RAPPORT PROJET AGILE

HARB CEDRIC
HOUMEL IDIR
MESSALI NASSIM
SIMONIAN CHARLY

Table des matières

I). Introduction et mise en contexte

Maintenant que vous avez fini de créer votre projet pour modéliser votre première voiture, vous avez décidé de prendre le volant pour faire une promenade. Même si votre mère vous as dit : "Fais attention mon chéri tu es encore un jeune conducteur !!", toi tu as décidé de faire comme dans ton film préféré, Fast & Furious...

Malheureusement tu n'es pas Vin Diesel et tu as fini dans le fossé. Bon tout va bien pour toi heureusement mais tu dois quand même faire un petit tour chez le médecin pour te faire soigner. Dans ton petit malheur, tu te rends compte qu'une autre des personnes qui attend sa consultation est comme toi il veut s'initier à l'informatique. Comble de l'ironie cette personne a suivi un tuto similaire à celui que tu as suivi mais lui sur les patients et les médecins. Vous décidez donc de faire un projet pour représenter ta triste histoire qui utilisera vos deux projets. Nous qui vous avons accompagné dans vos premiers pas en programmation agile allons vous aider à poursuivre votre apprentissage grâce à ce nouveau projet !!!

II). Mise en place du projet :

Lors de votre précédent tutoriel vous aviez réalisé votre projet seul. Ici vous n'êtes plus seul vous devez donc être d'autant plus rigoureux sur votre méthode de travail et sur l'organisation de votre projet. Si vous avez suivi avec soin le précédent tutoriel vous verrez qu'il n'y a pas de gros changement juste de petits ajustements.

Avant de commencer essayons d'utiliser un outil pour nous permettre de planifier et suivre l'avancement de notre projet. Nous vous conseillons d'utiliser trello car il est l'un des plus courant pour ce genre de tâche et gratuit mais vous pouvez sans aucun souci utiliser autre chose.

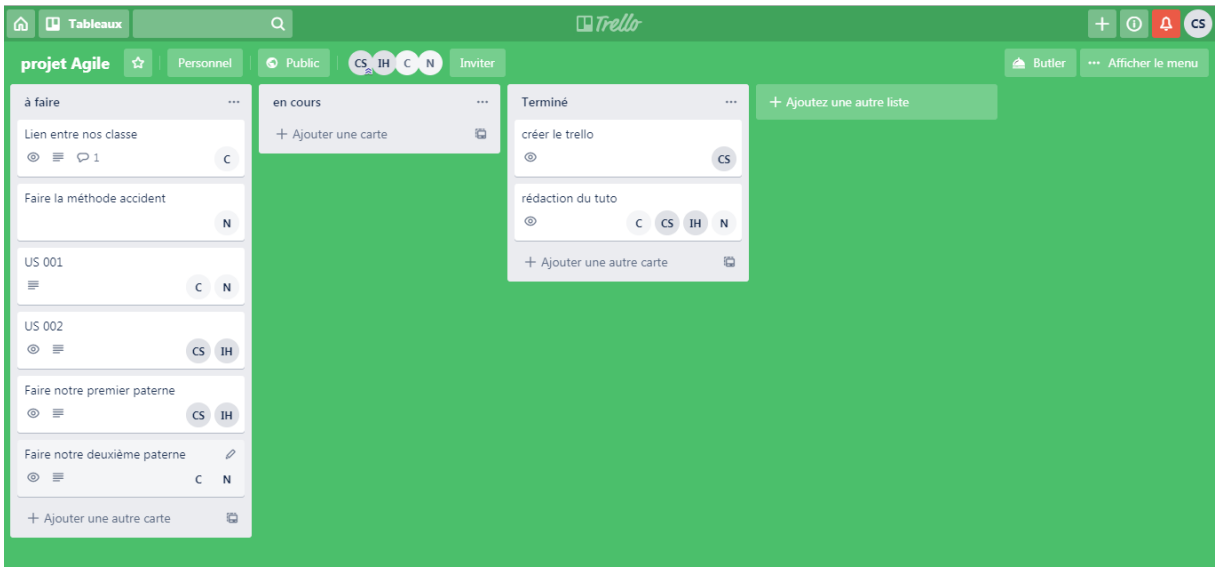
1). Votre premier *Trello*

Tout d'abord créez votre compte sur le site et suivez le mini tuto au début pour créer votre projet. Vous remarquerez que vous avez trois grandes sections (à faire / en cours /Terminé). Avant de créer vos différentes tâches pensez à ajouter le ou les membres de votre groupe. Ensuite vous pouvez ajouter de nouvelles tâches dans la partie travail à faire en y mettant une description et en précisant la ou les personnes qui sont responsables de chaque tâche (il y a aussi un espace commentaire qui peut avoir de multiples utilités comme poser une question, expliquer une complication ...). Pour changer l'état d'une tâche vous avez juste à cliquer dessus et à la faire glisser dans colonne suivante.

Je vous vois arriver là vous vous dites que je vous présente un truc rébarbatif qui ne vous servira pas car à deux vous pouvez vous téléphonez pour faire un point et voir ce que vous allez faire demain. Mais croyez-moi planifier son travail est une phase très importante et avoir un outil comme *Trello* qui vous permet de suivre l'avancement de chaque partie du projet et ainsi de

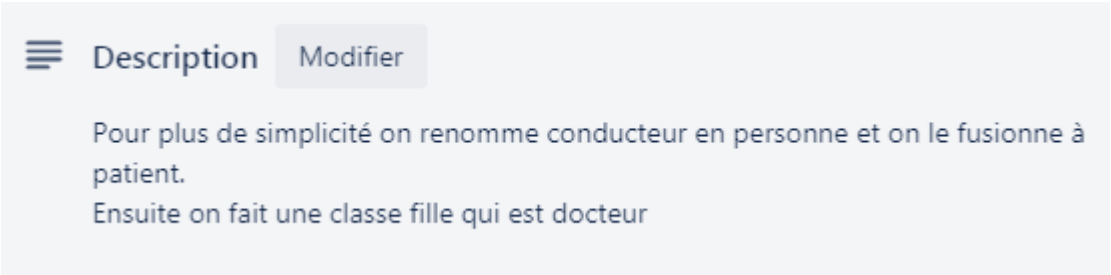
voir s’il y a des endroits où vous n'arrivez pas à progresser, tout ça sans avoir à faire une rencontre entre tous les membre est très pratique.

Donc regardons comment nous avons choisi de planifier notre projet. Ici, nous avons travaillé à 4 mais mis à part la répartition des tâches il ne devrait pas y avoir trop de changement.



Nous Pouvons voir que nous avons d'abord choisi de faire un lien entre nos classes venant des deux projets, si vous cliquez sur cette tâche vous verrez sa description pour voir plus précisément ce qu’il faut faire.

2). Fusion des deux projets :



Pour réunir vos deux projets et atteindre votre nouvel objectif, vous devez faire fusionner deux classes (conducteur et patient) et faire hériter médecin de cette nouvelle classe. Normalement faire ce genre de changement est très déconseillé car extrêmement compliqué et peut être la cause de nombreux bugs et problèmes de développement, mais dans notre cas au vu de la simplicité du problème et du fait que nous avons des classes très petites, aucune classe n'a de méthode similaire, rendant ainsi cette fusion très simple. De plus, cela nous évitera des castes futures. Allez mettons en pratique ce que nous venons d’expliquer ! Ainsi, sur votre branche develop de votre projet voiture, renommez votre classe 'conducteur' en 'personne' (n'oubliez pas que dans vos fichiers de test vous devez aussi la renommer). Ensuite vous devez aussi ajouter

tous les attributs et les méthodes de la classe 'patient' à votre classe 'personne', pensez ici aussi à faire les modifications pour les classes de tests qui correspondent à la classe patient. Enfin pour votre classe 'médecin' elle ne change presque pas à part qu'elle hérite de 'personne', donc normalement votre classe 'médecin' ne contient plus qu'un attribut qui est ListPatient. Vous pouvez maintenant mettre votre code sur Git Hub une fois que vous aurez vérifié que tous vos tests et users stories initiales sont correctes.

Maintenant que la fusion des deux projets réalisés et que tu as bien planifié ton travail tu vas enfin pouvoir ajouter au modèle une méthode qui va modéliser ta mésaventure. Dans la classe voiture tu vas donc faire une méthode ***accident***. Cette méthode va te permettre de te signaler comme un patient au près d'un médecin en lui signalant ton nombre de point de vie perdu. Pour cette méthode nous vous laissons votre indépendance on ne vous aidera pas mais elle reste très proche de celle de tomber malade donc inspirez-vous en. Si vous le désirer vous pouvez tout à fait rajouter des fonctionnalités comme casser la voiture, la seule limite c'est votre imagination...

III). Amélioration avec des designs patterns

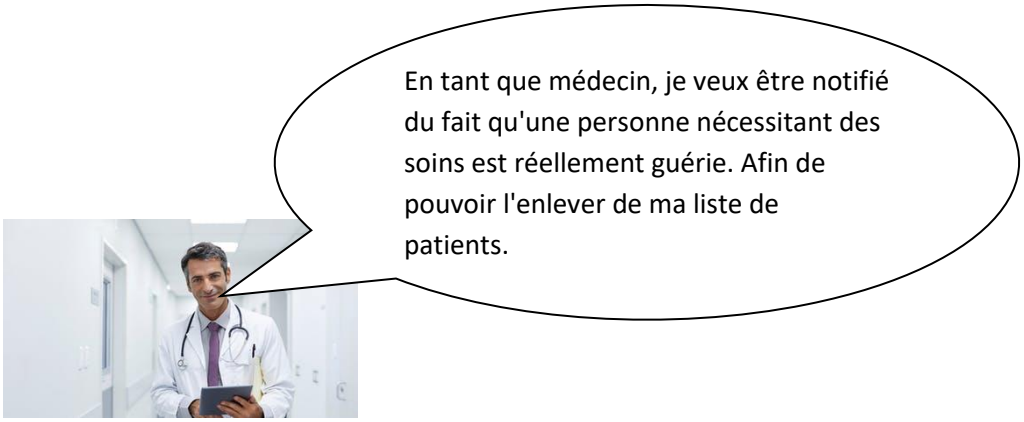
Tout est fait pour ce qui est de la modélisation vous pouvez être fier de vous ! Votre projet est complet et fonctionnel ! Mais pour rester dans les bonnes pratiques que nous essayons de vous transmettre depuis le début de ce tuto nous allons vous demandez quelques dernières modifications. Celle-ci n'auront pas pour but de vous faire rajouter d'importantes fonctionnalités à votre projet mais elles auront pour objectif de rendre votre code plus propre et de vous initier à deux designs patterns.

En informatique il y a de nombreux problème très récurrents et génériques auxquels il existe de nombreuses réponses. Pour résoudre ces derniers il existe des designs patterns qui sont des solutions de conception pour ces problèmes qui sont robustes et générique. Ils sont utilisés pour trois raisons principales à savoir :

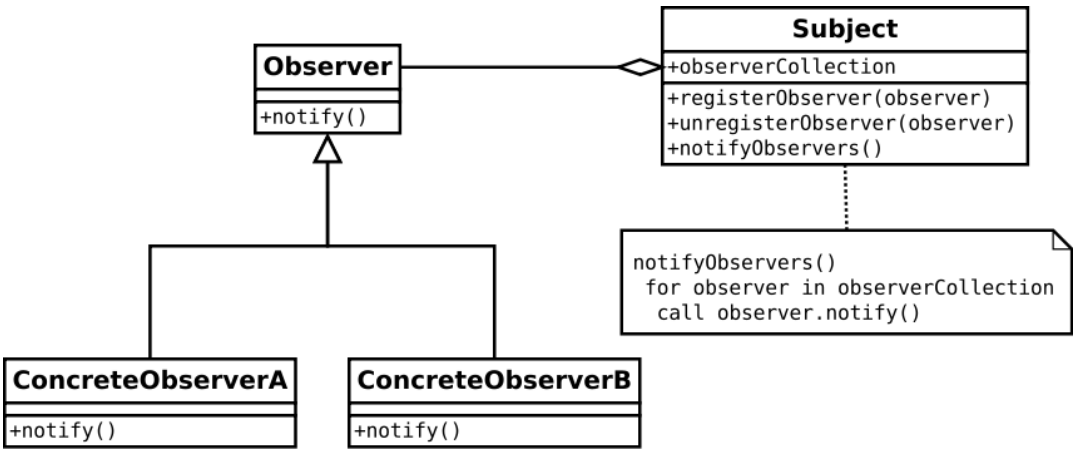
- Accélérer le processus de développement en fournissant des paradigmes de développement éprouvés.
- Anticiper des problématiques qui peuvent ne devenir visibles que plus tard dans la mise en œuvre.
- Améliorer la lisibilité du code en fournissant une standardisation.

1). Design Pattern Observer :

Ainsi, le premier besoin que l'on a pu discerner est consigné dans notre première user story qui peut s'énoncée de cette façon :



Ça tombe bien ! Ce besoin peut être modélisé à l'aide d'un design pattern particulier nommé *Observer*. Ce dernier est utilisé en programmation pour envoyer un signal à des modules qui jouent le rôle d'observateur (dans notre cas un médecin). En cas de notification, les observateurs effectuent alors l'action adéquate en fonction des informations qui parviennent depuis les modules qu'ils observent appelés les « observables » (dans notre cas les patients). Voici le schéma général de ce patron de conception :



(Pour réaliser proprement ceci vous allez devoir utiliser des interfaces. Une interface définit le comportement que devra avoir une classe, pour ce faire elle définit les méthodes et attribut de la classe, ce que devras faire chaque méthode mais ne précise pas comment. Le comment se définie dans la classe qui implémente cette interface. On dit qu'une classe implémente une interface)

Si nous faisons l'analogie avec notre situation, il faudrait que la classe Personne qui contient la donnée qui sera régulièrement modifiée implémente *Observable* (dont le code est présenté ci-contre) :

```java
package blueJ;

@SuppressWarnings("hiding")
public interface Observable<Personne> {
    public void addObserver(Observer<Medecin> observer);
    public void notifyObservers();
}
```

La classe Medecin quant à elle qui sera notifiée par la modification de l'objet Personne implémente *Observer* :

```java
package blueJ;

@SuppressWarnings("hiding")
public interface Observer<Medecin> {
    public void update(Personne personne);
}
```

Dans la classe Personne qui doit déclencher un évènement (à savoir que la personne n'est plus malade autrement dit ses points de vies sont égaux à 100), on ajoute:

- En attribut : une liste d'Observateurs
- Une méthode permettant d'ajouter un Observateur dans la liste
- Une méthode permettant d'envoyer un signal a tous ses obsevateurs.

Ceci comme illustré à travers ces lignes de codes :

```java
public class Personne implements Observable<Personne> {

        public Voiture voiture;
        private String nom;
        private String prenom;
        private int pointDeVie = 100;
        private boolean malade = false;
        private Medecin medecinTraitant;
        private List<Observer<Medecin>> observers = new ArrayList<Observer<Medecin>>();
```

```
public void setPointDeVie(int p){
    this.pointDeVie = p;

    if(this.pointDeVie == 100){
        this.setMalade(false);
        this.medecinTraitant.notifyObservers();
    }
}

public void addObserver(Observer<Medecin> observer) {
    observers.add(observer);

}

public void notifyObservers() {
    for (Observer<Medecin> observer : observers) {
        observer.update(this);
    }
}

}
```

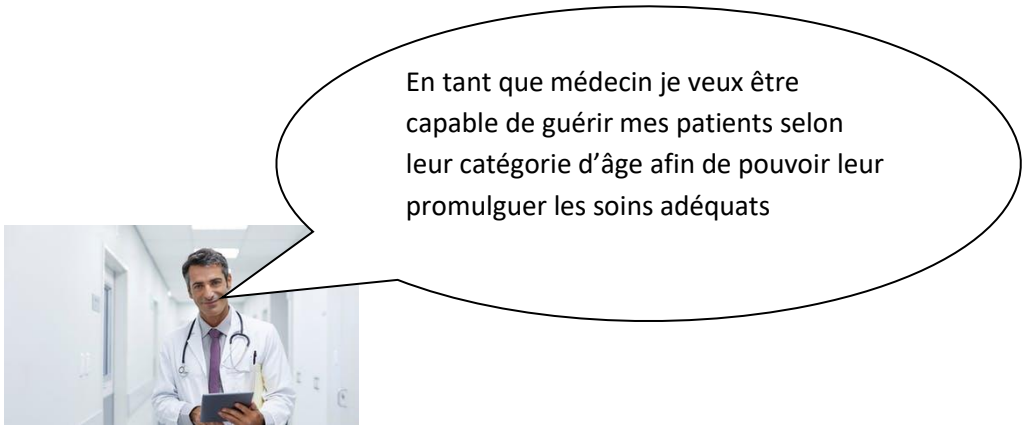
Ainsi, quand l'état de la classe Personne change elle doit envoyer un signal à tous ses observateurs (Médecin) qui doivent effectuer l'action nécessaire (enlever la personne de sa liste de patients grâce à *update*) en fonction du nouvel état de la classe.

```
public void update(Personne personne) {
    patients.remove(personne);

}
```

2). Design Pattern Stratégie:

Fier de votre projet vous avez décidé de le montrer à votre médecin traitant. Impressionné par votre travaille ce dernier vous félicite de votre travail et vous dit :« Je suis surpris qu'un casse-cou comme vous soit un aussi bon programmeur, c'est bluffant ! » (ne nous remerciez pour ça le mérite vous revient 😉)Puis il enchaine en disant : « votre projet est vraiment sympathique mais je pense que pour plus de réalisme vous devriez rajoutez le fait que on ne soigne pas tous les patients de la même façon, cela peut prendre plus de temps selon la gravité de votre problème, votre âge , vos antécédents médicaux… » . C'est vrai que cela semble une bonne idée mais là il y a beaucoup trop de paramètres essayons de le faire en utilisant seulement l'âge.

Ainsi, le deuxième besoin que l'on a pu discerner est consigné dans notre deuxième user story énoncée de cette façon :



Encore une fois cela tombe vraiment très bien car on peut facilement le faire et en plus pour ce genre de cas il existe un design pattern qui correspond parfaitement qui est Stratégie. Avant de vous expliquer comment utiliser ce design pattern on va préparer son implémentation. Créer une nouvelle branche depuis *develop* sur votre projet git. Dans la classe personne on va ajouter un attribut âge qui est un int et l'ajouter dans le constructeur afin qu'elle y soit initialisée (n'oubliez pas que dans le constructeur de médecin celui-ci appelle aussi le constructeur de personne il y a donc des modifications à faire dans la classe médecin).

Nous pouvons maintenant rentrer dans le vif du sujet avec qu'est-ce que ce design pattern ? Vous savez ce que l'on veut faire donc vous avez compris que stratégie nous permettra de faire la distinction entre plusieurs cas. C'est un problème qui est très simple je vous l'accorde et l'utilisation d'un pattern « juste pour ça » peut vous paraitre être un excès de zèle. En réalité ce pattern va vous permettre de faire plusieurs méthodes distinctes pour soigner une personne et quand votre projet sera plus complexe vous pourrez ainsi rajouter autant de méthodes pour soigner que vous le désirez sans pour autant modifier le reste de votre programme.

Pour ce faire vous devrez donc créer une interface GuérirStrategy ***:***

```
1  package blueJ;
2
3  public interface GuerirStrategy {
4          public int PointDeVieAAjouter();
5
6  }
```

Puis faire deux classes qui implémentent cette interface, dans notre cas GuerirJeune et GuerirVieux :

```
1   package blueJ;
2
3   public class GuerirJeune  implements GuerirStrategy {
4
5           public int PointDeVieAAjouter() {
6
7                   return 40 ;
8           }
9
10
11  }
```

```
1   package blueJ;
2
3   public class GuerirVieux implements GuerirStrategy {
4
5           public int PointDeVieAAjouter() {
6
7                   return 20 ;
8           }
9
10  }
```

Vous devez maintenant créer une méthode dans la classe personne qui va vous permettre de choisir la bonne stratégie :

```
39              public GuerirStrategy getStrategie () {
40                  GuerirStrategy s;
41
42                  if (this.age<50) {
43                          s = new GuerirJeune();
44                  }else {
45                          s = new GuerirVieux();
46                  }
47                  return s ;
48              }
```

Enfin il ne vous reste plus que à faire en sorte que votre Médecin applique la bonne méthode quand il vous guérit :

```
28      public int guerir(Personne patient, GuerirStrategy g ){
29          if(patient.getPointDeVie() + g.PointDeVieAAjouter()<= 100){
30              patient.setPointDeVie(patient.getPointDeVie() + g.PointDeVieAAjouter());
31          }else {patient.setPointDeVie(100) ;}
32
33          return patient.getPointDeVie();
34      }
```

Maintenant que vous avez tout fait vous pouvez faire commit sur votre git hub, puis fusionner votre branche avec develop.

IV).Tests Unitaires :

Vous n'avez pas l'impression d'avoir oublié quelque chose ? Normalement on vous a dit que tout développement devait être soumis avec ses testes unitaires validés ! Si vous l'avez fait sans que l'on ait à vous le dire alors BRAVO, sinon ce n'est pas grave mais cela doit devenir un automatisme chez vous car, « Tout ajout de code doit être testé avant et avoir sa barre verte ». Au cas où voici une liste non exhaustive de tests pour les ajouts de code.

Pour la classe voiture :

```
58      @Before
59      public void setUp() // throws java.lang.Exception
60      {
61          // Initialisez ici vos engagements
62          medecin1 = new Medecin("Messali", "Nassim",22);
63          ferrari = new Voiture(20);
64          lambo = new Voiture(30);
65          charly = new Personne(ferrari, "charly", "simonian",23);
66      }
```

```
123     @Test
124     public void testAccident()
125     {
126
127         ferrari.accident(30, medecin1);
128         assertEquals(70,ferrari.getConducteur().getPointDeVie() );
129         assertEquals(true,ferrari.getConducteur().getMalade() );
130         assertEquals(true, medecin1.getPatients().contains(charly) );
131         assertEquals(medecin1, charly.getMedecinTraitant());
132
133
134     }
```

Pour la classe Personne :

```
69      @Before
70      public void setUp() // throws java.lang.Exception
71      {
72
73          lambo = new Voiture(65);
74          clio = new Voiture(55);
75          idir = new Personne(clio, "idir", "houmel",20);
76
77
78          ferrari = new Voiture(20);
79          charly = new Personne(ferrari, "charly", "simonian",22);
80
81          patient2 = new Personne("c", "c",22);
82          patient1 = new Personne("Harb", "Cedric",22);
83          medecin1 = new Medecin("Messali", "Nassim",21);
84
85          idir.ajoutMedecinTraitant(medecin1);
86          charly.ajoutMedecinTraitant(medecin1);
87          patient1.ajoutMedecinTraitant(medecin1);
88          patient2.ajoutMedecinTraitant(medecin1);
89
90
91      }
```

```
198     @Test
199     public void testNotifyObservers(){
200
201         patient1.tomberMalade(20);
202         patient2.tomberMalade(10);
203         medecin1.guerir(patient1, patient1.getStrategie());
204         assertEquals(false, medecin1.getPatients().contains(patient1));
205         assertEquals(true, medecin1.getPatients().contains(patient2));
206
207
208     }
```

Pour la classe Medecin :

```
37      @Before
38      public void setUp()
39      {
40          patientUn = new Personne("Harb", "Cedric",22);
41          patientDeux = new Personne("Houmel", "Idir",80);
42          patientTrois = new Personne("Simonian", "Charly",22);
43          medecinUn = new Medecin("Messali", "Nassim" , 22);
44
45          patientUn.ajoutMedecinTraitant(medecinUn);
46          patientDeux.ajoutMedecinTraitant(medecinUn);
47          patientTrois.ajoutMedecinTraitant(medecinUn);
48
49
50
51      }
```

```
63      @Test
64      public void testGuerir()
65      {
66          assertEquals(90, patientUn.tomberMalade(10));
67          assertEquals(100, medecinUn.guerir(patientUn, patientUn.getStrategie()));
68      }
69
70      @Test
71      public void testGetPatients()
72      {
73          ArrayList<Personne> patients1 = new ArrayList<Personne>();
74          patients1.add(patientUn);
75          medecinUn.ajoutPatient(patientUn);
76          assertEquals(patients1, medecinUn.getPatients());
77      }
78
79      @Test
80      public void testAjoutPatient(){
81
82          medecinUn.ajoutPatient(patientTrois);
83          assertEquals(true, medecinUn.getPatients().contains(patientTrois));
84      }
```

V). Test automatisés :

Si vous aviez bien suivi le précédent tutoriel, vous allez alors tout de suite deviner quelle étape suit l'archivage de notre projet dans le processus d'intégration agile. Il s'agit bien-sûr de l'exécution et des tests automatisés. Il n'est plus nécessaire de vous présenter le framework utilisé dans ce cas-là à savoir ***Cucumber***. Selon la légende, ce dernier permettrait aux personnes ayant une fibre fonctionnelle de créer des tests automatisés sans écrire de code (à moins qu'ils maitrisent l'approche BDD (Behaviour Driven Development) bien entendu).

D'un point de vu plus formel en utilisant notre outil de modélisation de nos users story on obtient donc pour la première user story ceci :

```
@tag
Feature: Notifier Guerison
En tant que medecin
Je veux etre notifie du fait qu'une personne necessitant des soins est reellement guerie
Afin de pouvoir l'enlever de la liste des patients a traiter

  @tag1
  Scenario Outline: Guerison personne et notification medecin
    Given l <age> de la personne
    When elle tombe malade avec des <points_maladie> a enlever
    And  je la gueris
    Then je suis notifie de sa guerison et elle ne se trouve plus dans ma liste de patient


    Examples:
      | age|  points_maladie       |
      | 50 |        20                                |
      | 20 |        30             |

```

Avec un fichier steps ressemblant à ceci :

```java
package userStory;

import static org.junit.Assert.assertEquals;

import blueJ.*;
import cucumber.api.java.en.*;

public class notifierGuerisonSteps {

        private Personne patient1;
        private Medecin medecin1;

        @Given("l {int} de la personne")
        public void l_de_la_personne(Integer int1) {
            // Write code here that turns the phrase above into concrete actions
            patient1 = new Personne ("toto", "titi", int1);
            medecin1 = new Medecin("Messali", "Nassim" , 22);

            patient1.ajoutMedecinTraitant(medecin1);
        }

        @When("elle tombe malade avec des {int} a enlever")
        public void elle_tombe_malade_avec_des_a_enlever(Integer int1) {
            // Write code here that turns the phrase above into concrete actions
            patient1.tomberMalade(int1);
        }

        @When("je la gueris")
        public void je_la_gueris() {
            // Write code here that turns the phrase above into concrete actions
            medecin1.guerir(patient1, patient1.getStrategie());
        }

        @Then("je suis notifie de sa guerison et elle ne se trouve plus dans ma liste de patient")
        public void je_suis_notifie_de_sa_guerison_et_elle_ne_se_trouve_plus_dans_ma_liste_de_patient() {
            // Write code here that turns the phrase above into concrete actions
                assertEquals(false, medecin1.getPatients().contains(patient1));
        }



}
```

Et pour la deuxième User Story on obtiendrait plutôt ceci :

```
@tag
Feature: Strategie guerison
En tant que medecin
Je veux etre capable de guerir mes patients selon leur categorie d'age
Afin de pouvoir leur promulguer les soins adequats
  @tag1
  Scenario Outline: Guerir patient
    Given l <age1> de la personne inferieur a cinquante ans
    And   l <age2> de la personne superieur ou egal a cinquante ans
    When elles tombent malade avec des <points_maladie> a enlever
    Then je les gueris en ajoutant quarante aux <points_vies1> de la personne jeune et vingt aux <points_vies2> de la personne vieille

    Examples:
      | age1 | age2 | points_maladie  | points_vies1 | points_vies2 |
      | 30   |  70  | 70              |    70        | 50           |
      | 15   |  50  | 40              |    100       | 80           |
      | 25   |  60  | 20              |    100       | 100          |
```

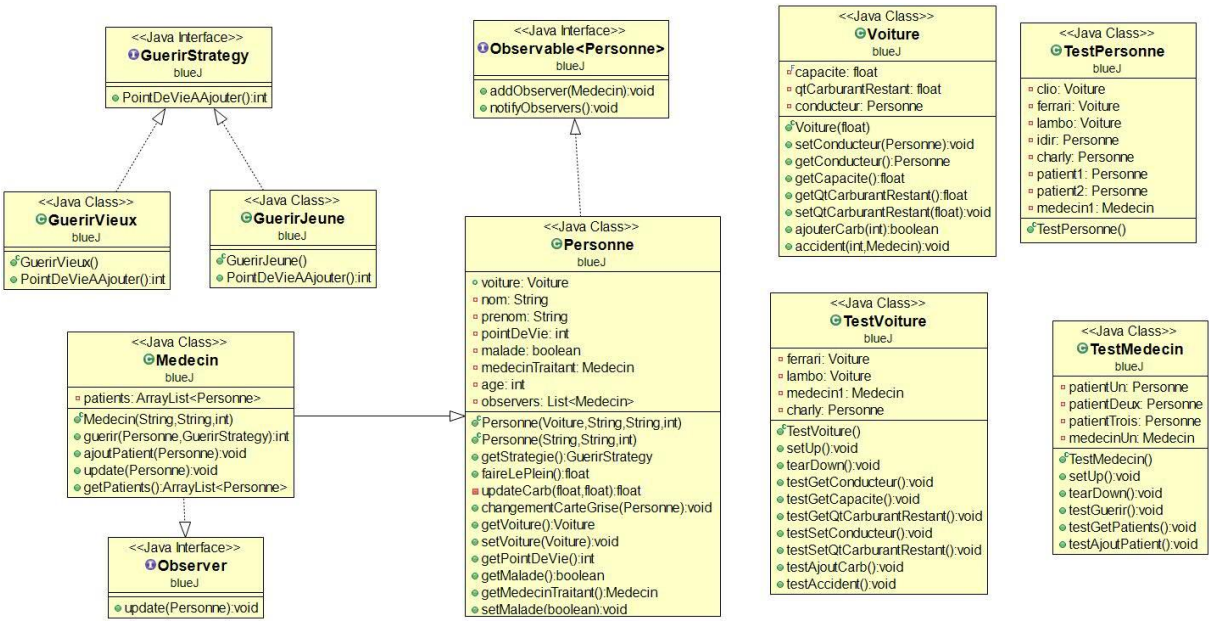
Avec un fichier steps semblable l'image ci-dessous :

```java
package userStory;

import static org.junit.Assert.assertEquals;

import blueJ.Medecin;
import blueJ.Personne;
import cucumber.api.java.en.*;

public class strategieGuerisonSteps {

	private Personne patient1;
	private Personne patient2;
	private Medecin medecin1;


	@Given("l {int} de la personne inferieur a cinquante ans")
	public void l_de_la_personne_inferieur_a_cinquante_ans(Integer int1) {
	    // Write code here that turns the phrase above into concrete actions
		patient1 = new Personne ("toto", "titi", int1);
	    medecin1 = new Medecin("Messali", "Nassim" , 22);

	    patient1.ajoutMedecinTraitant(medecin1);
	}

	@Given("l {int} de la personne superieur ou egal a cinquante ans")
	public void l_de_la_personne_superieur_ou_egal_a_cinquante_ans(Integer int1) {
	    // Write code here that turns the phrase above into concrete actions
		patient2 = new Personne ("momo", "mimi", int1);
	    medecin1 = new Medecin("Messali", "Nassim" , 22);

	    patient2.ajoutMedecinTraitant(medecin1);
	}

	@When("elles tombent malade avec des {int} a enlever")
	public void elles_tombent_malade_avec_des_a_enlever(Integer int1) {
	    // Write code here that turns the phrase above into concrete actions
	    patient1.tomberMalade(int1);
	    patient2.tomberMalade(int1);
	}

	@Then("je les gueris en ajoutant quarante aux {int} de la personne jeune et vingt aux {int} de la personne vieille")
	public void je_les_gueris_en_ajoutant_quarante_aux_de_la_personne_jeune_et_vingt_aux_de_la_personne_vieille(Integer int1, Integer int2) {
	    // Write code here that turns the phrase above into concrete actions
	    medecin1.guerir(patient1, patient1.getStrategie());
	    medecin1.guerir(patient2, patient2.getStrategie());

	    assertEquals(patient1.getPointDeVie(), int1,0.1);
	    assertEquals(patient2.getPointDeVie(), int2,0.1);
	}
```

V) Fin et perspective

Nous arrivons maintenant à la fin de notre périple qui fut semé d'embûches et de péripéties en tout genre ! C'est le moins que l'on puisse dire !

Si vous avez bien suivi toutes les étapes de notre tuto votre projet devrait avoir la structure suivante :



Vous pouvez maintenant finir notre petit tutoriel en continuant à vous amuser pour poursuivre et améliorer votre projet. Pour notre part, nous avons tout fait pour que votre projet parte avec les meilleures bases possible alors bonne continuation et peut être à bientôt pour un prochain tuto !!

On vous laisse méditer sur cette citation de Steve Jobs (1955-2011) :

« La seule manière de faire du bon travail, c'est d'aimer ce que vous faites »