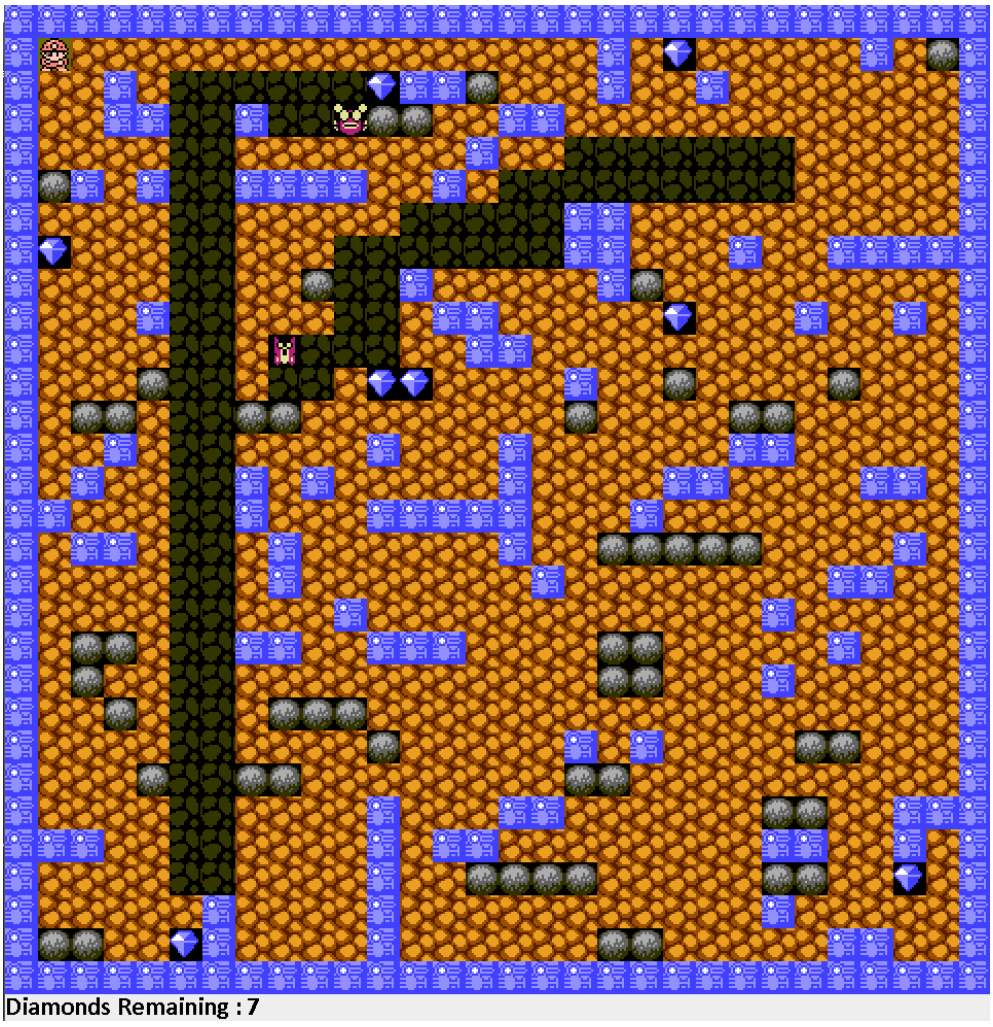
JAVA PROJECT

BOULDERDASH



Caron Alexis – Descamps Anthony – Fritsch Florian – Libessart Dimitri

Contents

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:

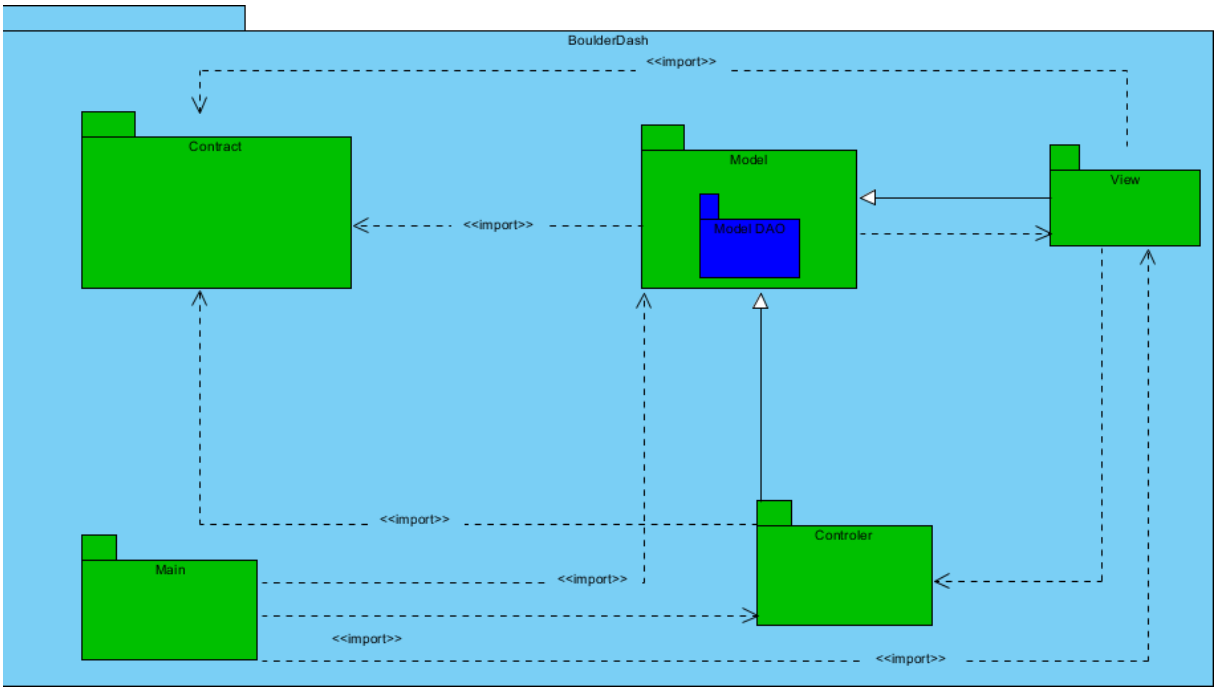
The Unified Modelling Language, or UML Language, is a modelling language in the field of software engineering. The UML Language offers a way to visualize a system architectural blueprints in a diagram, including elements such as activities, components, relationship, how the system run, and external user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

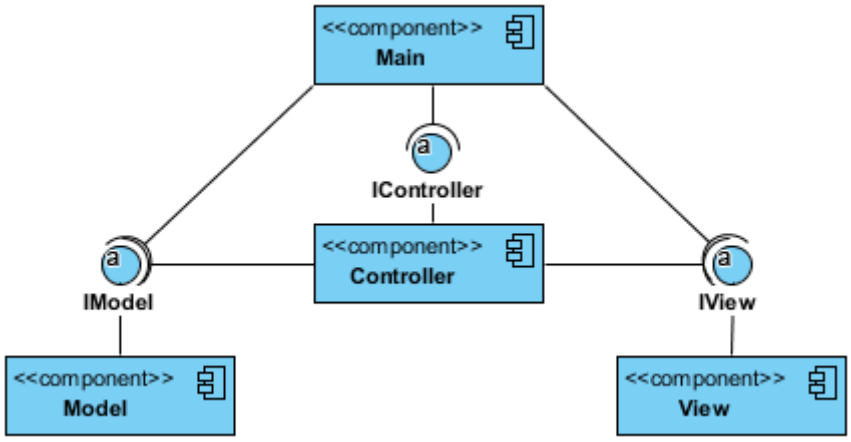


Component Diagram

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.

The semi-circles on the circles represent the way where the interfaces are dispose.

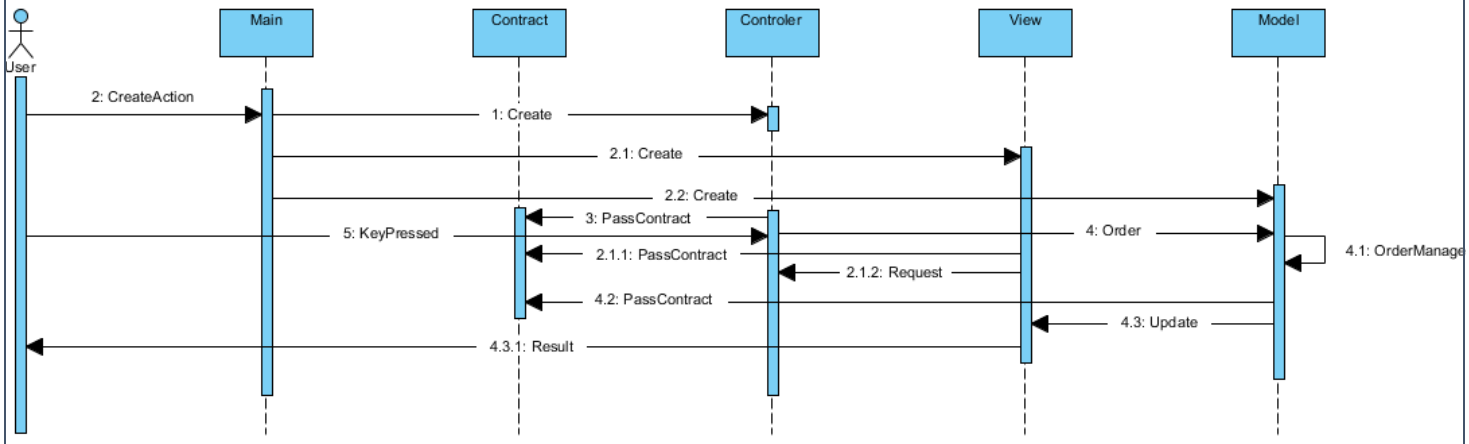
This is our Component Diagram:



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:

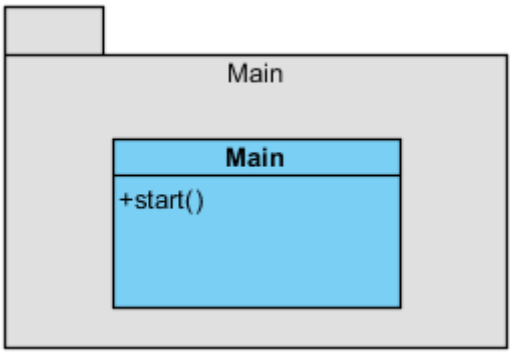


Class Diagram

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:

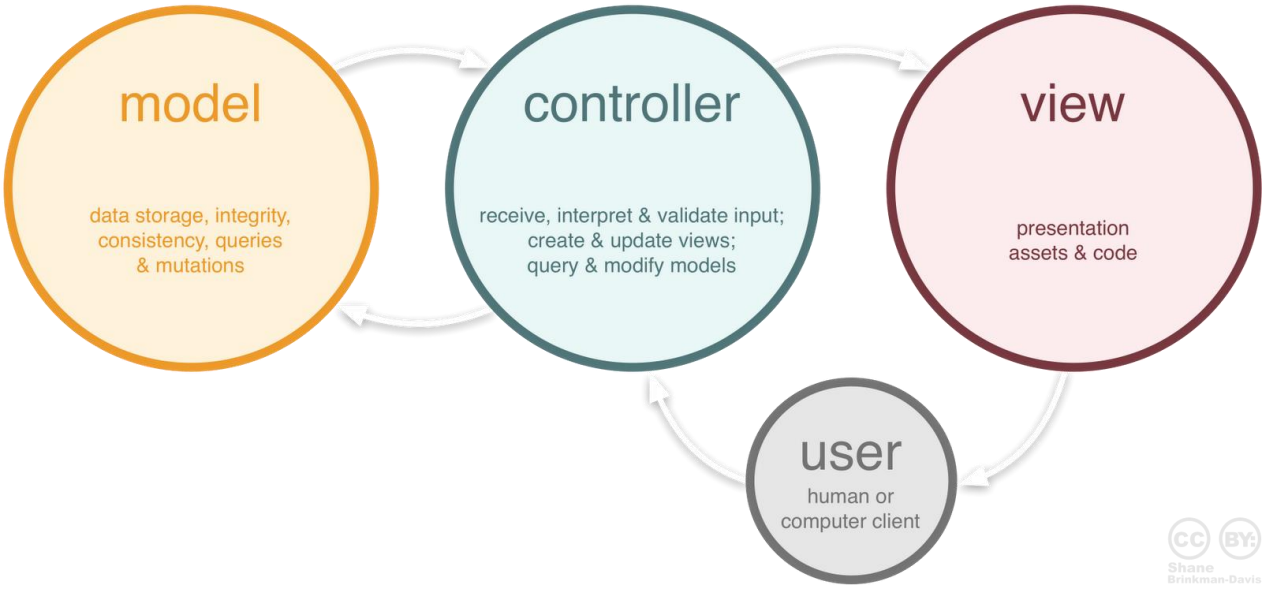


Model-View-Controller:

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.

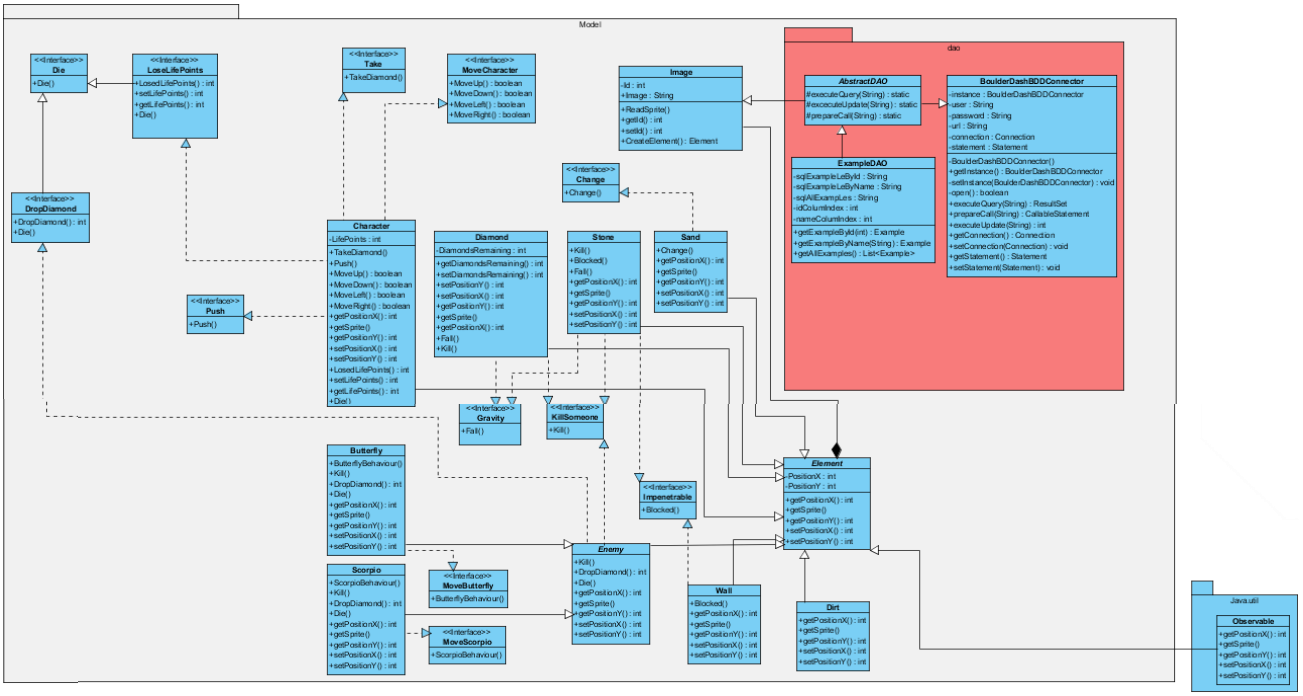


Model Class Diagram:

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.

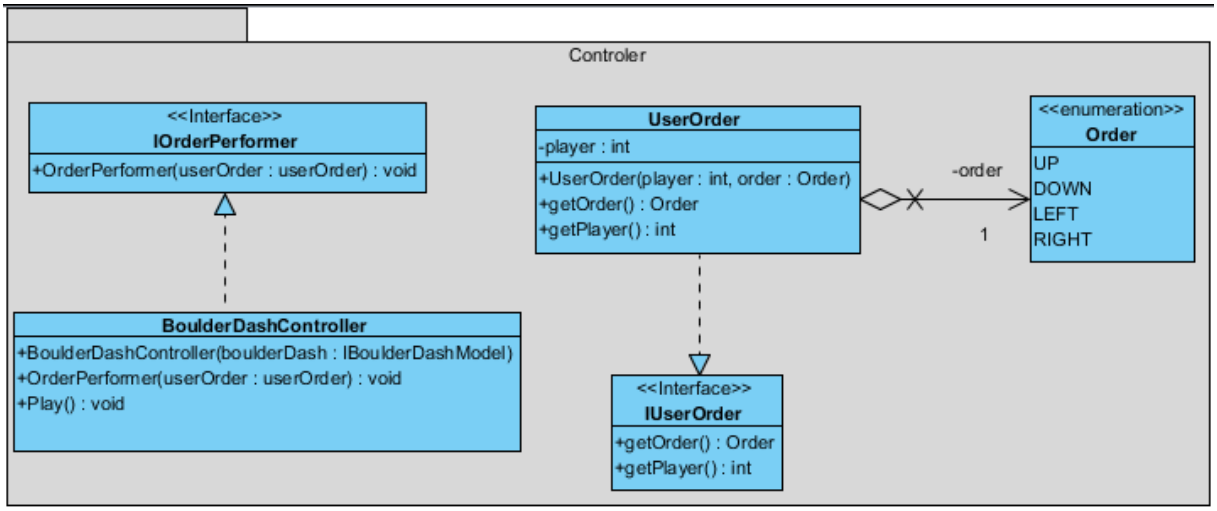
This is our Model Class Diagram:



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.

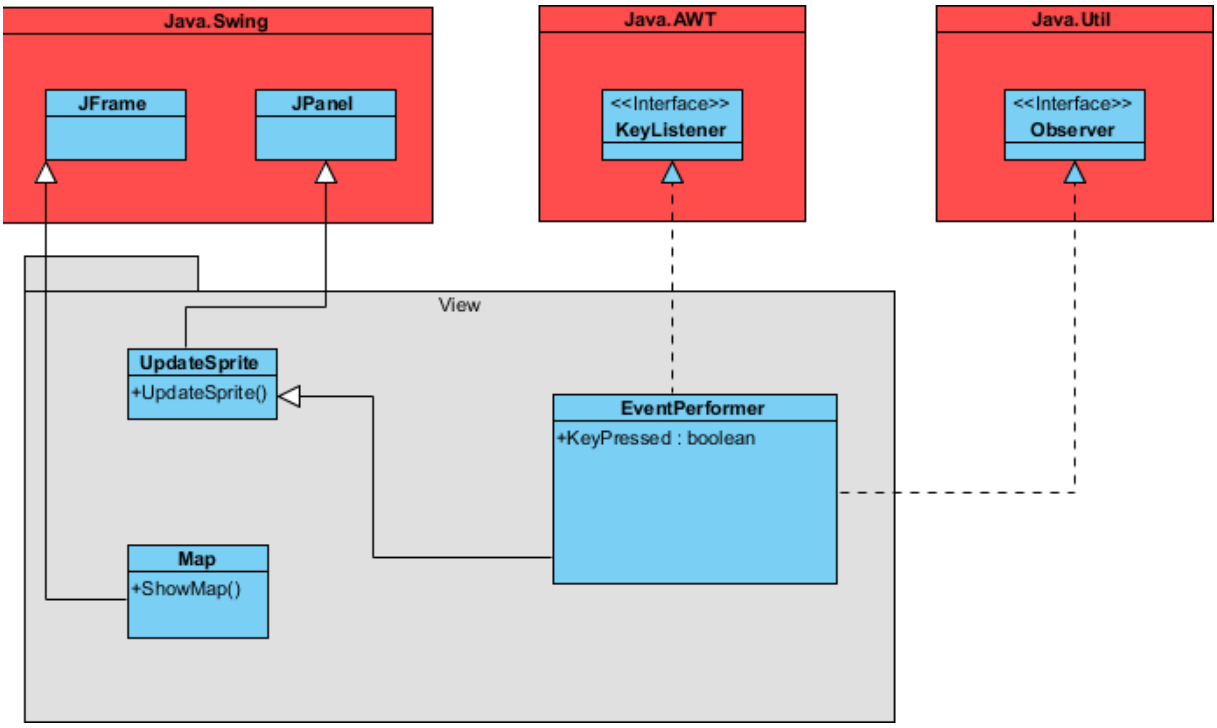
This is our Controller Class Diagram:



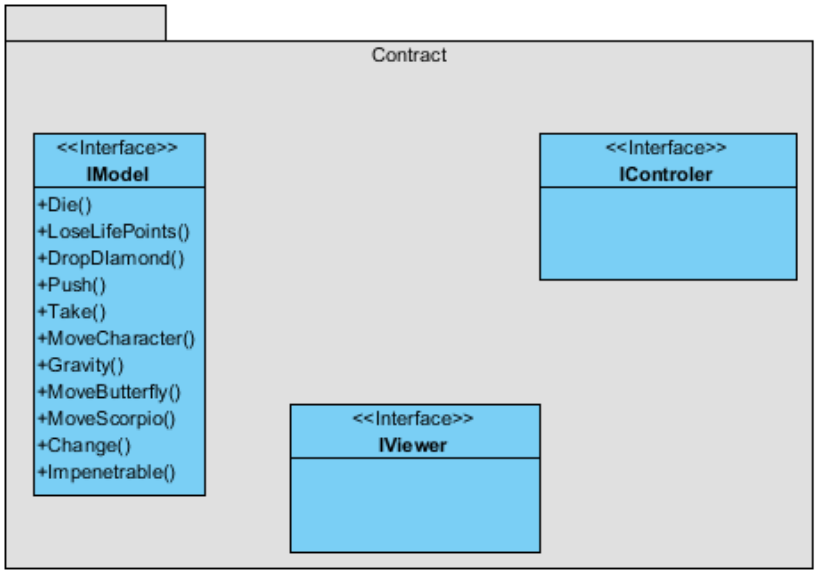
View Class Diagram:

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:



Maven Plugin

Maven is a build automation tool used primarily for Java projects.

This tool is a plugin that allow to make a several number of managing and developing tasks. It can also optimize the realized tasks in order to make a better order of manufacturing.

Maven also use a paradigm called Project Object Model, or POM, to describe a software project.

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 wrote a set of tests. These tests were at first supposed to be with the MVC architectural pattern, but, as we will see later, we were confront to several problem, particularly with the MVC pattern, and we didn't succeed to use it , that prevent us to reorganize them, and to write other 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());
	}

}
```

With these tests, we made a SureFire Report. This report is used with the Maven plugin, and he had to goals to compile and verify the test code.

At this part, these tests correspond at the test we used actually. There are all in the programming code we used to launch the game, and there are functionals.

PlayerTest:

This test allows us to know if the character moves well in the map

```java
package test;

import static org.junit.Assert.*;

public class PlayerTest {



    private Player player;

    /**
     * we instantiate a new player
     * @throws Exception
     */

    @Before
    public void setUp() throws Exception {
        this.player = new Player();
    }

    /**
     *   This test allows us to see if the character can move
     */

    @Test
    public void testMove() {
        this.player.move(1, 0);
        assertSame(2,this.player.getPosPlayerX());
    }

}
```

DiamandCountTest:

This test allows us to know if the assessors work perfectly.

```java

import static org.junit.Assert.*;

public class DiamondCountTest {

    @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 {
    }

    /**
     *  this test allows us to see if the getters and the setters of the diamonds taked by the character are functional
     */

    @Test
    public void testDiamond() {
        Pannel.setDiamondTaked(2);
        assertSame(2,Pannel.getDiamondTaked());
    }

    /**
     *   this test allows us to see if the getters and the setters the diamonds remaining on the map are functional
     */

    @Test
    public void testDiamondRemaining() {
        Pannel.setDiamondRemaining(8);
        assertSame(8,Pannel.getDiamondRemaining());
    }
```

EndTestGame:

This test allows us to know if the end game is functional

```java

public class EndTestGame {


    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
    }


    @AfterClass
    public static void tearDownAfterClass() throws Exception {
    }

    /**
     * we set the diamond count to 0 before the test
     * @throws Exception
     */

    @Before
    public void setUp() throws Exception {
        main.Pannel.diamondCount = 0;
    }



    @After
    public void tearDown() throws Exception {
    }

    /**
     * this test allows us to see if the end game is functional
     */

    @Test
    public void test() {
        final int expected = 0;
        assertEquals(expected, main.Pannel.endGame);
    }

}

```

Encountered Issues:

At the beginning, and during the project, we had difficulties to use the website GitHub. In fact, it was hard to set all of the member in the same project, and to manage the conflicts.

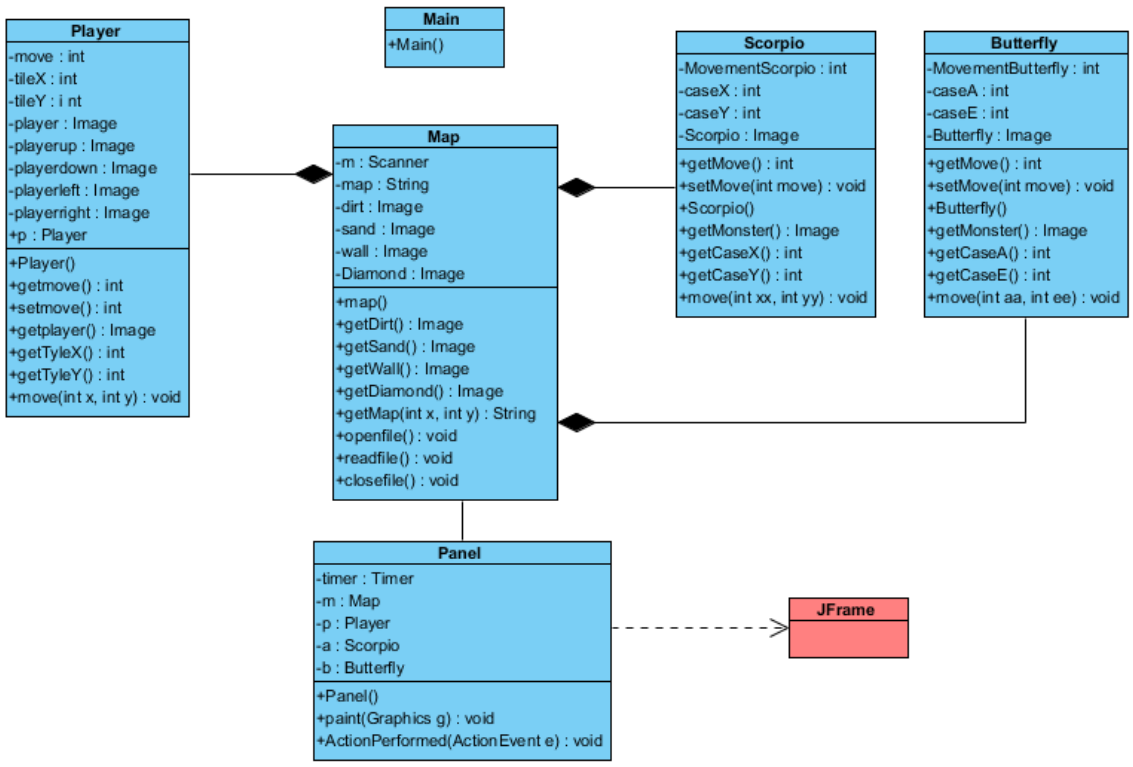
In the middle of the project, we realized that the Model-View-Controller architecture was too harder, and we didn't succeed to implement our game with this pattern.

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.

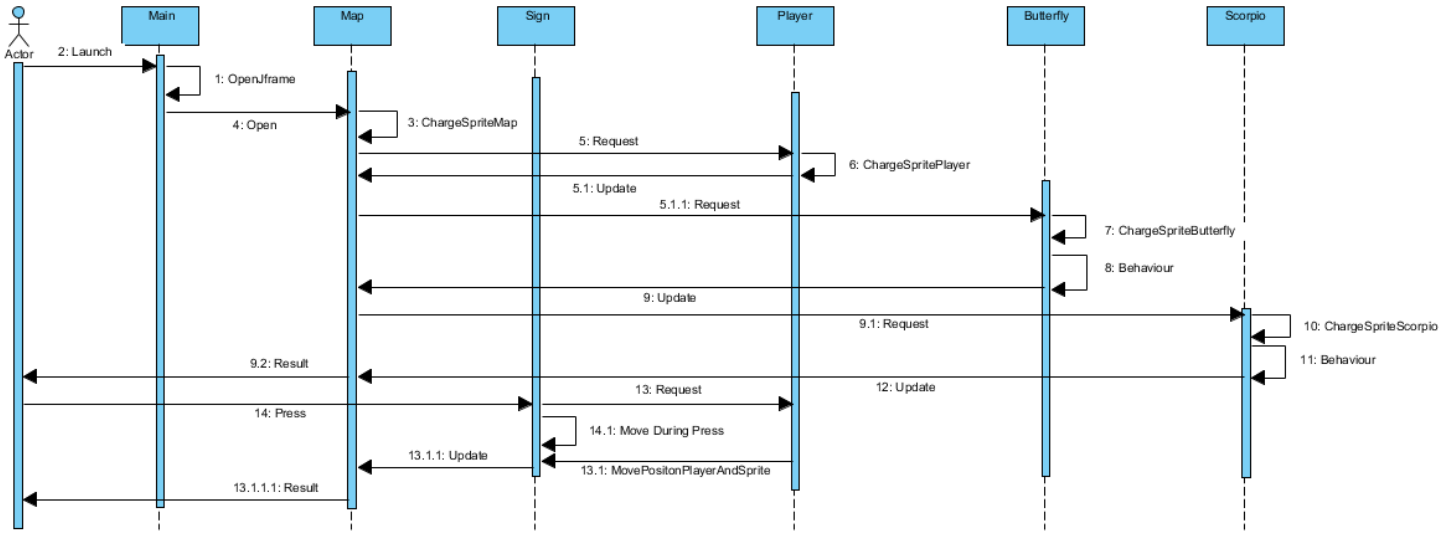
Then, we didn't use the MVC pattern, because we lack time, and we didn't succeed to set our programming code.

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:



Github:

We encountered some problems on Github.

First, we had problems getting commits from the Eclipse IDE. To counter this problem, we used branches. The branches have served us to be able to commit without having conflicts with the master. Therefore, our lines of code cannot be put into the master and so this will show us no rows.

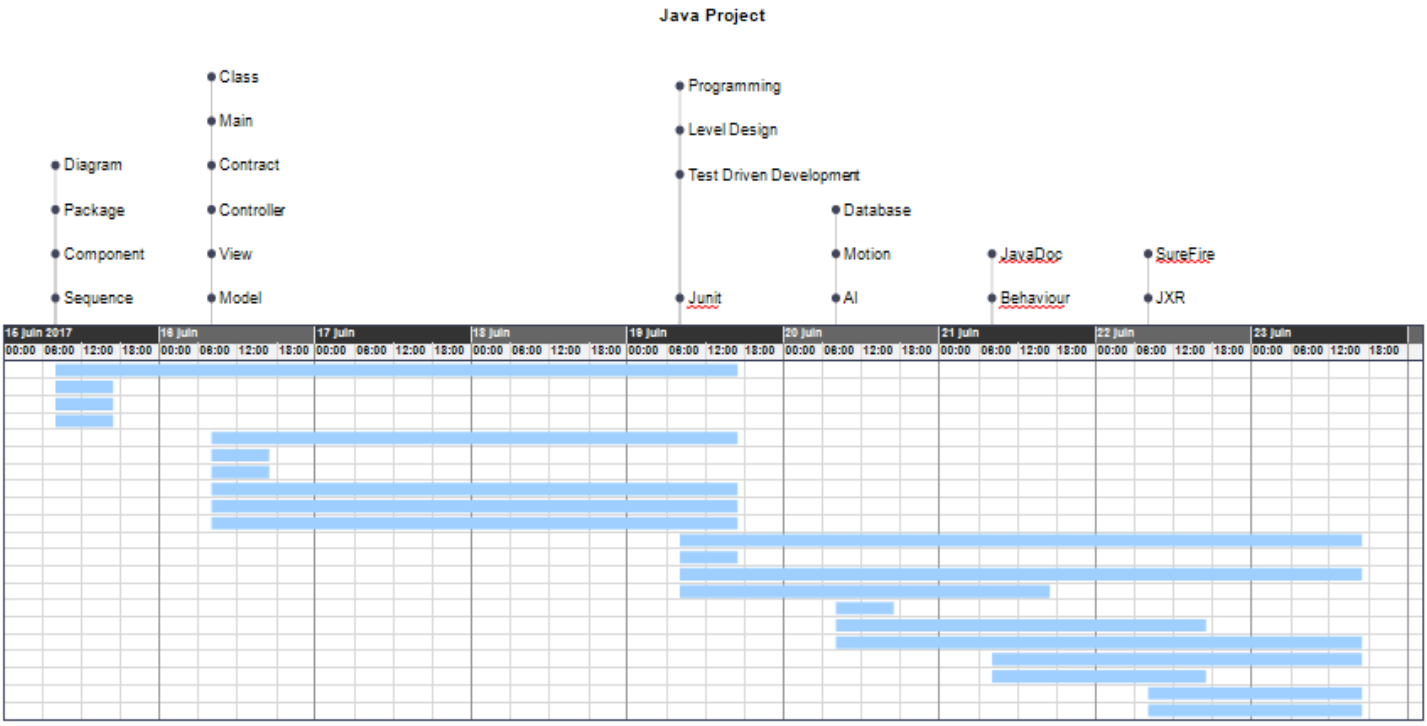
Planning

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.

Nom de branche	Durée	Début	Fin
Java Project	9 jours	14/06/2017	26/06/2017
Deliverables	9 jours	14/06/2017	26/06/2017
Oral Presentation	1 jour	26/06/2017	26/06/2017
⊟ Diagrams	3 jours	14/06/2017	16/06/2017
Package Diagram	1 jour	15/06/2017	15/06/2017
Class Diagrams	2 jours	15/06/2017	16/06/2017
Sequence Diagram	1 jour	16/06/2017	16/06/2017
Components Diagram	1 jour	16/06/2017	16/06/2017
⊟ DataBase	2 jours	20/06/2017	21/06/2017
Implementation	2 jours	20/06/2017	21/06/2017
Design	2 jours	20/06/2017	21/06/2017
Stored Procedures	2 jours	20/06/2017	21/06/2017
⊟ Programming	3 jours	21/06/2017	23/06/2017
Motion	1 jour	21/06/2017	21/06/2017
Level Design	1 jour	21/06/2017	21/06/2017
AI	1 jour	21/06/2017	21/06/2017
JavaDoc	1 jour	23/06/2017	23/06/2017
Behaviour	2 jours	22/06/2017	23/06/2017
⊟ Test Driven Development	4 jours	19/06/2017	22/06/2017
Junit	2 jours	19/06/2017	20/06/2017
SureFire	1 jour	22/06/2017	22/06/2017
JXR	1 jour	22/06/2017	22/06/2017

Effective Planning:

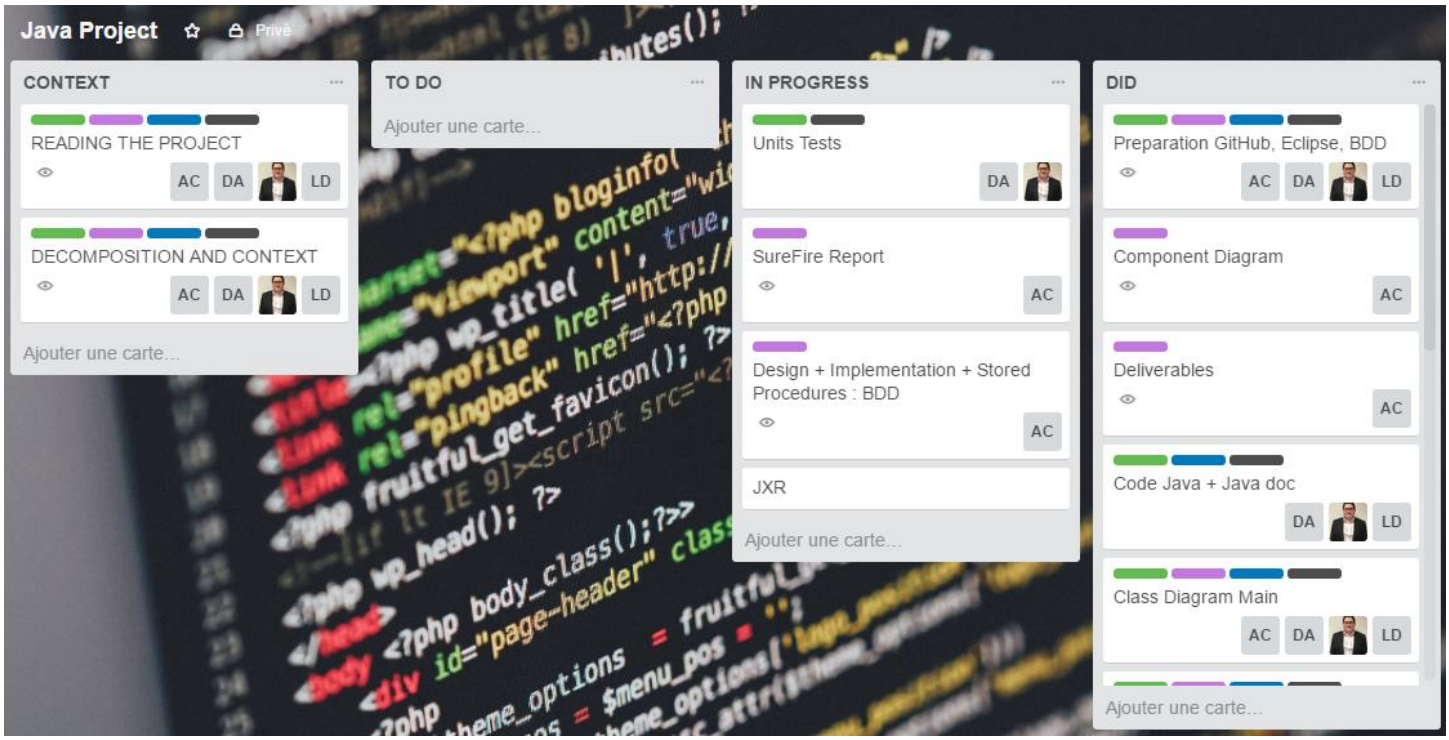


At the end of this project, we can have realized that the diagrams and the programming were the most important and the most difficult parts.

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

Trello:

In order to visualize at a better point the different steps of our project, we realized a Trello Project, on the Trello website. This let us to concentrate us and see what was the left steps.



Results of the project:

Group Result:

In general, this project went relatively smoothly. Unfortunately, we encountered a lot of problems throughout the duration of this one. This project allowed us to highlight the lessons that we learned during the prosits. The group was productive and allowed us to have a functional game. We will all keep a good memory of this project despite all the difficulty of it.

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: As the project leader, I was very scared to not be qualify for this place. But my teammates were very helpful with me, they support in order to dress a project report, and they also help me for the programming code. Despite the problems we met, this project was very nice.

Fritsch Florian: The project was very stressful because It was really complicated to succeed. We had problems of organization but we made a good progress in the project. The fact of having created a game was pleasant. The level of our group was very equal which made it possible to advance well while understanding everything.

Libessart Dimitri: For project of the end of year, I found this one very interesting due to the concept and needs asked, we have some problems with Github and the MVC but we managed to make a functional program and diagrams of our project, the group was very involved because the success of the group maybe a lack of time to finalize with the MVC of the project.