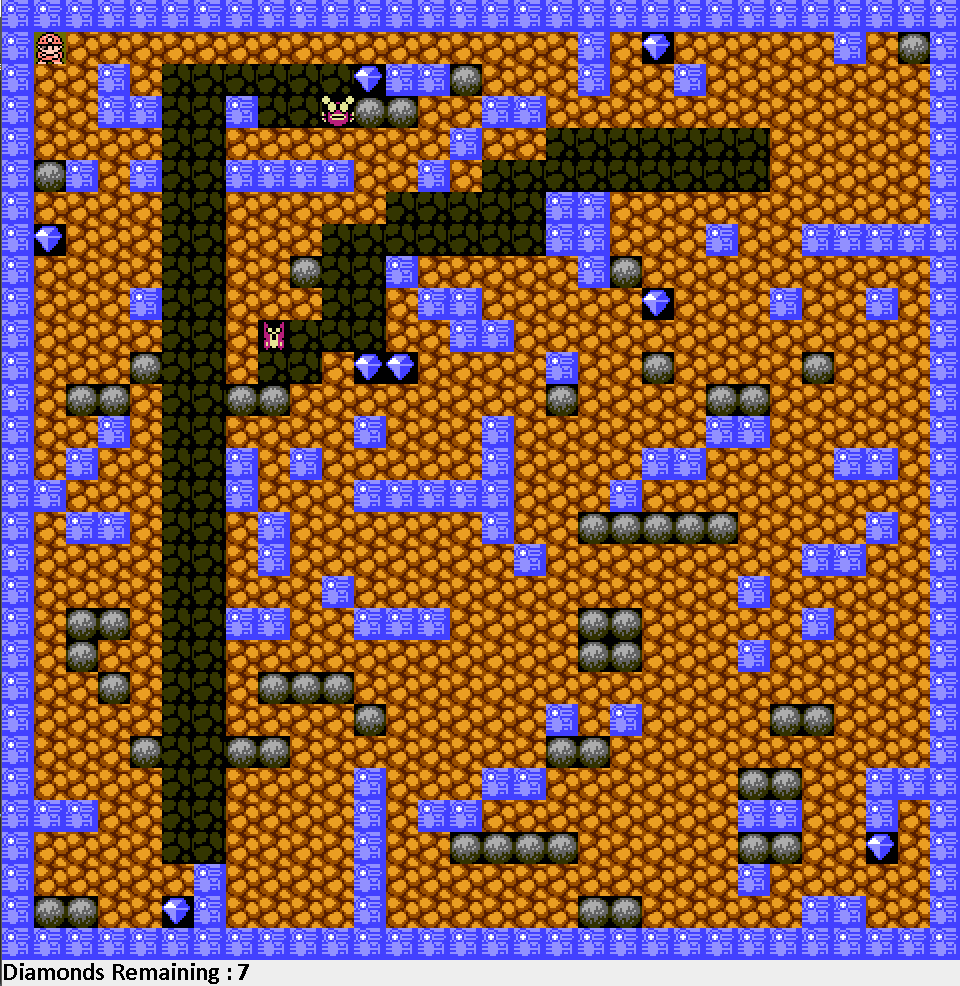
JAVA PROJECT

BOULDERDASH

*Caron Alexis – Descamps Anthony – Fritsch Florian – Libessart Dimitri*

****

Contents

[Team Presentation 2](#_Toc485994286)

[Context Analysis: 2](#_Toc485994287)

[Objectives: 2](#_Toc485994288)

[Requirement of the Project: 2](#_Toc485994289)

[Constraints: 2](#_Toc485994290)

[UML Language: 3](#_Toc485994291)

[Package Diagram 3](#_Toc485994292)

[Component Diagram 4](#_Toc485994293)

[Sequence Diagram 4](#_Toc485994294)

[Class Diagram 5](#_Toc485994295)

[Model-View-Controller: 6](#_Toc485994296)

[JUnit Tests 9](#_Toc485994297)

[In the Model Package: 9](#_Toc485994298)

[Encountered Issues: 15](#_Toc485994299)

[Planning 16](#_Toc485994300)

[Projected Planning: 16](#_Toc485994301)

[Effective Planning: 17](#_Toc485994302)

[Results of the project: 18](#_Toc485994303)

[Group Result: 18](#_Toc485994304)

[Individual Results: 18](#_Toc485994305)

# Team Presentation

**Caron Alexis: Leader**

**Descamps Anthony**

**Fritsch Florian**

**Libessart Dimitri**

# Context Analysis:

## Objectives:

* Design 5 levels of the game Boulder Dash.
* Write the code of the game with the Java programming language.
* Use a Database to load the sprites of the game.

## Requirement of the Project:

* Use the UML language.
* Create a Package Diagram, a Component Diagram, a Sequence Diagram and five Class Diagrams (for Model, View, Contract, Controller and Main).
* Create several JUnit tests and create a SureFire Report.

## Constraints:

* The code must be in a Model-View-Controller software architectural pattern.
* Java, Maven, Git Hub and JUnit are mandatory.
* Neither other graphic framework than Swing is allowed.

## UML Language:

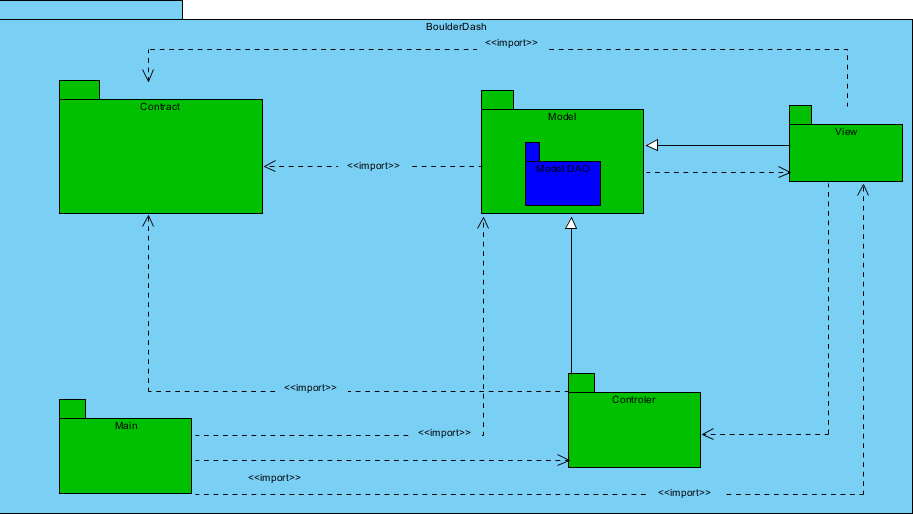
The Unified Modelling Language, or UML Language, is a modelling language in the field of software engineering. The UML Language offers a way to visualize a system architectural blueprints in a diagram, including elements such as activities, components, relationship, how the system run, and external user interfaces

For the project, we had to draw several diagrams. All of them had to respect the UML language.

### Package Diagram

The Package Diagram is a diagram which describe and represent all the package in the system. In this one, we must draw the package and the sub-package, and the relations between them. The classes and the interfaces don’t must be represented.

This is our Package Diagram:

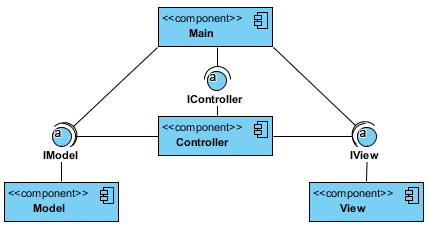


### Component Diagram

The Component Diagram is a diagram which describe how component are wired together. They are used to illustrate the structure of complex systems. A component is represented by a rectangle with either the keyword “component”, and the interfaces are represented by a circle.

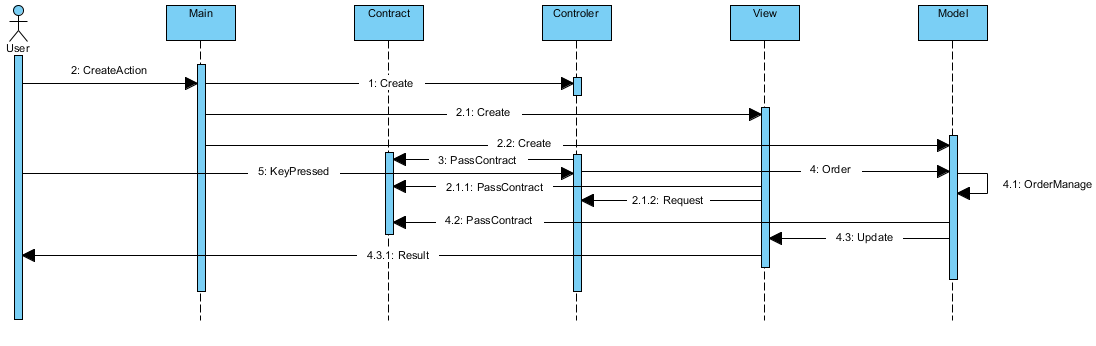
----------------------

This is our Component Diagram:



### Sequence Diagram

The sequence Diagram is an interaction diagram that shows how objects operate with one another and in what order. It is a construct of a message sequence chart.  
The objects are represented by a vertical lifeline, and the message which are the actions operated, are represented by horizontal arrows.

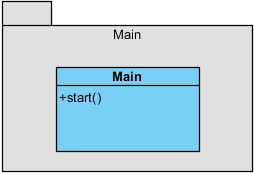
This is our Sequence Diagram:

### Class Diagram

The Class Diagram is the most important diagram in the UML language, because he’s the most detailed. He describes the structure of a system by showing the system’s classes, and their attributes, their operations, and their relationship among their objects.

For this project, we must draw five classes diagrams, one per each package.

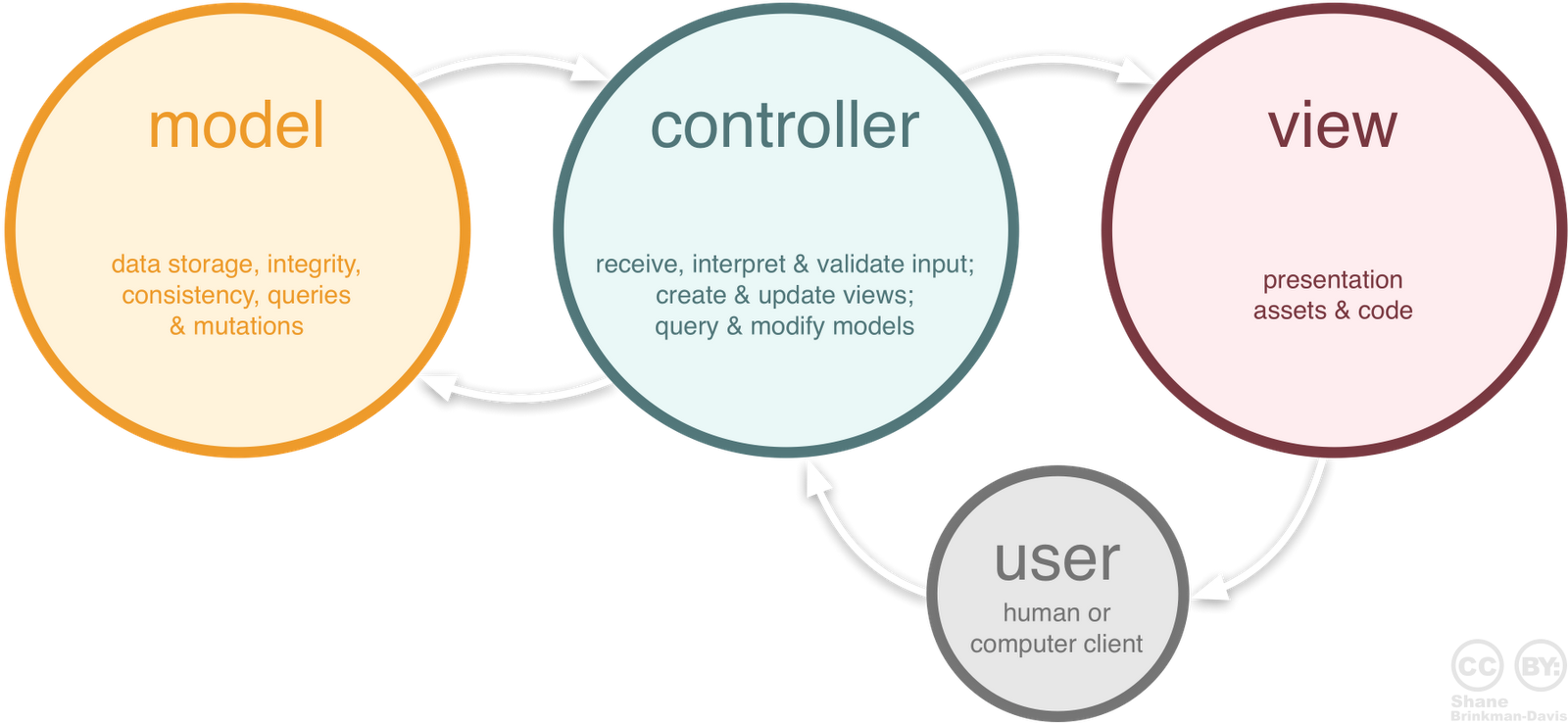
#### Main Class Diagram:



## Model-View-Controller:

The Model-View-controller architectural pattern, also call the MVC pattern, is a graphic interface for implementing user interfaces on computers.

It divides a given application into three interconnected parts, in order to separate representations of information from the ways that information is presented to and accepted from the user.

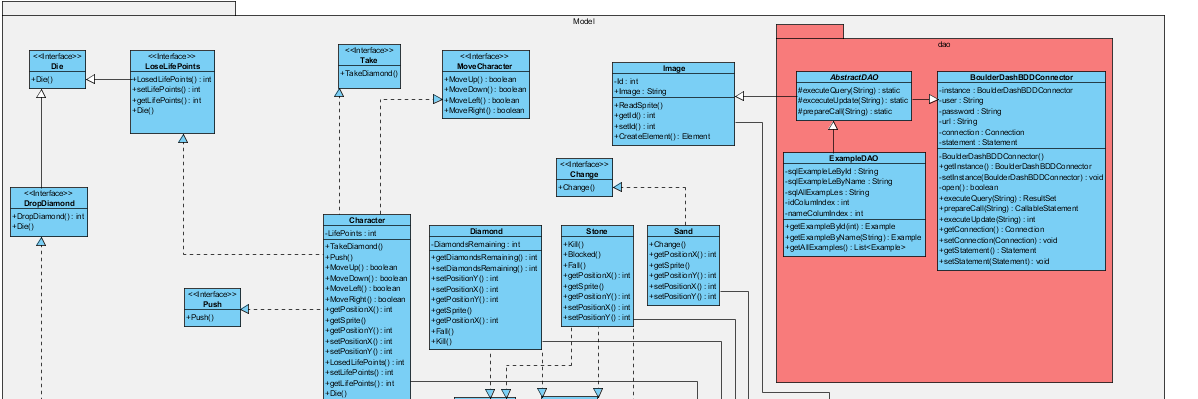
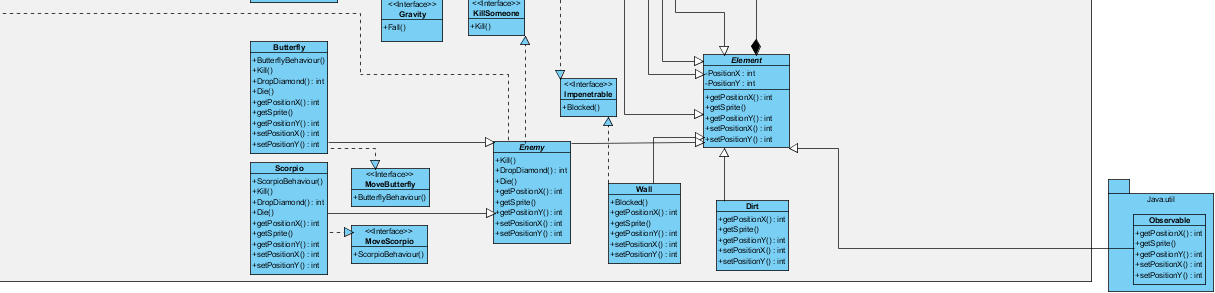
These three parts are the Model, the View, and the Controller.

#### Model Class Diagram:

The Model part is the central component of the pattern. It expresses the application’s behaviour in terms of the problem, independent of the user interface.

The model also looks after the data, the logic, and the rules of the application.

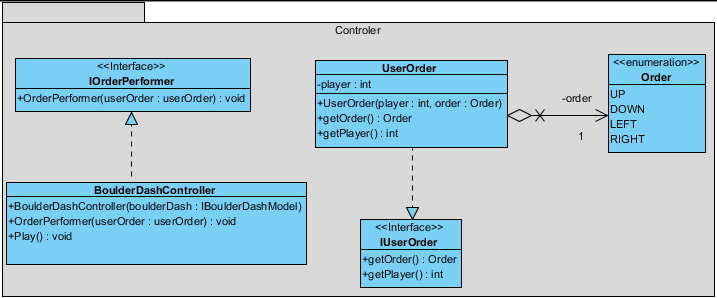
This is our Model Class Diagram:



#### Con

#### controller Class Diagram:

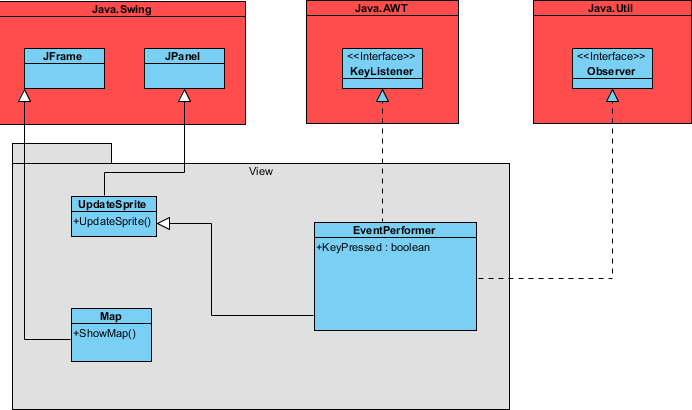
The Controller part is the part that accepts input and converts it to commands for the model or the view. It’s the only part that contain no data.

This is our Controller Class Diagram:

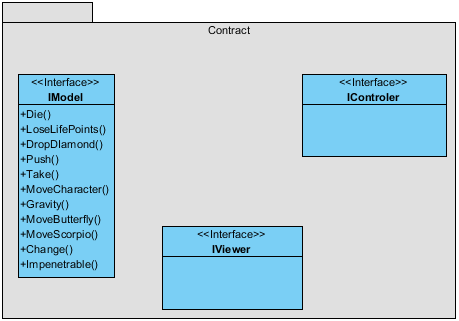
#### View Class Diagram:

The View part is the part which can be any output representation of information. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants.

This is our View Class Diagram:



#### Contract Class Diagram:



## JUnit Tests

JUnit is a unit testing framework using by the Java programming language. The advantage of this framework is that he allows to create tests before programming, that is related to the Test-Driven Development.

During this project, we write a set of tests:

#### ChangeDirtTest:

This test shows us if the element “dirt” can be changed in another element.

**package** model**;**

**import** **static** org**.**junit**.**Assert**.\*;**

**import** org**.**junit**.**After**;**

**import** org**.**junit**.**AfterClass**;**

**import** org**.**junit**.**Before**;**

**import** org**.**junit**.**BeforeClass**;**

**import** org**.**junit**.**Test**;**

**public** **class** ChangeDirtTest **{**

@BeforeClass

**public** **static** **void** setUpBeforeClass**()** **throws** Exception **{**

**}**

@AfterClass

**public** **static** **void** tearDownAfterClass**()** **throws** Exception **{**

**}**

@Before

**public** **void** setUp**()** **throws** Exception **{**

**}**

@After

**public** **void** tearDown**()** **throws** Exception **{**

**}**

@Test

**public** **void** ChangeDirt**()** **{**

**final** **int** expected **=** 0**;**

Change**();**

assertEquals**(**expected**,** model**.**Change**.**Changer**());**

**}**

**}**

#### DropDiamondTest:

This test shows us if the element “Diamond” is affected by the method “Fall” in the interface “Gravity”.

**package** model**;**

**import** **static** org**.**junit**.**Assert**.\*;**

**import** org**.**junit**.**After**;**

**import** org**.**junit**.**AfterClass**;**

**import** org**.**junit**.**Before**;**

**import** org**.**junit**.**BeforeClass**;**

**import** org**.**junit**.**Test**;**

**public** **class** DropDiamondTest **{**

@BeforeClass

**public** **static** **void** setUpBeforeClass**()** **throws** Exception **{**

**}**

@AfterClass

**public** **static** **void** tearDownAfterClass**()** **throws** Exception **{**

**}**

@Before

**public** **void** setUp**()** **throws** Exception **{**

**}**

@After

**public** **void** tearDown**()** **throws** Exception **{**

**}**

@Test

**public** **void** DropDiamonds**()** **{**

**public** **int** expected **=** 1**;**

DropDiamond**();**

assertEquals**(**expected**,** model**.**DropDiamondTest**.**DropDiamond**());**

**}**

**}**

#### EndTestGame:

This test can verify if the game can be finished when the play hasn’t life yet.

**package** model**;**

**import** **static** org**.**junit**.**Assert**.\*;**

**import** org**.**junit**.**After**;**

**import** org**.**junit**.**AfterClass**;**

**import** org**.**junit**.**Before**;**

**import** org**.**junit**.**BeforeClass**;**

**import** org**.**junit**.**Test**;**

**public** **class** EndGameTest **{**

@BeforeClass

**public** **static** **void** setUpBeforeClass**()** **throws** Exception **{**

**}**

@AfterClass

**public** **static** **void** tearDownAfterClass**()** **throws** Exception **{**

**}**

@Before

**public** **void** setUp**()** **throws** Exception **{**

**}**

@After

**public** **void** tearDown**()** **throws** Exception **{**

**}**

@Test

**public** **void** EndTest**()** **{**

**final** **int** expected1 **=** 1**;**

**final** **int** expected2 **=** 0**;**

LosedLifePoints**();**

LosedLifePoints**();**

LosedLifePoints**();**

assertEquals**(**expected2**,** model**.**LoseLifePoints**.**getLifePoints**());**

assertEquals**(**expected1**,** model**.**LoseLifePoints**.**EndGame**());**

**}**

**}**

#### EnemyHitTest:

This test shows us if the enemy can drop a life point when he touches the player.

**package** model**;**

**import** **static** org**.**junit**.**Assert**.\*;**

**import** org**.**junit**.**After**;**

**import** org**.**junit**.**AfterClass**;**

**import** org**.**junit**.**Before**;**

**import** org**.**junit**.**BeforeClass**;**

**import** org**.**junit**.**Test**;**

**public** **class** EnemyHitTest **{**

@BeforeClass

**public** **static** **void** setUpBeforeClass**()** **throws** Exception **{**

**}**

@AfterClass

**public** **static** **void** tearDownAfterClass**()** **throws** Exception **{**

**}**

@Before

**public** **void** setUp**()** **throws** Exception **{**

**}**

@After

**public** **void** tearDown**()** **throws** Exception **{**

**}**

@Test

**public** **void** EnemyHit**()** **{**

**final** **int** expected **=** 2**;**

LosedLifePoints**();**

assertEquals**(**expected**,** model**.**LoseLifePoints**.**getLifePoints**());**

**}**

**}**

#### GravityLeftTest:

-------manque de note-------

**package** model**;**

**import** **static** org**.**junit**.**Assert**.\*;**

**import** org**.**junit**.**After**;**

**import** org**.**junit**.**AfterClass**;**

**import** org**.**junit**.**Before**;**

**import** org**.**junit**.**BeforeClass**;**

**import** org**.**junit**.**Test**;**

**public** **class** GravityLeftTest **{**

@BeforeClass

**public** **static** **void** setUpBeforeClass**()** **throws** Exception **{**

**}**

@AfterClass

**public** **static** **void** tearDownAfterClass**()** **throws** Exception **{**

**}**

@Before

**public** **void** setUp**()** **throws** Exception **{**

**}**

@After

**public** **void** tearDown**()** **throws** Exception **{**

**}**

@Test

**public** **void** GravityRight**()** **{**

**final** **int** expected1 **=** **-**1**;** //Position X

**final** **int** expected2 **=** **-**1**;** //Position Y

assertEquals**(**expected1**,** model**.**Gravity**.**getPositionX**());**

assertEquals**(**expected1**,** model**.**Gravity**.**getPositionY**());**

**}**

**}**

#### GravityRightTest:

-----------manque de note---------------

**package** model**;**

**import** **static** org**.**junit**.**Assert**.\*;**

**import** org**.**junit**.**After**;**

**import** org**.**junit**.**AfterClass**;**

**import** org**.**junit**.**Before**;**

**import** org**.**junit**.**BeforeClass**;**

**import** org**.**junit**.**Test**;**

**public** **class** GravityRightTest **{**

@BeforeClass

**public** **static** **void** setUpBeforeClass**()** **throws** Exception **{**

**}**

@AfterClass

**public** **static** **void** tearDownAfterClass**()** **throws** Exception **{**

**}**

@Before

**public** **void** setUp**()** **throws** Exception **{**

**}**

@After

**public** **void** tearDown**()** **throws** Exception **{**

**}**

@Test

**public** **void** GravityRight**()** **{**

**final** **int** expected1 **=** 1**;** //Position X

**final** **int** expected2 **=** **-**1**;** //Position Y

assertEquals**(**expected1**,** model**.**Gravity**.**getPositionX**());**

assertEquals**(**expected1**,** model**.**Gravity**.**getPositionY**());**

**}**

**}**

# Encountered Issues:

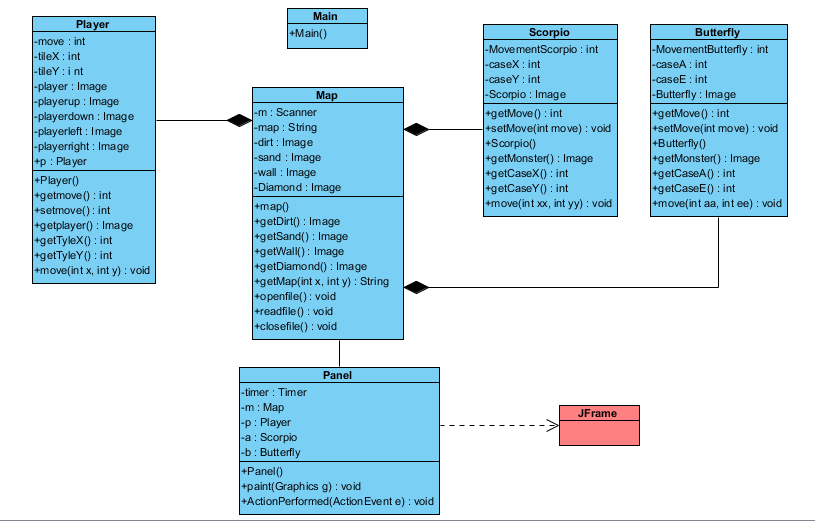
In the middle of the project, we realized that the Model-View-Controller architecture was too harder, and we didn’t succeed to implement our game with this pattern.

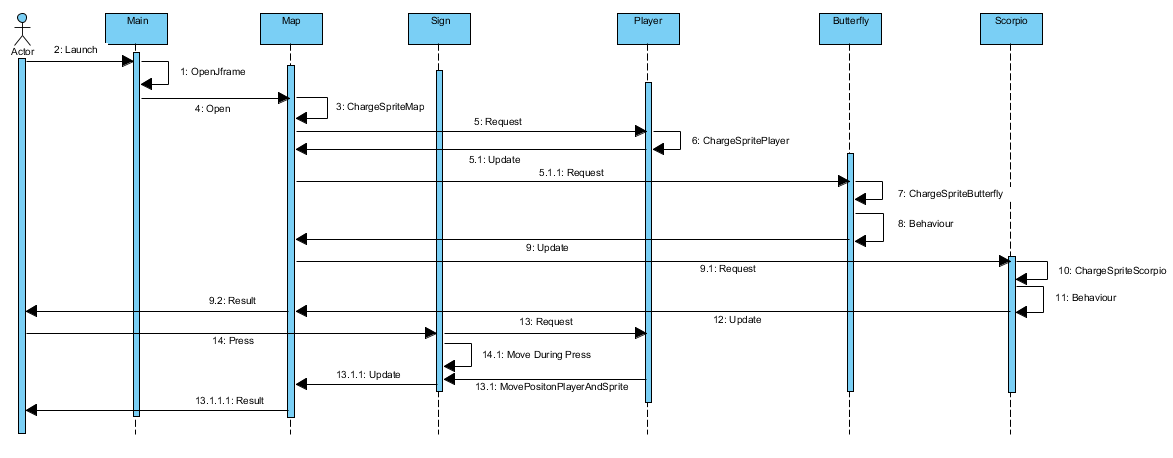
Actually, we encountered problems with the Controller Package, because we failed to find what to set in this. Also, the Contract Package was vague. We didn’t know if we must set interfaces, or classes.

Then, we didn’t use the MVC pattern, because we lack time, and we didn’t succeed to set our programming code.

Therefore, we try to set a game without this architecture, in another project.

So, we made another class diagram here:



And we also re-draw the Sequence Diagram:

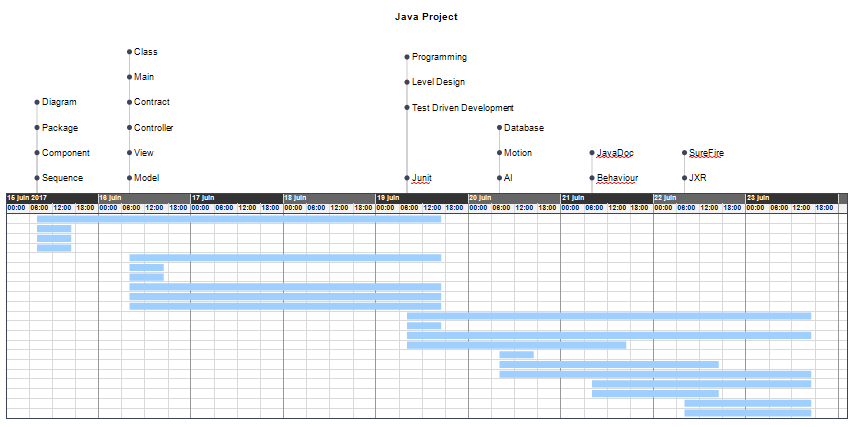
# Planning

## Projected Planning:

At the beginning of the project, we plan this planning with a Gantt project. We supposed that the project will takes place without time problems, so we space out all of the steps.



## Effective Planning:



At the end of this project, we can have realized that the diagrams and the programming were the most important and the most difficult parts.

Therefore, we set more time that we planned at the beginning.

Trello:

Faire une partie sur trello

# Results of the project:

## Group Result:

This project

## Individual Results:

**Descamps Anthony:** Overall, this project was intense because it was very difficult. We have respected all the functionality of the game even if we didn’t respect the MVC. Furthermore, my group has been productive during all the project. This project was very interesting because it regroups all the Prosit. I would keep a good memory of this week of project.

**Caron Alexis:**

**Fritsch Florian:**

**Libessart Dimitri:**