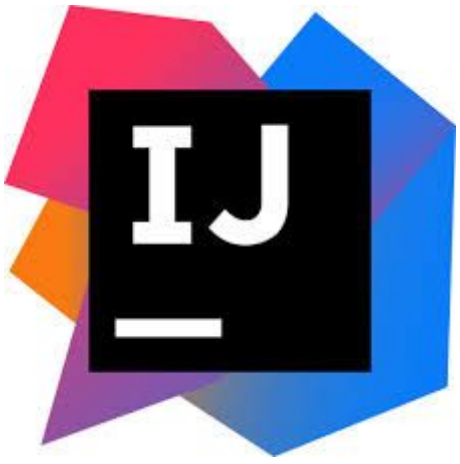
Dauphine | PSL
UNIVERSITÉ PARIS

Tutoriel final Agilité
Fusion projet ‘Planete Plat’

MEILLEUR PLAT DU SYSTÈME SOLAIRE

M. Michel ZAM



Contributeurs :

- **Said AMRANI**
- **Ibrahim DIDI**
- **Christian RAVOAVY HARIMAMPIANINA**
- **Cylia SADI OUFELLA**

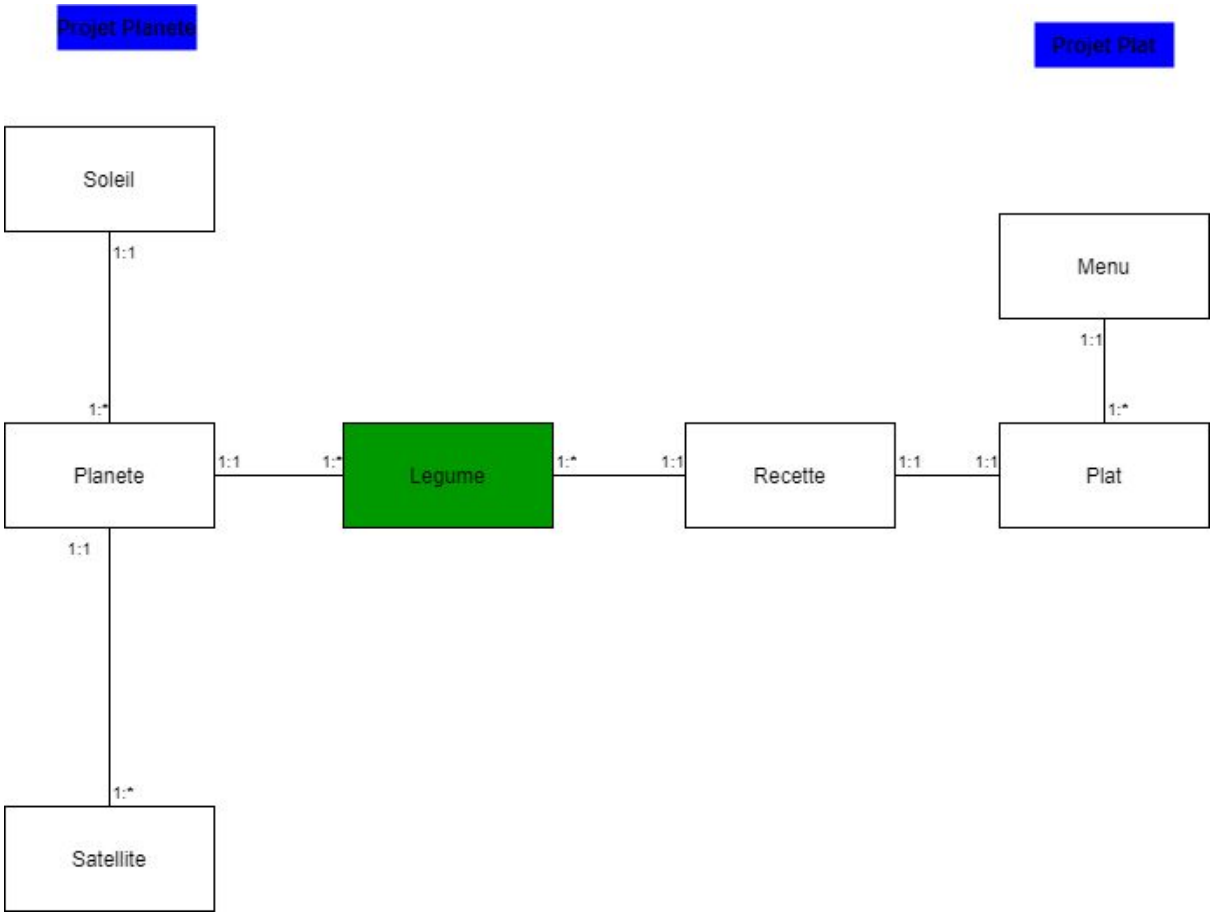
I. Introduction de la fusion des deux mondes

Quand deux monde différents se confrontent , quelquefois il n'est pas facile de trouver des points communs entre eux. Sauf que, ces deux monde ont décidé d'apporter leurs soutient à toute l'humanité en cette période difficile, car oui, en ce moment même, nous sommes en 2020, une grande pandémie se répand sur la planete terre et fait de terribles dégâts humains d'où ces habitants se retrouvent coincés et ne retrouvent plus leurs habitudes.

Le monde des planètes et le monde des recettes et plats ont décidé d'apporter leurs soutiens à tous les habitants de la terre, par une organisation inédite, un concours ou les plantes représentants les humains et réaliseront le grand concours du :

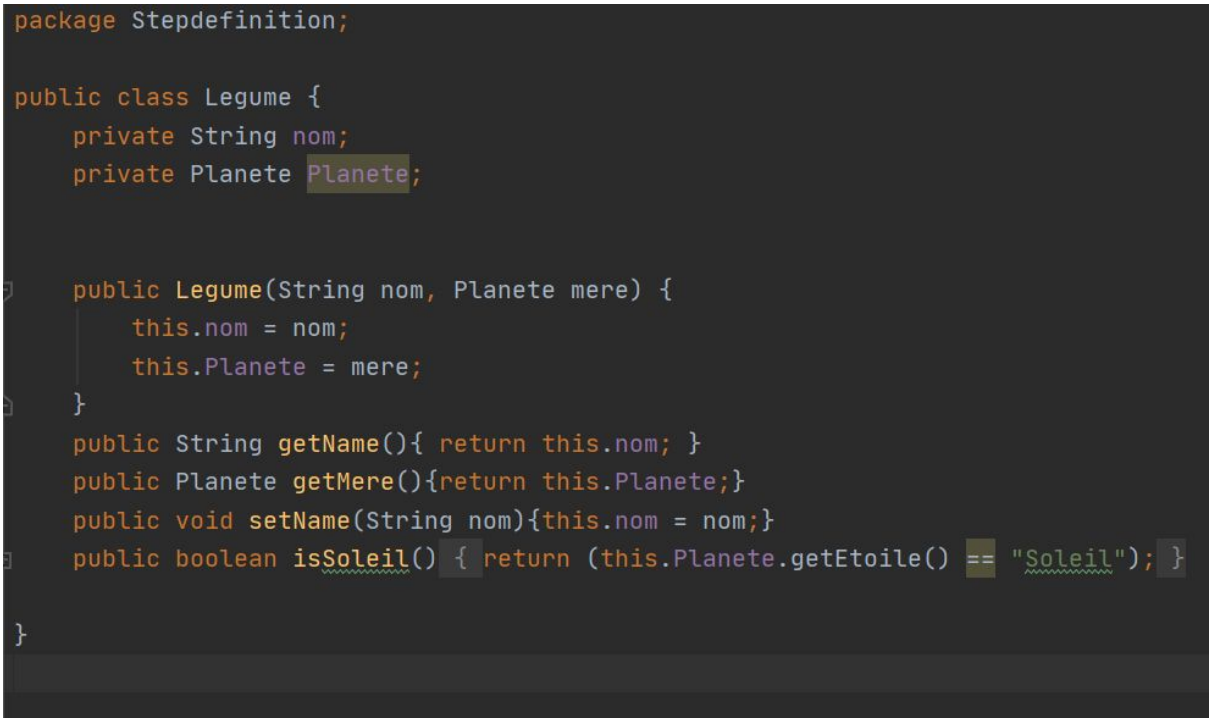
MEILLEUR PLAT DU SYSTÈME SOLAIRE

Malheureusement, la terre qui excelle dans ce domaine, ne pourra participer à ce concours à cause de la pandémie. Mais la bonne NOUVELLE ! toutes les planètes du système solaire se sont mises à cultiver des légumes à fin de réaliser des plats savoureux !



II. Classe LEGUME :

Voici notre classe légume qui appartiendra à une planète. Un légume n'apparaît que sur une planète



```java
package Stepdefinition;

public class Legume {
    private String nom;
    private Planete Planete;


    public Legume(String nom, Planete mere) {
        this.nom = nom;
        this.Planete = mere;
    }
    public String getName(){ return this.nom; }
    public Planete getMere(){return this.Planete;}
    public void setName(String nom){this.nom = nom;}
    public boolean isSoleil() { return (this.Planete.getEtoile() == "Soleil"); }

}
```

Testons si nos légumes sont bien des légumes, faisons des tests unitaires :

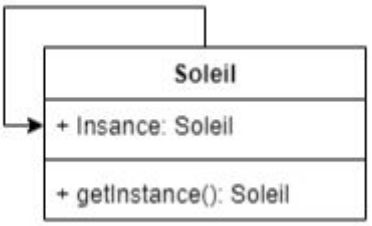
```java
package Stepdefinition;
import ...

public class LegumeTest {

    Legume l1 = new Legume( nom: "Patate",new Planete( name: "Avril", age: 1));
    @Test
    public void testLegume() {
        String nom  ="patate";
        Legume l = new Legume(nom,new Planete( name: "Avril", age: 1));
        assertEquals( expected: "patate",l.getName());
        assertEquals( expected: "Avril",l.getMere().getName());
    }
    @Test
    public void TestgetName() { assertEquals( expected: "Patate",l1.getName()); }
    @Test
    public void TestgetMere(){assertEquals( expected: "Avril",l1.getMere().getName());}
    @Test
    public void TestsetName(){
        l1.setName("carotte");
        assertEquals( expected: "carotte",l1.getName());
    }
    @Test
    public void TestisSoleil() { assertEquals( expected: true,l1.isSoleil()); }
}
```

III. Design Pattern Singleton : classes et tests unitaires

Donc voilà, notre potager de légume a besoin d'un soleil pour pousser. Mais comme nous le savons très bien, notre système solaire ne contient qu'un seul et unique soleil. Nous ne pouvons en créer qu'un seul dans la logique des termes. De ce fait le design pattern Singleton est bel et bien notre solution pour cela. Donc si quelqu'un se prend pour un dieu et essaie de modifier la logique de notre système solaire en créant un autre soleil, il ne pourra pas. La nouvelle création renverra tout simplement le soleil qui existe déjà grâce à la méthode getInstance();




```java
public class Soleil {
    private static  Soleil instance;
    private static String etoile;

    private  Soleil() { }

    public static Soleil getInstance()
    {
        if (instance == null){
            instance = new Soleil();
            etoile="Soleil";
        }
        return instance;
    }

    @Override
    public String toString()
    {
        return String.format("Je suis l'etoile : %s", etoile);
    }
    public String getEtoile() {return etoile; }
}
```

Testons maintenant notre Singleton soleil, en essayant de créer 2 soleils :

```java
public class SoleilTest {
    Soleil s1 = Soleil.getInstance();
    Soleil s2 = Soleil.getInstance();

    @Test
    public  void TestSingleton(){
        assertEquals(s1,s2);
    }

}
```

IV. Design Pattern Strategy : classes et tests unitaires

Dans les temps normal la planète terre organise plusieurs concours que ce soit des concours de beauté, des concours sportifs comme ça peut être aussi des concours de cuisine. Comme notre malheureuse terre est frappé par le Covid-19 donc il pourra pas créer des concours ni de faire une activité de cuisine, Pour cela il y a trois planète qui ont profité de la pandémie pour prendre la main à la terre et de faire lancer le concours de la cuisine, chacune de ces trois planete va présenter une recette qui contiendra plusieurs légumes, pour permettre à ces planètes de créer leurs recette dans notre application, on a mis en place le design pattern Strategy qui permet la création de ces recettes.
on sait que toutes les autre planètes vont être jalouses dans les années à venir et voudront rejoindre le concours pour cela une évolution de notre application est très envisageable c'est pour celà qu'on a mis en place ce pattern qui permet l'ajout d'autre planète facilement et c'est le plus adapté dans ce genre de situation (pour éviter les if...else)
la figure suivante montre le diagramme de classe de notre design pattern strategy

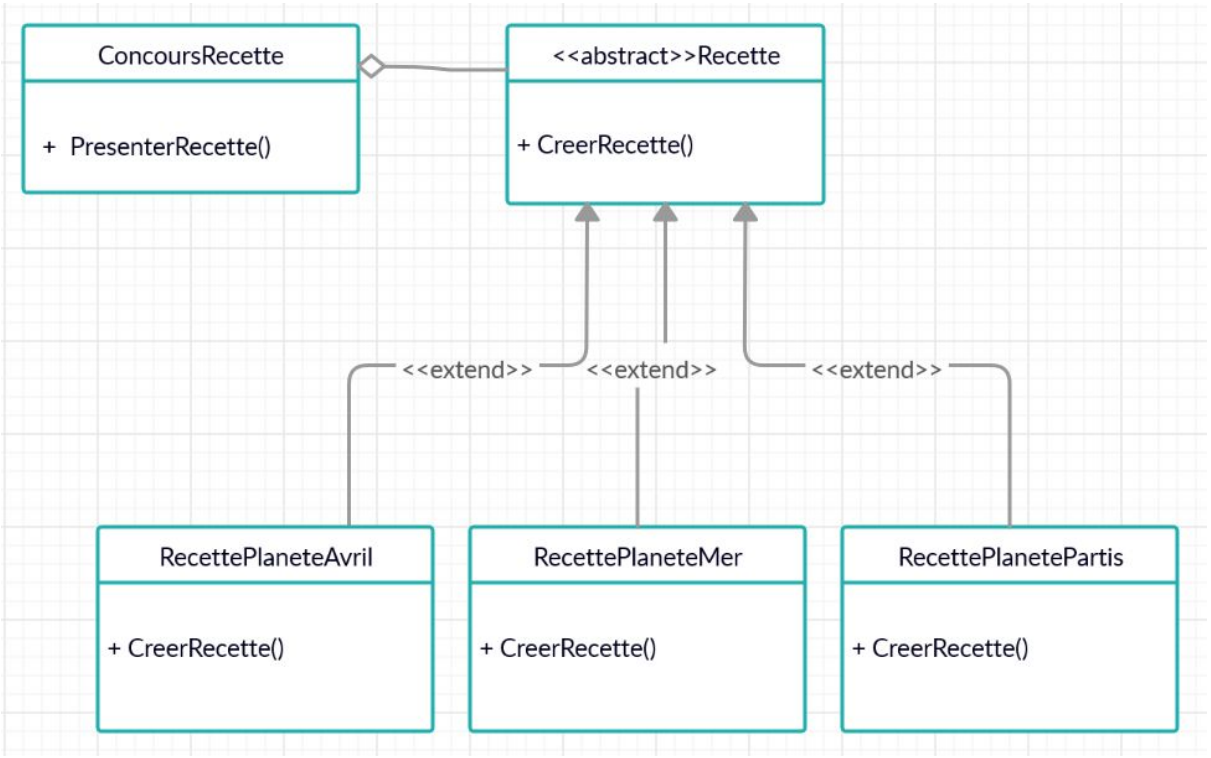


Diagramme de classe pour la création des recettes (design pattern strategy)

Maintenant comme nous l'avons dit, chaque planète aura sa propre recette. comme chaque planète évolue de manière indépendante par rapport à toutes les autres, au niveau du nombre de légumes etc… Il est donc plus propre de traiter la recette de chaque planète indépendamment. Pour cela nous avons annoncé le design pattern Strategy. Notre classe devient donc une classe abstraite comme suit, et nous ajoutons dedans la liste des légumes qui composent la recette ainsi que la manière dont la recette est créée.

```
package Stepdefinition;
import ...

public abstract class Recette
{
    // variables d'instance - remplacez l'exemple qui suit par le vôtre
    private int nbIng;

    protected List<Legume> legumeList;
    abstract public String creerRecette();
```

Nous aurons ensuite pour chaque planète sa propre classe de recette qui héritera de la classe recette. Nous lui ajoutons ses légumes correspondant. Dans notre système solaire nous avons 3 planètes dont la planète Avril, la planète Mer ainsi que la planète Partis.

RecettePlaneteAvril pour la planète Avril dont les légumes sont PommeAvril et PatateAvril :

```java
package Stepdefinition;


import java.util.ArrayList;

public class RecettePlaneteAvril extends Recette{
    @Override
    public String creerRecette() {
        Legume l1 = new Legume( nom: "PommeAvril", new Planete( name: "Avril",  age: 10000));
        Legume l2 = new Legume( nom: "PatataAvril", new Planete( name: "Avril",  age: 10000));
        legumeList = new ArrayList<Legume>();
        legumeList.add(l1);
        legumeList.add(l2);
        return ("la recette de la Planete Avril est composee de PatateAvril et PommeAvril");
    }

}
```

RecettePlaneteMer pour la planète Mer dont les légumes sont PommeMer et PatateMer:

```java
package Stepdefinition;

import java.util.ArrayList;

public class RecettePlaneteMer extends Recette {

    @Override
    public String creerRecette() {
        Legume l1 = new Legume( nom: "PommeMer", new Planete( name: "Mer",  age: 10000));
        Legume l2 = new Legume( nom: "PatataMer", new Planete( name: "Mer",  age: 10000));
        legumeList = new ArrayList<Legume>();
        legumeList.add(l1);
        legumeList.add(l2);
        return("la recette de la Planete Mer est composee de PatateMer et PommeMer");
    }
}
```

RecettePlanetePartis pour la planète Avril dont les légumes sont PommePartis et PatatePartis :

```java
package Stepdefinition;

import java.util.ArrayList;

public class RecettePlanetePartis extends Recette{

    @Override
    public String creerRecette() {
        Legume l1 = new Legume( nom: "PommeAvril", new Planete( name: "Partis",  age: 10000));
        Legume l2 = new Legume( nom: "PatataAvril", new Planete( name: "Partis",  age: 10000));
        legumeList = new ArrayList<Legume>();
        legumeList.add(l1);
        legumeList.add(l2);
        return("la recette de la Planete Partis est composee de PatatePartis et PommePartis");
    }
}
```

Testons maintenant unitairement que chacun de nos planètes peut bien créer ses propres recettes.

```java
package Stepdefinition;

import ...

public class RecettePlaneteTest {
    Recette rAvril = new RecettePlaneteAvril();
    Recette rMer = new RecettePlaneteMer();
    Recette rPartis = new RecettePlanetePartis();

    @Test
    public void TestRp(){
        assertEquals( expected: "la recette de la Planete Avril est composee de" +
                " PatateAvril et PommeAvril",rAvril.creerRecette());
        assertEquals( expected: "la recette de la Planete Mer est composee de " +
                "PatateMer et PommeMer",rMer.creerRecette());
        assertEquals( expected: "la recette de la Planete Partis est composee de " +
                "PatatePartis et PommePartis",rPartis.creerRecette());
    }
}
```

Cependant, vous vous poserez peut-être la question du pourquoi et quand utiliser ces recettes par planète ? Eh ben ce sera notre contexte dans le designe Pattern Strategy. Notre contexte sera ici notre concours de recette pour les planètes.

```java
package Stepdefinition;

public class ConcoursRecette {
    protected Recette recette;

    public String presenterRecette() { return recette.creerRecette(); }
    public void setRecette(Recette recipe) { recette = recipe; }
}
```

Finalement faisons un petit test de présentation de sa recette pour une planète quelconque de notre système solaire. La planète Avril par exemple… :

```java
package Stepdefinition;
import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class ConcoursRecetteTest {

    @Test
    public void testcreerRecette(){
        ConcoursRecette cr = new ConcoursRecette();
        cr.setRecette(new RecettePlaneteAvril());
        assertEquals( expected: "la recette de la Planete Avril est " +
                "composee de PatateAvril et PommeAvril", cr.presenterRecette());
    }

}
```

Et voilà, notre design Pattern Strategy est mis en place.

V. User Stories et tests fonctionnels

US 001 : Donc faisons un petit feed back en arrière.Comme nous avons dit, faire pousser les légumes nécessite l'existence d'un soleil. Donc l'information soleil est logiquement contenue dans le légume. Ceci constitue donc notre User Story dans notre fichier .feature (cf partie 3)

```
Feature: SL_000 creation du systeme solaire

  Scenario: faire pousser des legumes
    Given la pousse-legume necessite les rayons du soleil
    When la creation du soleil fait pousser les legumes
    Then l'etoile de la planete du legume est Soleil
```

Générons nos étapes correspondant à notre histoire de création de soleil en faisant alt+entrée ou en compilant :

```
package Stepdefinition;

import ...

public class SoleilSteps {
    Legume l;
    boolean isSoleil;

    @Given("la pousse-legume necessite les rayons du soleil")
    public void la_pousse_legume_necessite_les_rayons_du_soleil() {
        l = new Legume( nom: "carotteAvril",new Planete( name: "Avril", age: 10000));
    }

    @When("la creation du soleil fait pousser les legumes")
    public void la_creation_du_soleil_fait_pousser_les_legumes() {
        la_pousse_legume_necessite_les_rayons_du_soleil();
        isSoleil = l.isSoleil();
    }

    @Then("l'etoile de la planete du legume est Soleil")
    public void l_etoile_de_la_planete_du_legume_est_Soleil() {
        la_creation_du_soleil_fait_pousser_les_legumes();
        assertEquals( expected: true,isSoleil);
    }
```

Il nous suffit maintenant de tester que notre histoire tient la route avec notre TestRun.

```java
package Stepdefinition;

import ...
@RunWith(Cucumber.class)
@CucumberOptions(
        features = "C://Users//CSadiO//IdeaProjects//UNION PROJ//cucumber//Features" +
                "//ModifRcpPlanete.feature")

public class SoleilTestRun {
```

US002 : Vous vous demanderez peut-être : “mais si on découvrait un nouveau légume dans une planète quelconque, sa recette pourrait changer ?”
Nous vous répondons oui, nous pouvons traiter le cas en tant que User Story :)

```gherkin
Feature: CN_000 lancement du concours de la meilleure recette

  Scenario: Changement recette d'une planete
    Given une recette est composé des legumes de la planete
    When un nouveau legume apparait dans une planete
    Then la recette change
```

Créons donc nos étapes de changement de recette:

```java
package Stepdefinition;
import ...
public class ModifRcpPlaneteSteps {
    Recette r;
    Legume nvlegume ;
    String nomLegume;

    @Given("une recette est composé des legumes de la planete")
    public void une_recette_est_composé_des_legumes_de_la_planete() {
        r = new RecettePlaneteMer();
        r.creerRecette();
    }

    @When("un nouveau legume apparait dans une planete")
    public void un_nouveau_legume_apparait_dans_une_planete() {
        une_recette_est_composé_des_legumes_de_la_planete();
        nvlegume = new Legume( nom: "haricot",new Planete( name: "Mer", age: 10000));
        nomLegume = r.addLegume(nvlegume);
    }

    @Then("la recette change")
    public void la_recette_change() {
        un_nouveau_legume_apparait_dans_une_planete();
        assertEquals( expected: "haricot",nomLegume);
    }
```

Et testons notre Use Case.

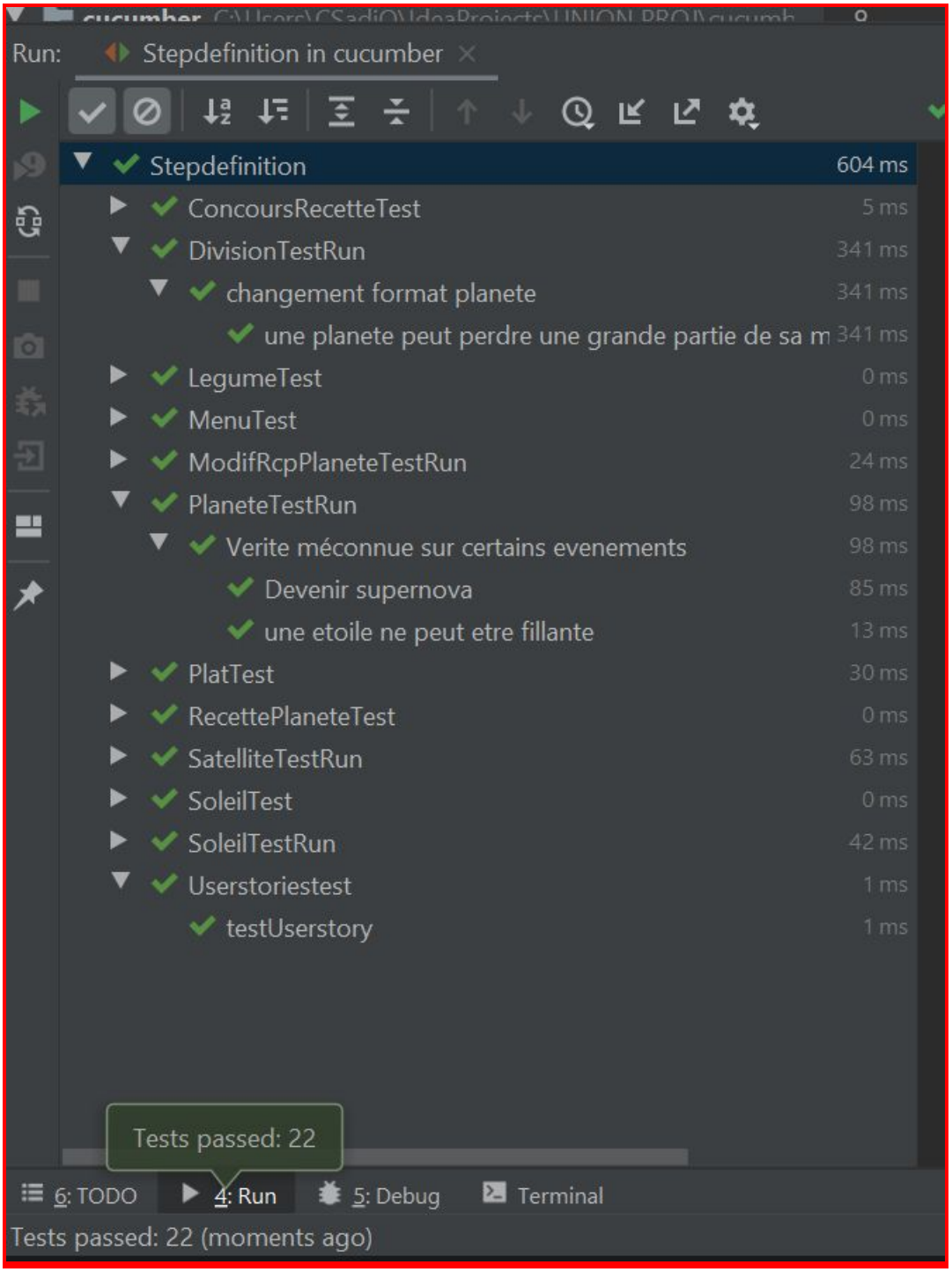
```java
package Stepdefinition;
import ...
@RunWith(Cucumber.class)
@CucumberOptions(
        features = "C://Users//CSadiO//IdeaProjects//UNION PROJ//cucumber//" +
                "Features//Soleil.feature")

public class ModifRcpPlaneteTestRun {
}
```

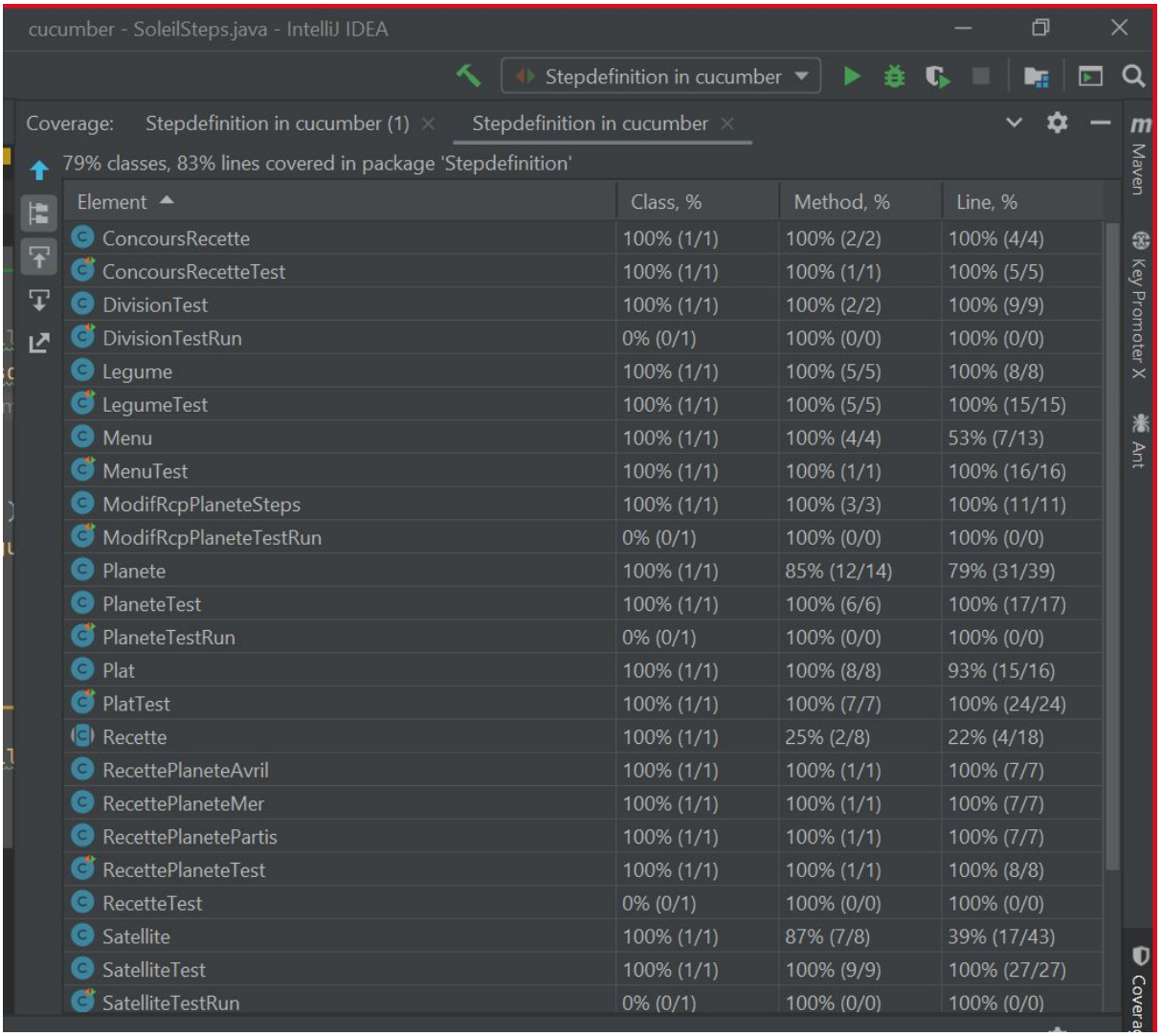
VI. Test global et Couverture du Code

Enfin nous allons tout tester globalement pour voir si nos 2 mondes ont pu bien fusionner. Faisons un test global avec Coverage. (cf Partie 2)

Nous n'avons que du vert, toutes nos tests ont réussis.



Et ben voilà, nos 2 mondes sont compatibles à 79% pour les classes et 83% des lignes de code. Soit globalement une mention très bien :)



Ce document ainsi que l'integralité du projet se trouvent sur le lien suivant par la contribution des collaborateurs de ce document :
https://github.com/cylrs/Agilite-AMRANI-DIDI-RAVOAVY-SADI/