameTimer = new Timer(50,(time)-> gameEngine.update());
        Timer physicTimer = new Timer(50,(time)-> physicEngine.update());

        renderTimer.start();
        gameTimer.start();
        physicTimer.start();

        displayZoneFrame.getContentPane().add(renderEngine);
        displayZoneFrame.setVisible(true);

      SolidSprite testSprite = new SolidSprite(250,300,(new File("./img/rock.png")),64,64);
        renderEngine.addToRenderList(testSprite);
        renderEngine.addToRenderList(hero);
        physicEngine.addToMovingSpriteList(hero);
        physicEngine.setEnvironment(new ArrayList<SolidSprite>(testSprite));
}
```

Listing 5: Test code for the Physic Engine

2.7 Level Loader : the PlayGround class

A small gift : I give you the PlayGround class. Copy and Paste it in your project.

```
public class Playground {
    private ArrayList<Sprite> environment = new ArrayList<>();

    public Playground (String pathName){
        try{
            final Image imageTree = ImageIO.read(new File("./img/tree.png"));
            final Image imageGrass = ImageIO.read(new File("./img/grass.png"));
            final Image imageRock = ImageIO.read(new File("./img/rock.png"));
            final Image imageTrap = ImageIO.read(new File("./img/trap.png"));

            final int imageTreeWidth = imageTree.getWidth(null);
            final int imageTreeHeight = imageTree.getHeight(null);

            final int imageGrassWidth = imageGrass.getWidth(null);
            final int imageGrassHeight = imageGrass.getHeight(null);
```

```
            final int imageRockWidth = imageRock.getWidth(null);
            final int imageRockHeight = imageRock.getHeight(null);

            BufferedReader bufferedReader = new BufferedReader(new FileReader(pathName));
            String line=bufferedReader.readLine();
            int lineNumber = 0;
            int columnNumber = 0;
            while (line!= null){
                for (byte element : line.getBytes(StandardCharsets.UTF_8)){
                    switch (element){
                        case 'T' : environment.add(new SolidSprite(columnNumber*imageTreeWidth,
                                lineNumber*imageTreeHeight,imageTree, imageTreeWidth, imageTreeHeight));
                                    break;
                        case ' ' : environment.add(new Sprite(columnNumber*imageGrassWidth,
                                lineNumber*imageGrassHeight, imageGrass, imageGrassWidth, imageGrassHeight));
                            break;
                        case 'R' : environment.add(new SolidSprite(columnNumber*imageRockWidth,
                                lineNumber*imageRockHeight, imageRock, imageRockWidth, imageRockHeight));
                            break;
                    }
                    columnNumber++;
                }
                columnNumber =0;
                lineNumber++;
                line=bufferedReader.readLine();
            }
        }
        catch (Exception e){
            e.printStackTrace();
        }
    }

    public ArrayList<Sprite> getSolidSpriteList(){
        ArrayList <Sprite> solidSpriteArrayList = new ArrayList<>();
        for (Sprite sprite : environment){
            if (sprite instanceof SolidSprite) solidSpriteArrayList.add(sprite);
        }
        return solidSpriteArrayList;
    }

    public ArrayList<Displayable> getSpriteList(){
        ArrayList <Displayable> displayableArrayList = new ArrayList<>();
        for (Sprite sprite : environment){
            displayableArrayList.add((Displayable) sprite);
        }
        return displayableArrayList;
    }
}
```

Listing 6: The Playground class

2.8 The final integration

All that's left is to integrate an instance of PlayGround in the Main constructor and to integrate its lists of Sprite and SolidSprite into the renderList and environment of the RenderEngine and PhysicEngine.
Congratulations, you now have a solid foundation for what comes next!

Figure 2.3: Let's play !