Agilité

La rencontre du Loto et de la Course et boom… ça fait gagner de l’argent.  
Version: 1.1

11 – 02 – 2019

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |

Sommaire

[Sommaire 2](#_Toc874155)

[Historique du document 3](#_Toc874156)

[1. Objectif & Contexte 4](#_Toc874157)

[1.1 Objectif 4](#_Toc874158)

[1.2 Contexte 4](#_Toc874159)

[2. Implémentation des Designs Patterns 5](#_Toc874160)

[2.1 Design Pattern Observer 5](#_Toc874161)

[2.2 Design Pattern AbstractFactory 7](#_Toc874162)

[3. Mini tutoriel 8](#_Toc874163)

[4. Tests JUnit 10](#_Toc874164)

[5. Cucumber 13](#_Toc874165)

[6. Annexe 15](#_Toc874166)

[6.1 Création d’un projet Eclipse 15](#_Toc874167)

[6.2 Tests JUnit 19](#_Toc874168)

[6.3 Cucumber via Eclipse 20](#_Toc874169)

[6.3.1 Installation de Cucumber 20](#_Toc874170)

[6.3.2 Configuration de Cucumber pour exécution des tests 21](#_Toc874171)

[6.3.3 Création des fichiers pour description des Users Story 22](#_Toc874172)

Historique du document

|  |
| --- |
| Informations générales |

|  |  |
| --- | --- |
| Rédacteur(s) : | HADDAD Abderraouf  AISSAT Yacine  TIOUCHICHINE Lounes  ANDRIAMBELOMAHERY Sam Michael |
| Valideur(s) : | ZAMFIROIU MICHEL |
|  |  |

|  |  |  |
| --- | --- | --- |
| Historique des mises à jour | | |
| Version | Date | **Description** |
| 1.1 | 11/01/2019 | Création du document |
| Documents de référence | | |
| https://github.com/yacineaissat/Agilite\_Course\_Loto\_AAHT.git | | |

1. Objectif & Contexte
   1. Objectif

Ce document a pour objectif de décrire les étapes de réalisation du projet d’Agilité. Ce document a été conçu à la suite de la fusion des deux projets qui étaient pourtant très prometteurs l’un et l’autre… Le jeu de Loto et la Course. Depuis leur fusion, le projet a pris une toute autre ampleur, les 4 contributeurs ont alliés leurs forces pour produire un projet de qualité. Accrochez-vous pour la suite…

* 1. Contexte

A une époque lointaine, où la Terre n’était composée que d’un seul continent, le monde était vaste et des lieux improbables existaient, citons par exemple Onizuka, Uranus, Numidia et l’incontournable Madina, des places mythiques où le lendemain de chaque pleine lune se déroulaient des courses mythiques. Durant ces courses, de nombreux paris étaient organisés et les joueurs tentaient de gagner le plus d’or possible en pariant sur le classement des courses.

« Uhtred le jeune », fils d’Uhtred de Madina, était un de ces parieurs. Paresseux, il ne s’intéressait que peu aux coureurs, à leurs capacités et à leurs résultats. Il n’avait que faire des favoris, des coureurs blessés ou d’autres caractéristiques. Sa manière de jeu était assez singulière. Il classait simplement les joueurs de manière aléatoire grâce à leur numéro de dossard, et à la fin de la course il comparait son classement avec le résultat pour savoir si oui ou non il avait gagné de l’or.

L’histoire d’Uhtred le jeune a inspiré, des siècles plus tard, de nombreux jeux, notamment le jeu du Loto.

Afin de retracer l’histoire de ce valeureux parieur, nous avons décidé d’unir les forces de la classe fétiche Loto et de la classe fétiche Course afin de recréer la méthode de pari d’Uhtred.

L’objectif est simple. 6 joueurs avec un numéro de dossard participent à une course. Le parieur va alors entrer ces 6 numéros de manière aléatoire afin d’avoir le résultat de sa course. Il aura la possibilité notamment de connaître le gain potentiel qu’il peut faire avec ce classement aléatoire, mais il pourra surtout savoir si oui ou non il a gagné à l’issue de la course.

1. Implémentation des Designs Patterns

C’est connu, les Designs Patterns n’ont pas été inventés pour faire beau. Ils ont un objectif bien défini. Répondre à des problèmes qui se posent couramment par une conception. Ici, on va donc présenter les deux principaux problèmes qui se sont présentés à nous et les deux designs patterns que nous avons choisis pour tenter d’y répondre.

* 1. Design Pattern Observer

Le premier Design Pattern que nous avons choisi d’implémenter est l’Observateur.

Le problème qui se posait à nous est que les valeurs de certains de nos attributs changent régulièrement. De plus, un certain nombre d’autres classes du projet doivent être tenues au courant de l’évolution de ces valeurs. C’est un problème récurrent qui se pose dans beaucoup de projets informatiques.

Forts de ce constat, nous avons cherché un moyen pour pallier ce problème de mise à jour des valeurs.

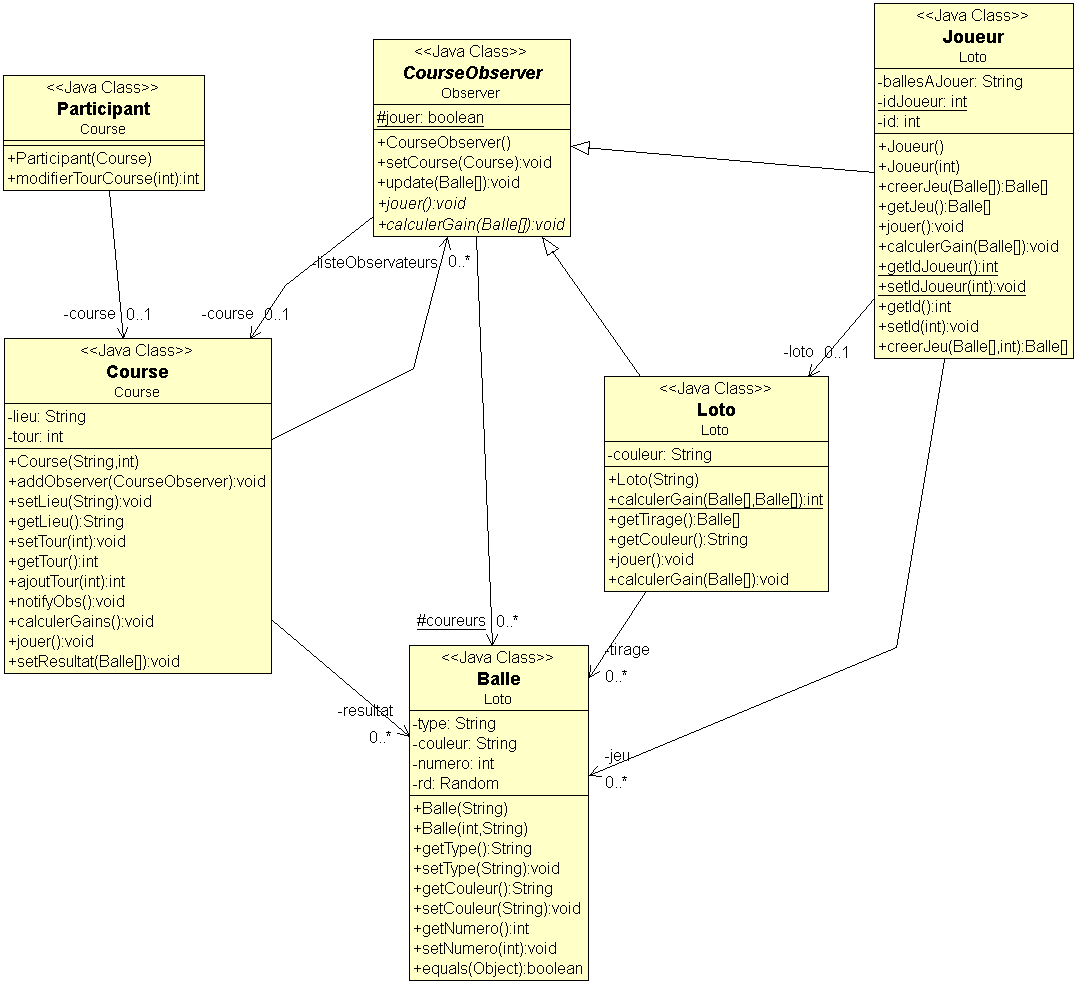
Première idée ? Les classes interrogent la classe qui est mise à jour régulièrement afin de lui demander si elle a mis à jour des valeurs récemment. Elle récupère alors les nouvelles valeurs et les met à jour localement.

On voit très vite que cette idée est… comment dire… un peu pourrie. En effet, quand les classes doivent demander si une valeur est mise à jour ? Toutes les 10 secondes ? Toutes les minutes ? Tous les 30 ans ? Bonne question… Et peu importe, pour tous les appels faits ‘inutilement’, dans le sens où on va demander si une modification à eu lieu et que la réponse est non, on aura une surcharge inutile d’appels qui peut être assez importante si on a beaucoup de classes et que les demandes sont faites régulièrement. Mauvaise idée donc.

Deuxième idée ? Le plus intelligent dans ce cas serait de mettre en place une conception qui permet que, dès lors que la classe se met à jour, elle va notifier les autres classes pour leur dire « Hé ho ! J’ai mis à jour plusieurs valeurs les amis, mettez-les à jour chez vous aussi sinon on va être en décalage. Bises ! »

C’est donc dans cette optique là que nous avons mis en place le premier design pattern, l’Observateur, qui répond parfaitement à ce problème d’implémentation.

Voici le diagramme de classe de l’implémentation du design pattern Observer.



* 1. Design Pattern AbstractFactory

Le second Design Pattern que nous avons choisi d’implémenter est l’AbstractFactory.

Le second problème qui se posait à nous est que, étant donné que nous faisons des paris sur des courses, très vite, étant donné que le business s’agrandit, nous allons devoir gérer différents types de courses. Et alors, me direz-vous ? Eh bien, on ne gère pas pareillement une course de voiture, une course de coureurs ou une course d’escargots… Nous avons donc besoin d’une conception simple qui nous permettent d’ajouter facilement de nouveaux types de courses à gérer. C’est la même chose pour les joueurs/parieurs, nous avons plusieurs types de parieurs. Les habitués, les professionnels, ceux qui jouent en ligne… Nous devons pouvoir les différencier et les implémenter facilement.

Forts de ce constat, nous avons cherché un moyen pour pallier ce problème de généricité et d’adaptabilité de notre code.

Première idée ? Ecrire une succession de conditions qui suivant le type de courses/ de personnes, instancie et retourne l’objet correspondant.

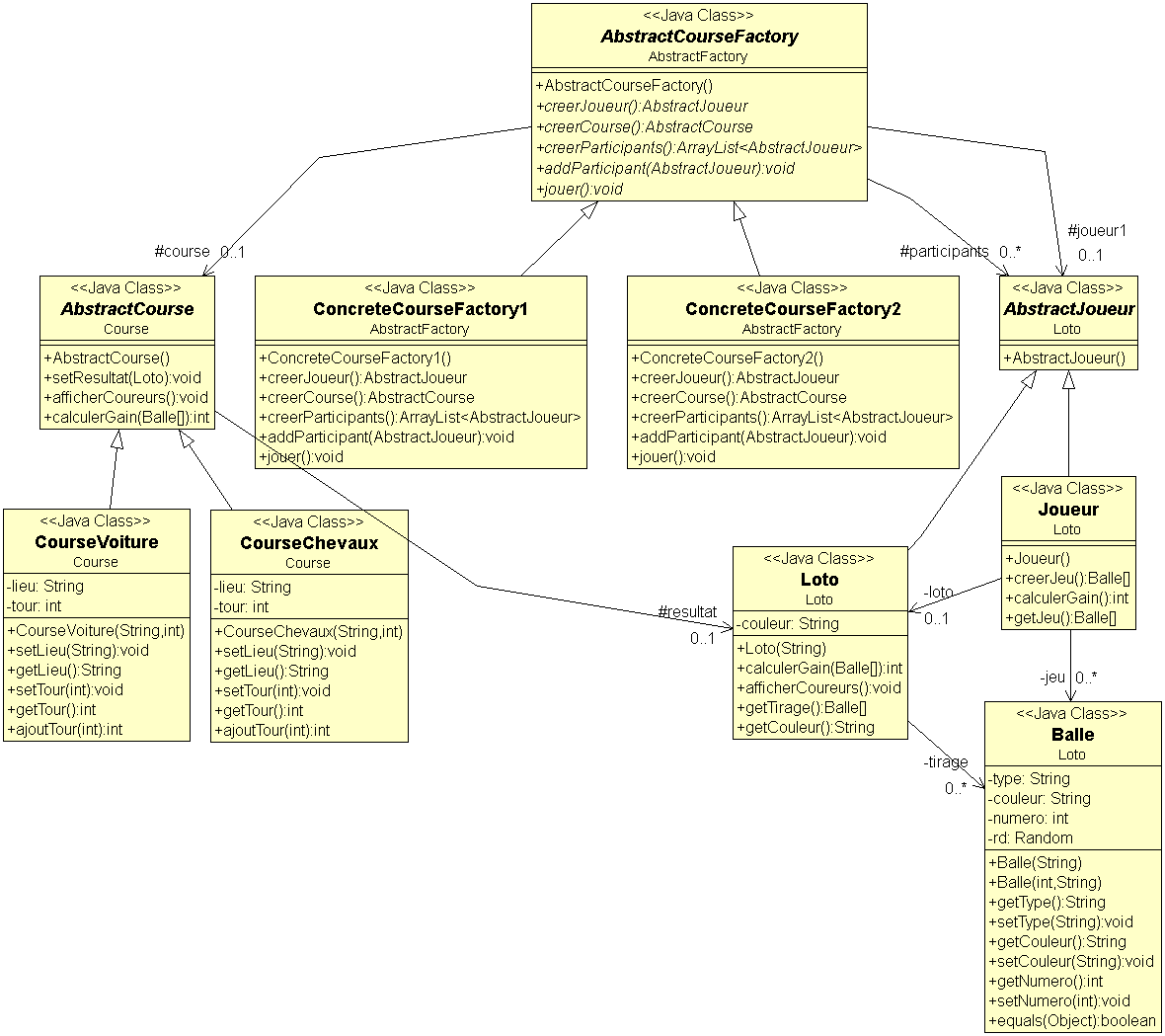
Le problème ici, c’est que la classe mère va être forcément couplée à tous les produits qu’elle va instancier. Et le problème se posera lorsqu’on souhaitera ajouter de nouveaux types de joueurs ou de nouvelles courses… et également lorsqu’on souhaitera supprimer par exemple un type de courses qui ne rapporte pas d’argent. Mauvaise idée encore…

Deuxième idée ? La meilleure solution serait d’avoir un créateur qui va pouvoir manipuler les sous-produits. Chaque créateur pourra donc créer des produits dont il est responsable. Tous les produits implémentent bien-sûr la même interface afin que les classes créatrices puissent s’y référer dans forcément connaitre le type concret.

Dans notre cas, étant donné qu’une partie de l’implémentation est identique à tous les produits, on peut alors choisir de faire de l’interface générale une classe abstraite afin d’intégrer le code partagé dans celle-ci.

C’est donc dans cette optique là que nous avons mis en place le second design pattern, l’AbstractFactory, qui répond parfaitement à ce problème d’implémentation.

Voici le diagramme de classe de l’implémentation du design pattern AbstractFactory.



1. Mini tutoriel

Dans cette partie, nous allons tenter d’expliquer de la manière la plus simple possible à l’utilisateur comment s’approprier notre code et surtout comment le faire fonctionner.

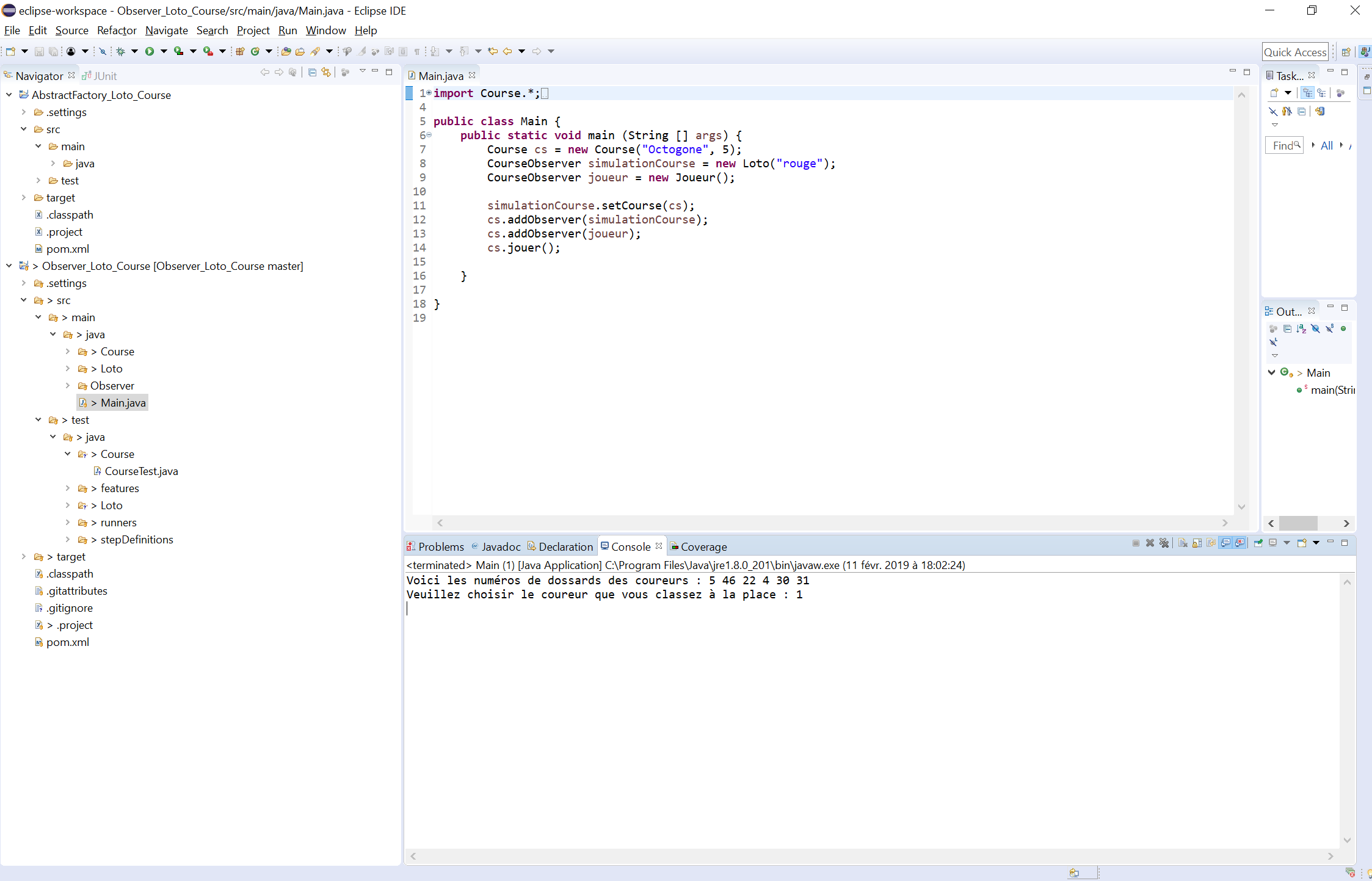
L’unique prérequis est d’avoir un IDE fonctionnel (Eclipse de préférence) sur son poste.

Pour commencer, l’utilisateur doit :

1. Récupérer le code via le lien Github suivant :

https://github.com/yacineaissat/Agilite\_Course\_Loto\_AAHT.git

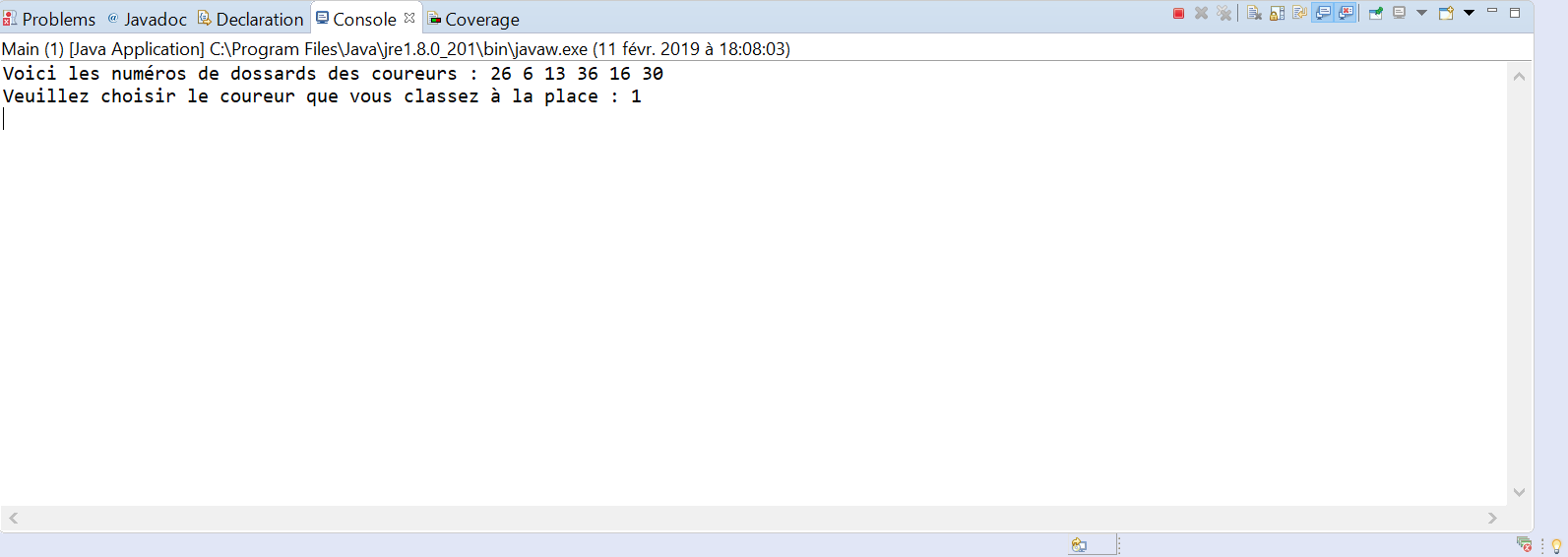
1. Importer le projet sur son environnement (Eclipse de préférence)



1. Une fois que les projets sont finalement ouverts, on va tester les deux Designs Patterns de manière distincte.

Tout d’abord, le pattern Observer.

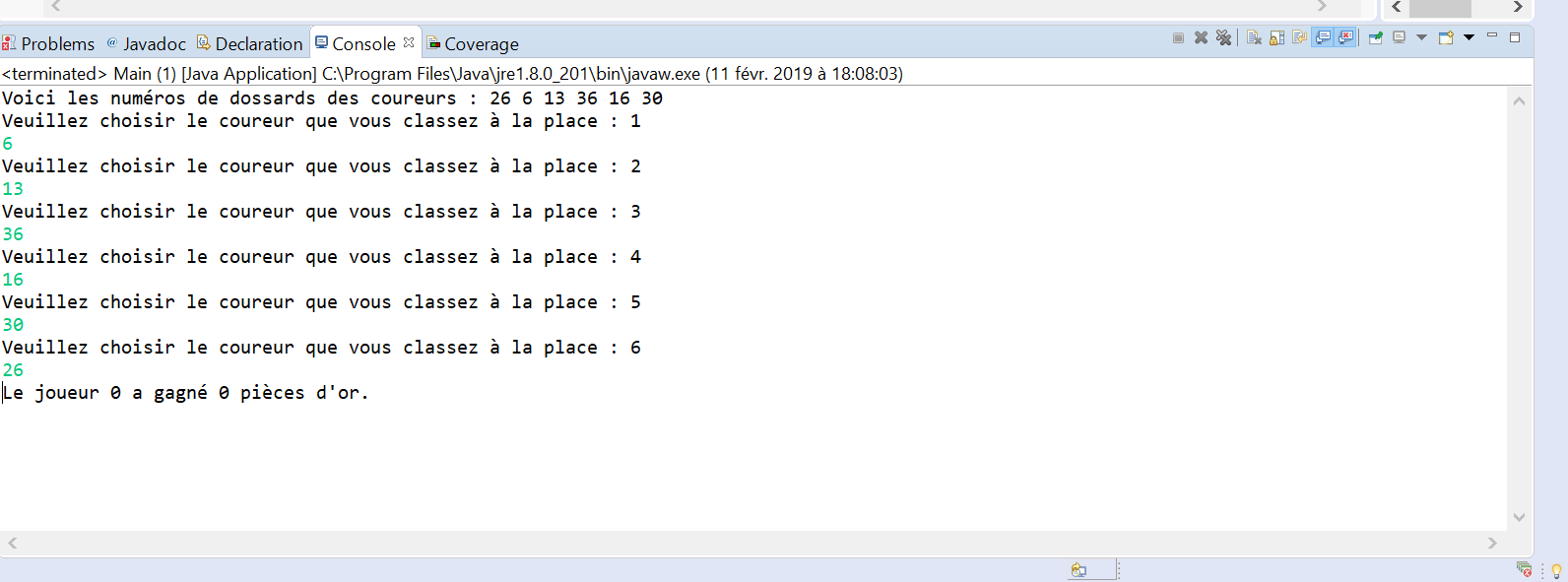
On lance la classe Main.java



On va alors tenter de deviner l’ordre d’arrivée des coureurs.

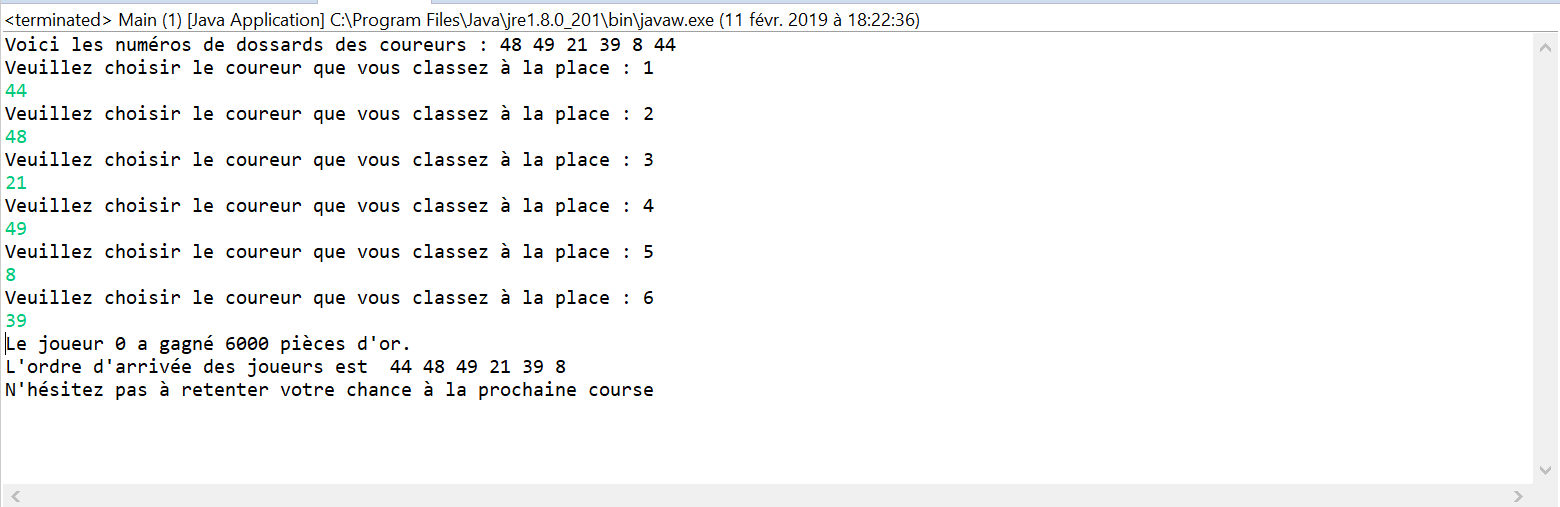
Pour cela, on va entrer un à un l’ordre d’arrivée des joueurs.

Et là… notre gain s’affiche…



Dans notre cas, on a tout perdu !! Nous finissons pauvres et endettés… On a pensé à braquer une banque pour essayer de se refaire mais finalement on va peut être retenter notre chance une deuxième fois sur la course suivante… peut-être qu’on arrivera à renflouer les caisses…

Réessayons donc une deuxième fois…



YESSSS ! On a trouvé 2 bonnes positions ! On termine riches et on se met à la retraite pour le restant de nos jours 😊😊😊

Trêve de plaisanteries…

A vous de tenter votre chance maintenant !

On a ici pu tester le code du pattern Observer, maintenant, testons le code du pattern AbstractFactory.

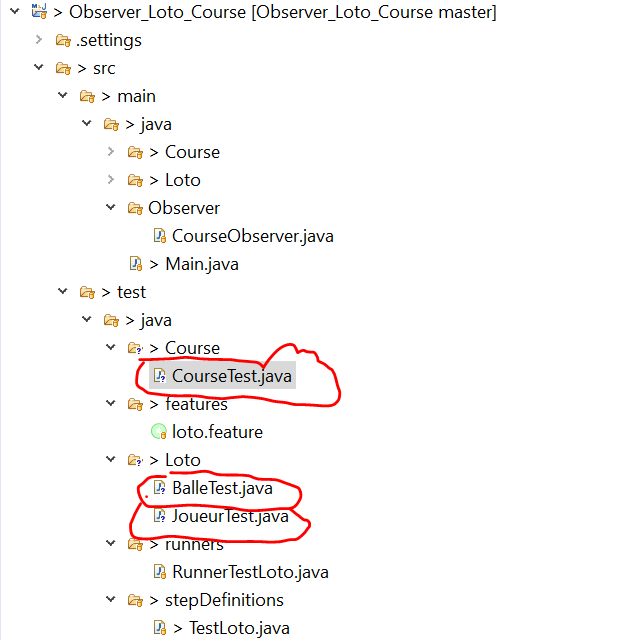
PATTERN ABSTRACTFACTORY

Maintenant, passons à la partie la plus intéressante, les tests ! On vous laisse découvrir dans la suite les tests JUnit et les tests Cucumber… restez bien accroché ça va être très intense.

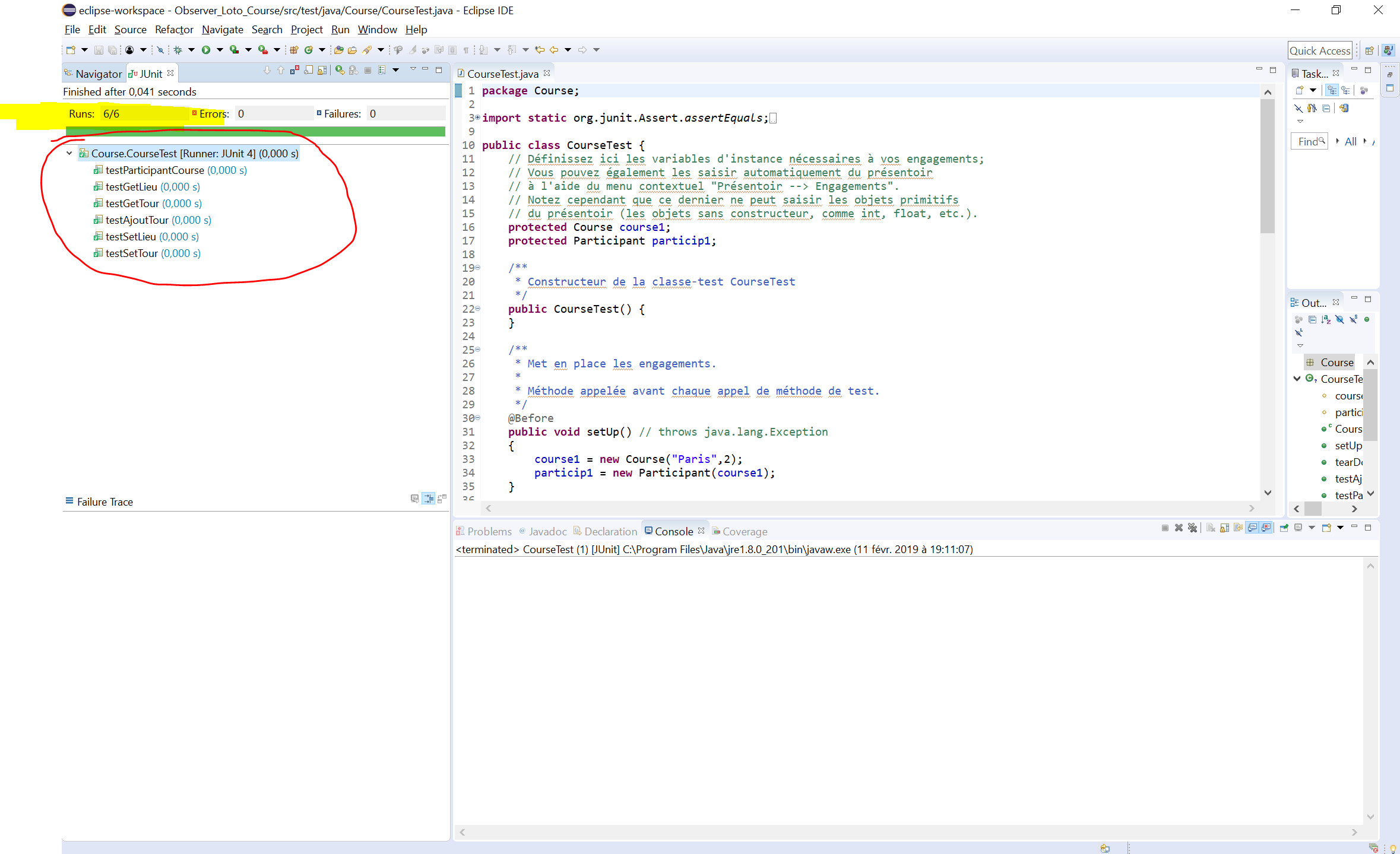
1. Tests JUnit

Afin d’exécuter les tests JUnit, il y a 3 classes à exécuter.

Vous les trouverez ci-dessous entourées en rouge.

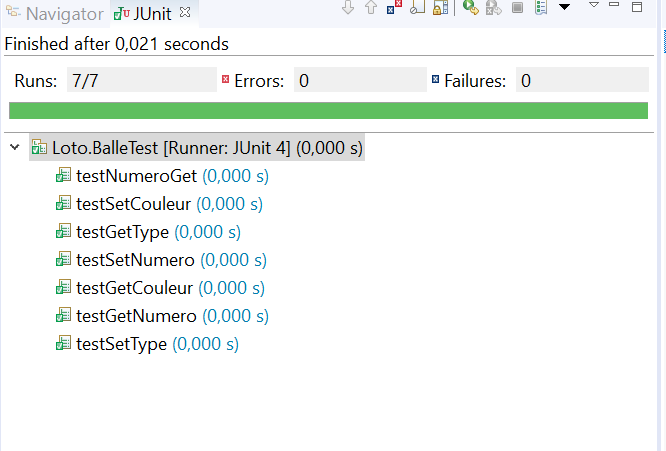


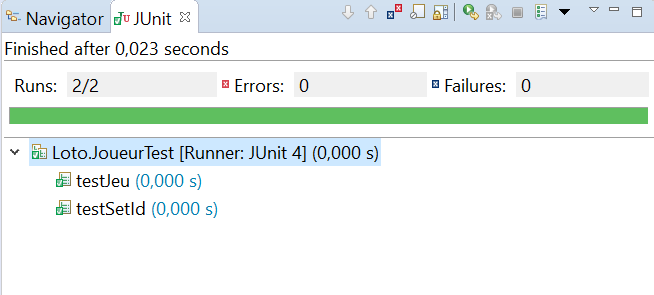
Exemple d’exécution de la classe : CourseTest.java



On voit à gauche que l’exécution s’est extrêmement bien passée, tout va bien dans le meilleur des mondes, on a la barre verte pour les 6 tests.

Exécution des deux autres classes de tests.

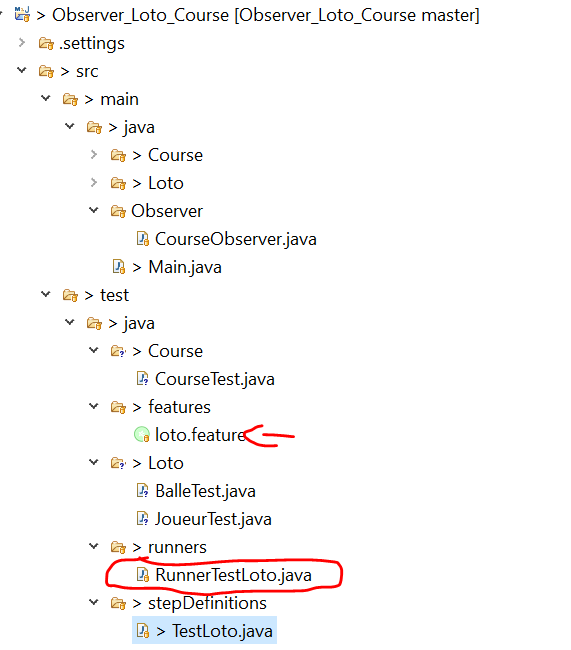




Les exécutions se déroulent bien pour chacun des classes… à vous maintenant de faire le test. Les tests sont identiques pour les deux patterns, cela n’a donc aucun intérêt de les exécuter pour les deux.

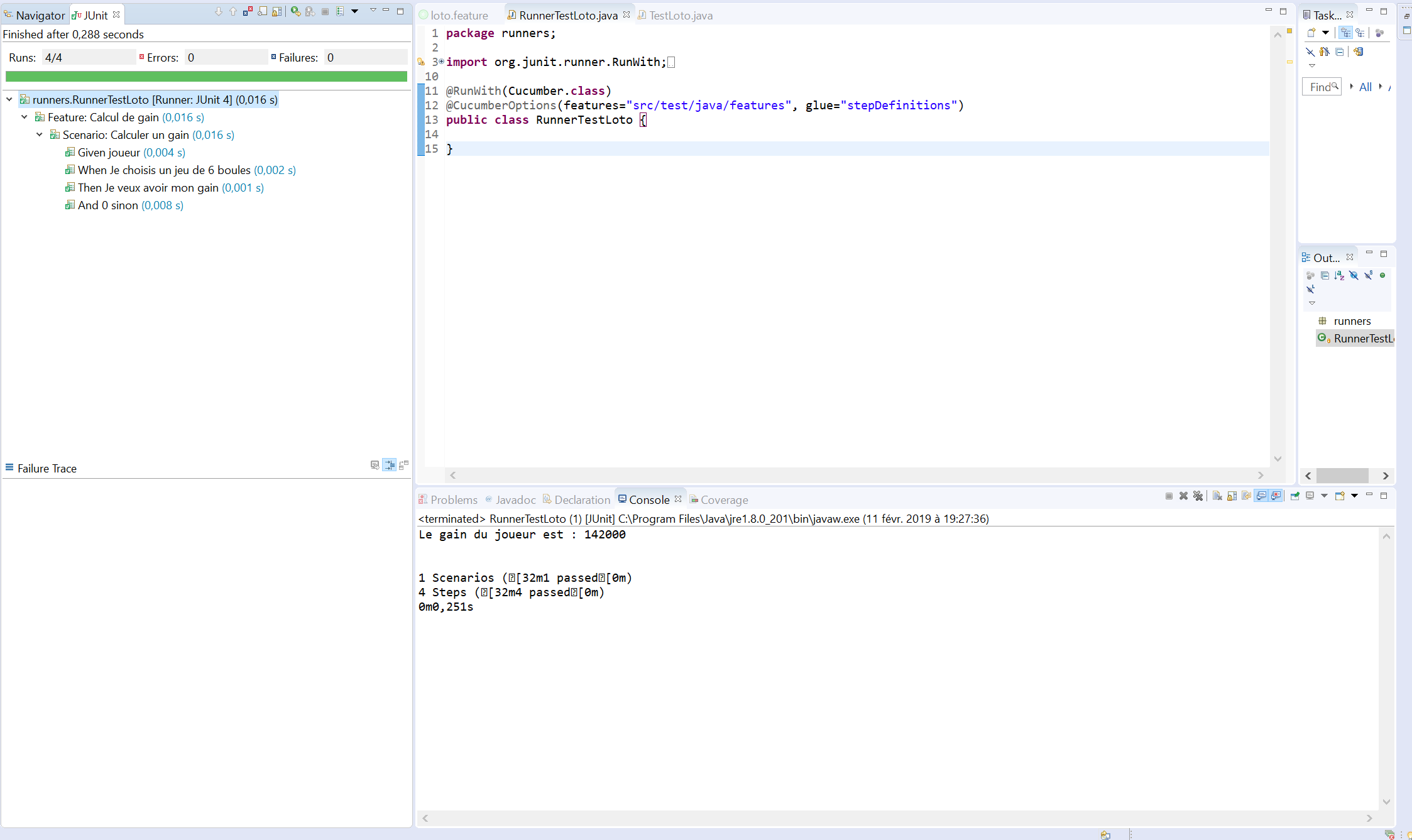
1. Cucumber

Afin d’exécuter les scenarii Cucumber, il va falloir exécuter la classe RunnerTestLoto.java comme (celle entourée sur la capture suivante) :



Petit détail : le fichier loto.feature (celui suivi d’une flèche rouge) contient le scénario à exécuter.

Voici le résultat de l’exécution :



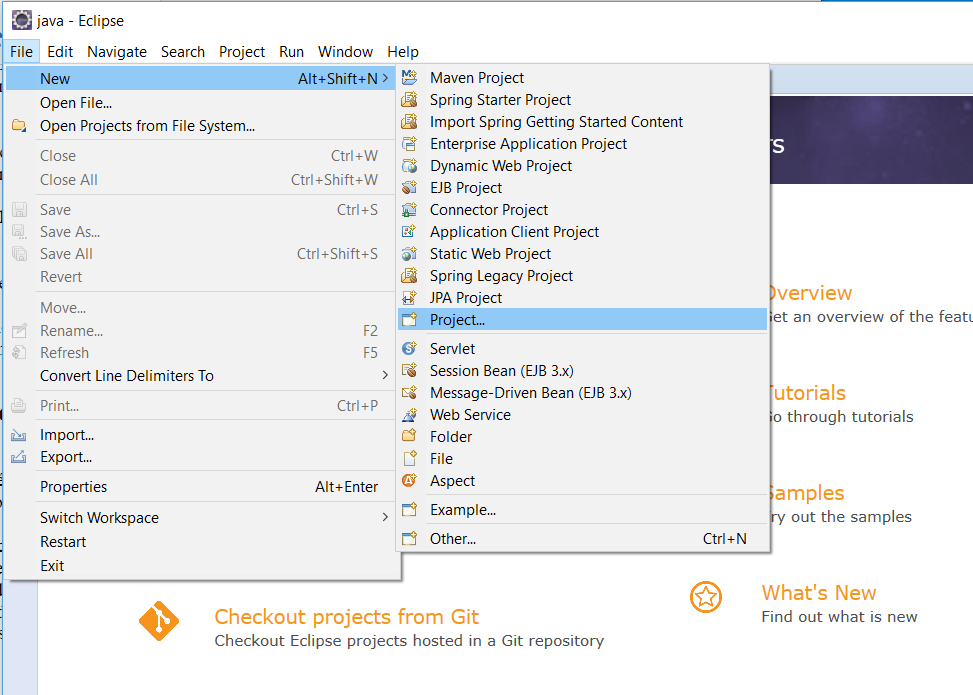
TADAM ! Encore des barres vertes qui signifient que les tests se sont bien exécutés…

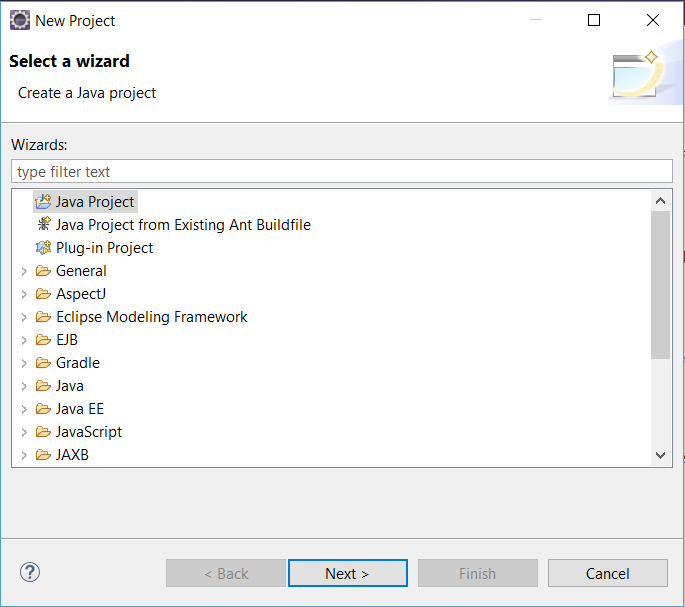
Encore une fois, nous vous laissons exécuter par vous-même les tests si l’envie vous prend… 😊 Attention, allez-y doucement, c’est super addictif !

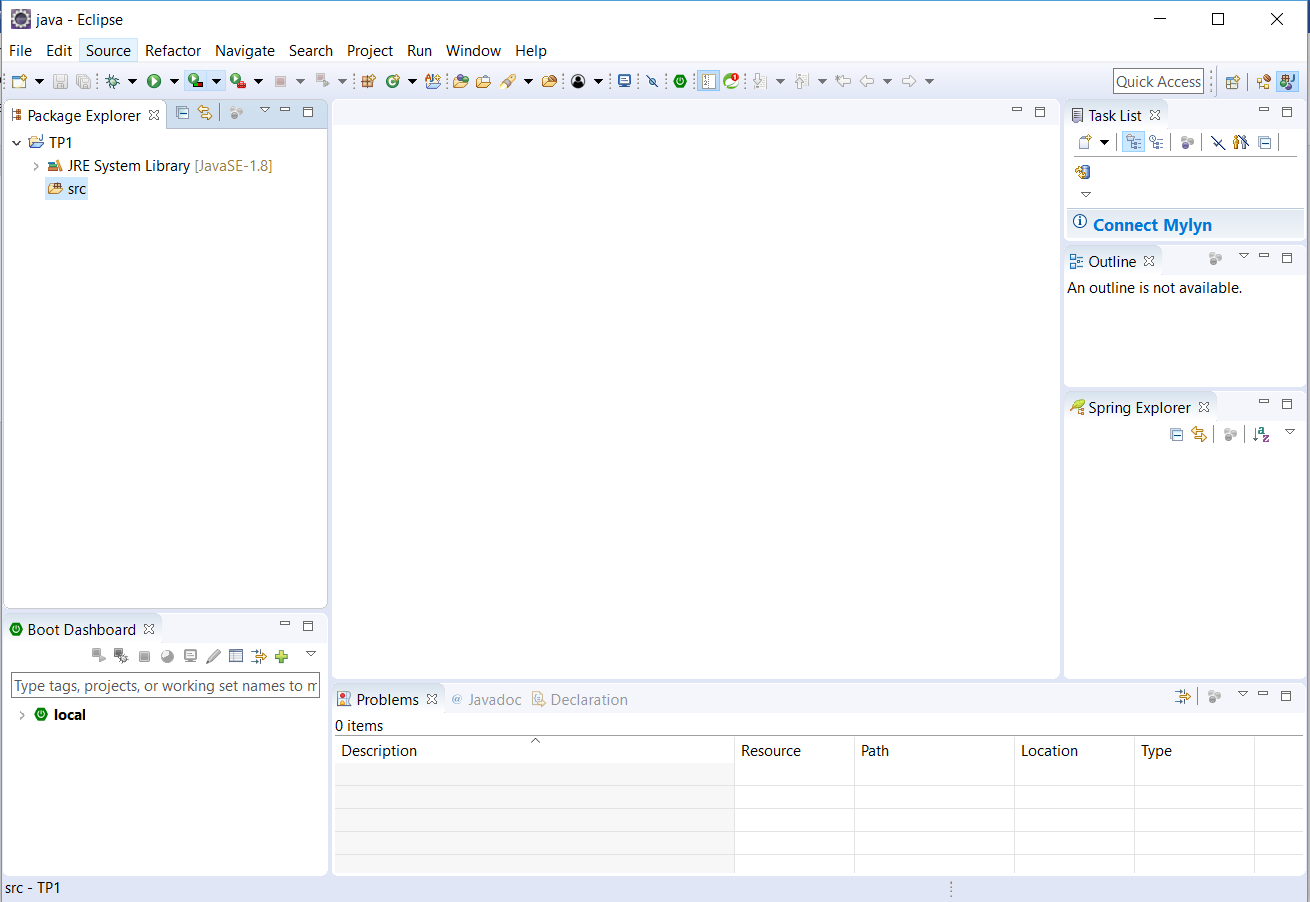
1. Annexe

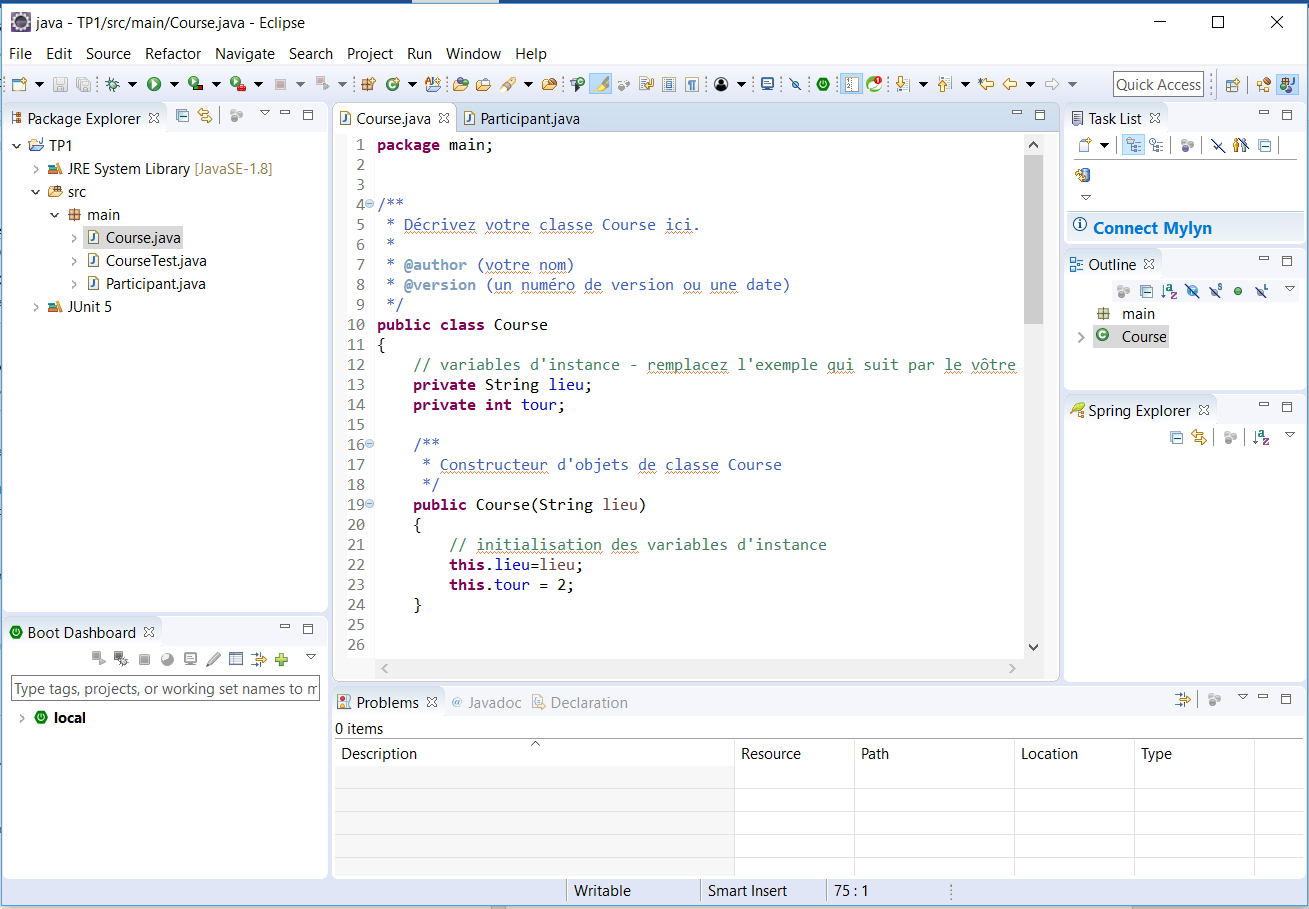
En annexe, vous trouverez quelques trucs et astuces qui nous ont beaucoup servi tout au long de la réalisation de notre projet. Nous trouverons notamment des archives des précédents rendus de TP d’Agilité et quelques tutoriels pour commencer à utiliser certains outils. Enjoy 😊

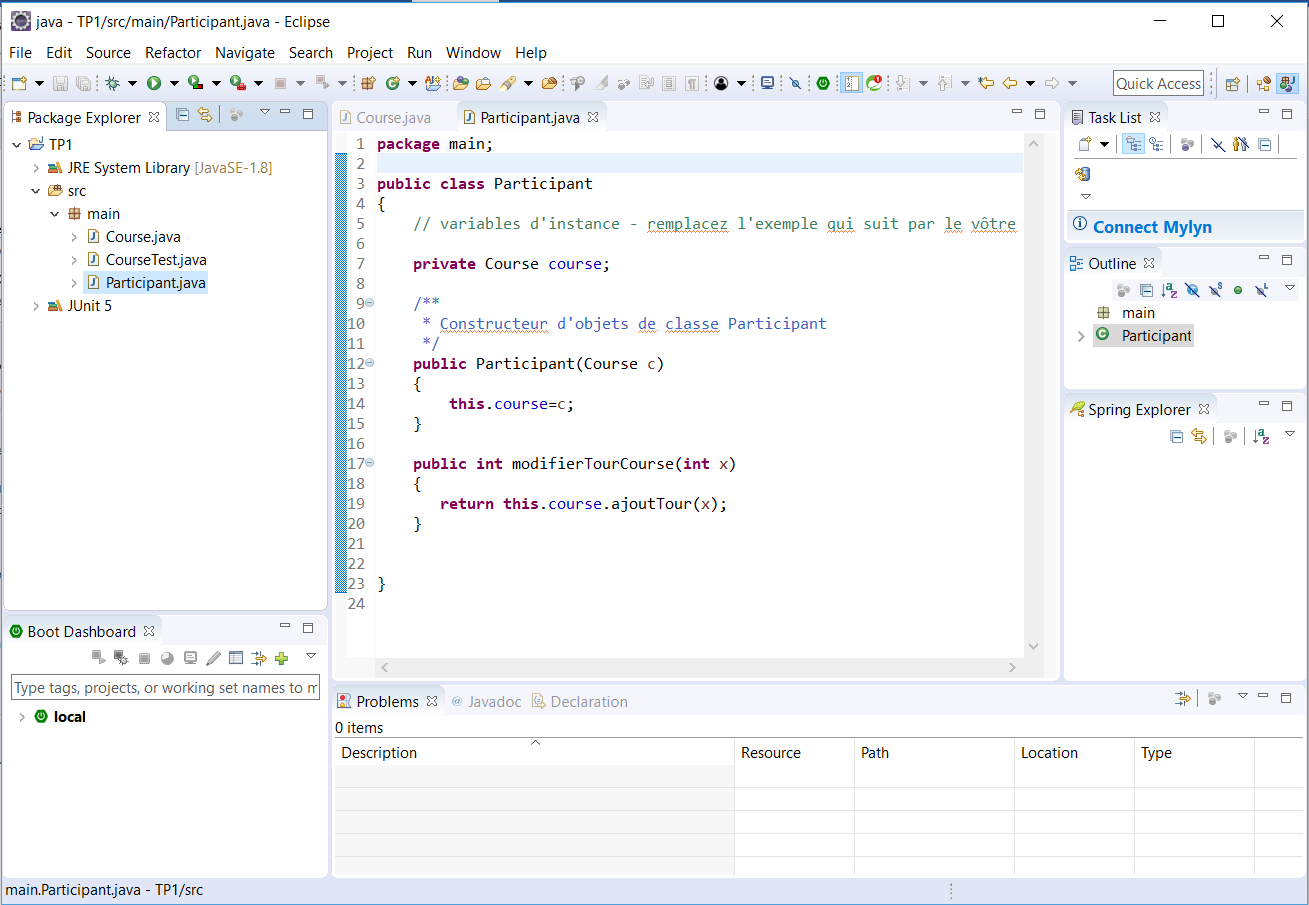
* 1. Création d’un projet Eclipse

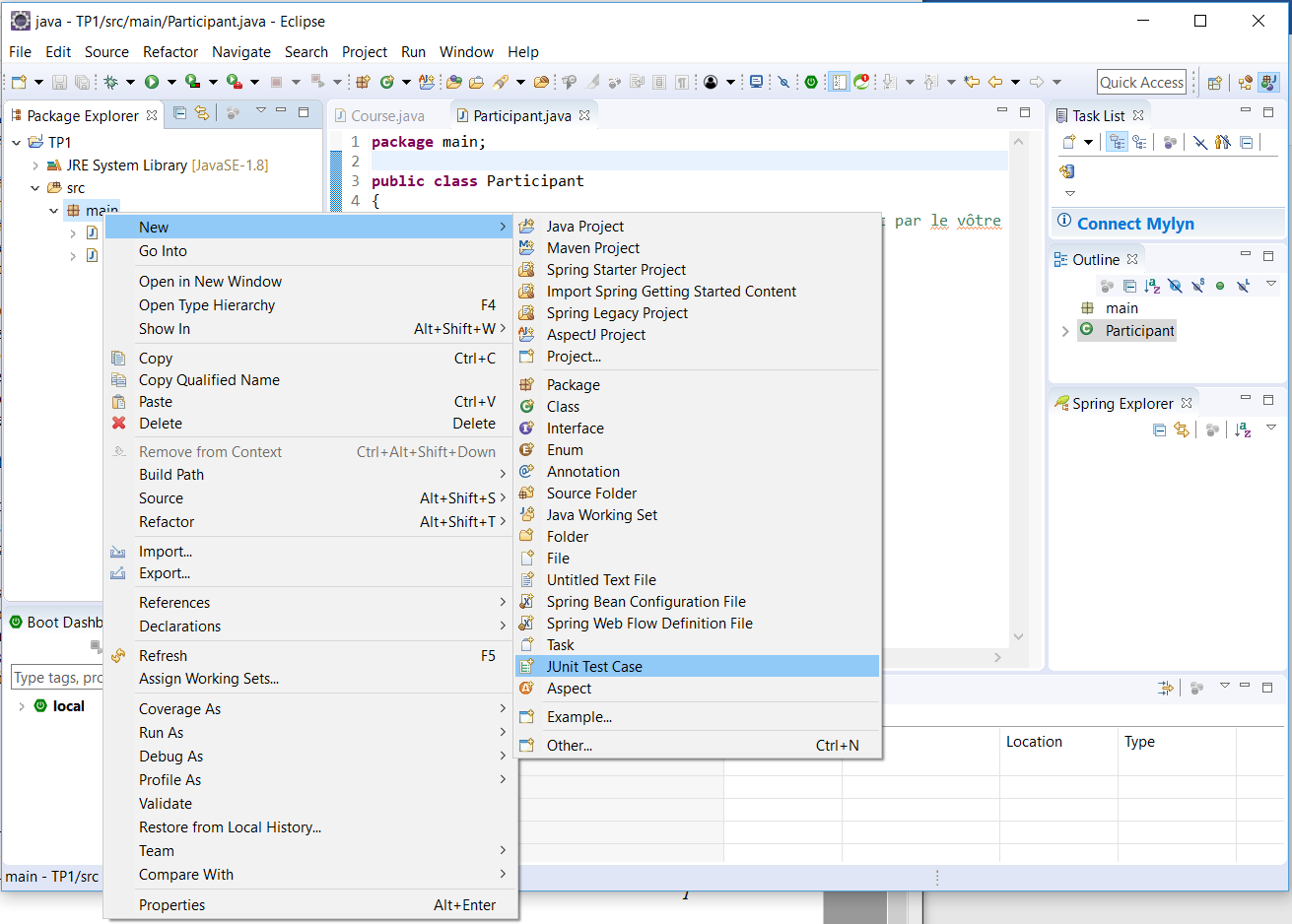


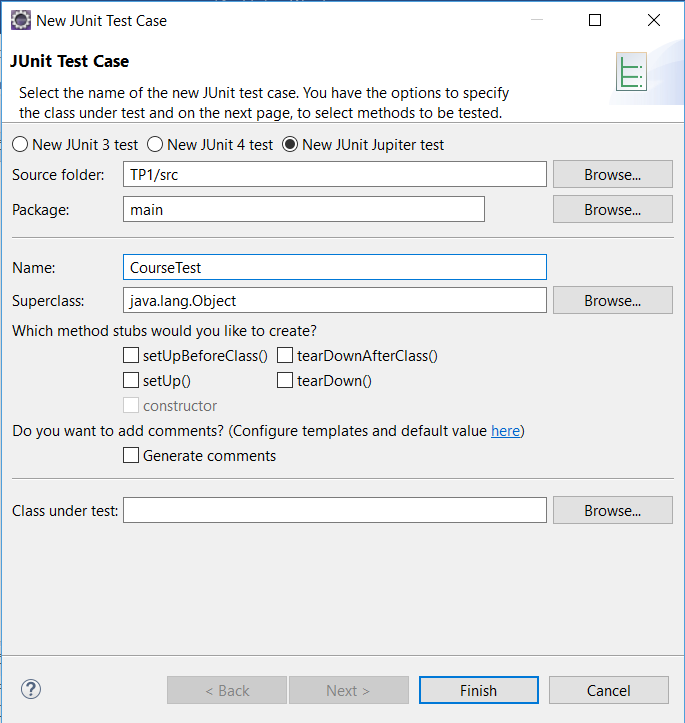




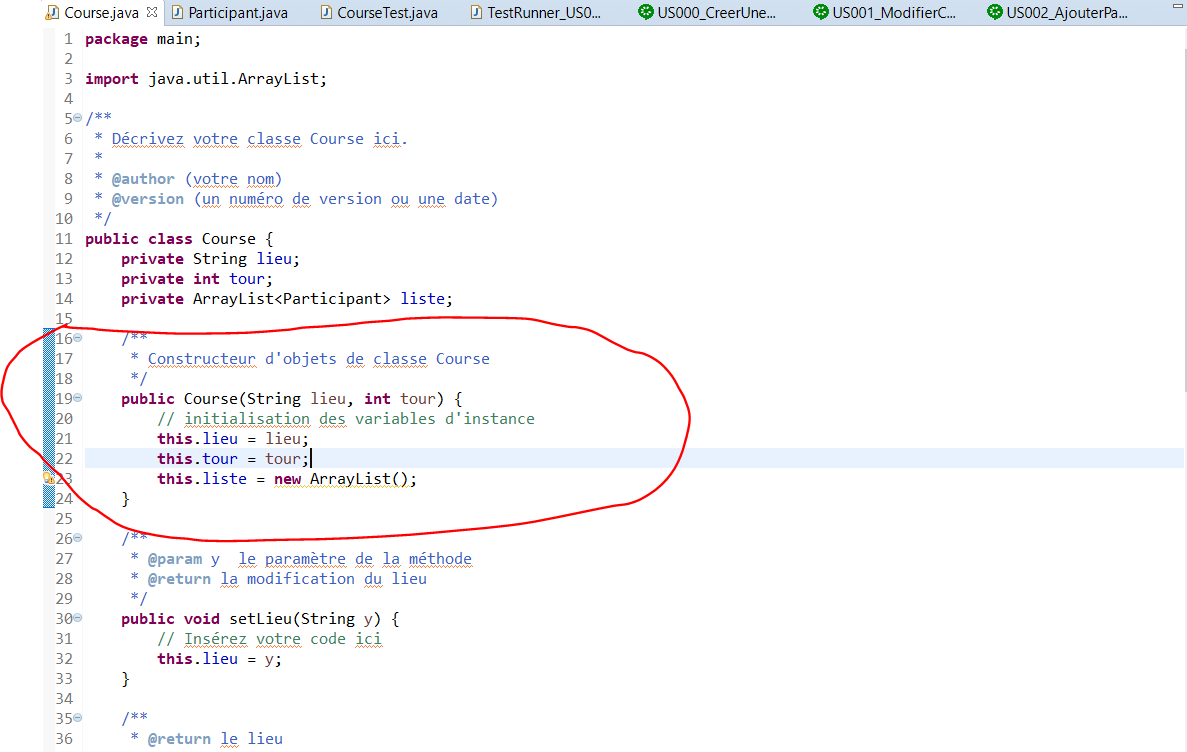


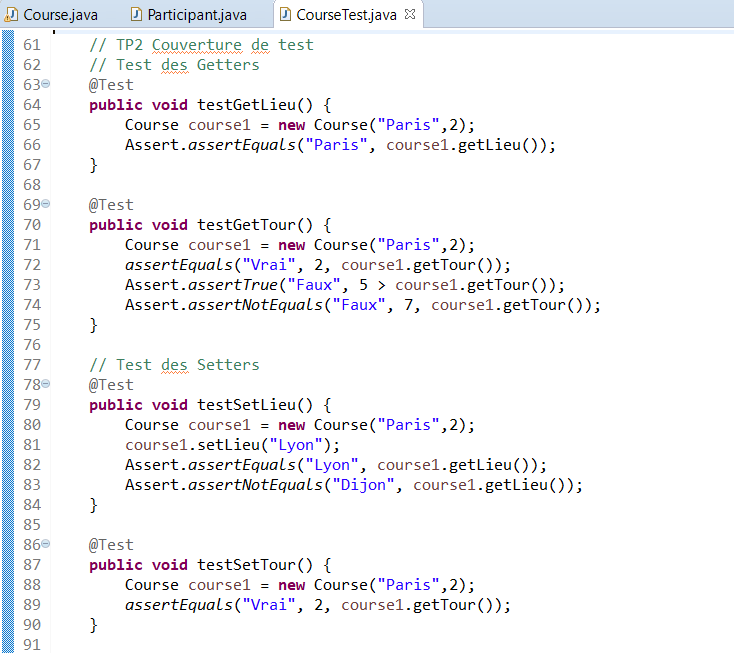


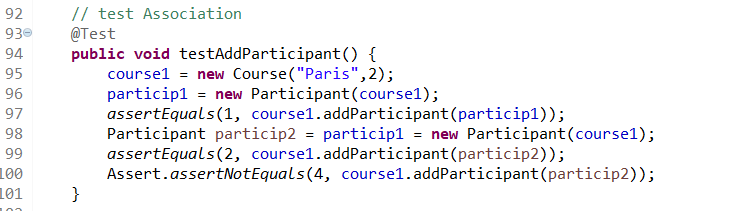


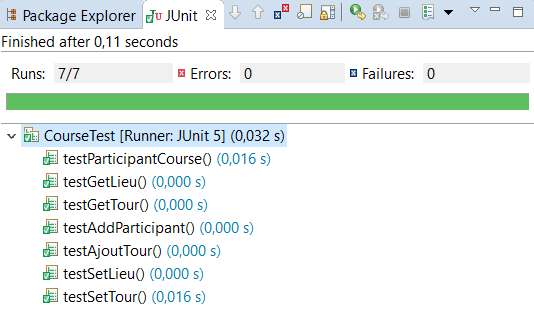


* 1. Tests JUnit



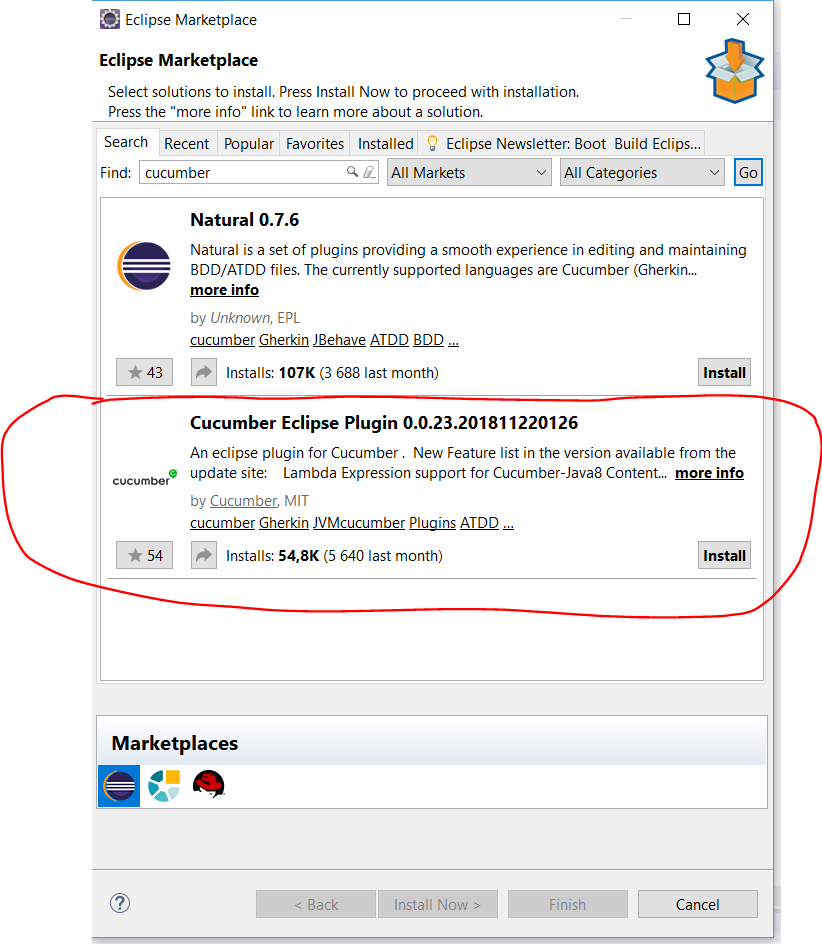




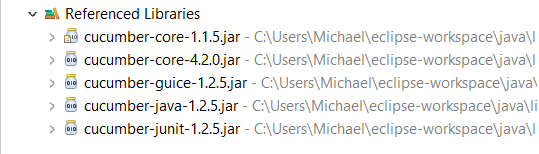


* 1. Cucumber via Eclipse
     1. Installation de Cucumber

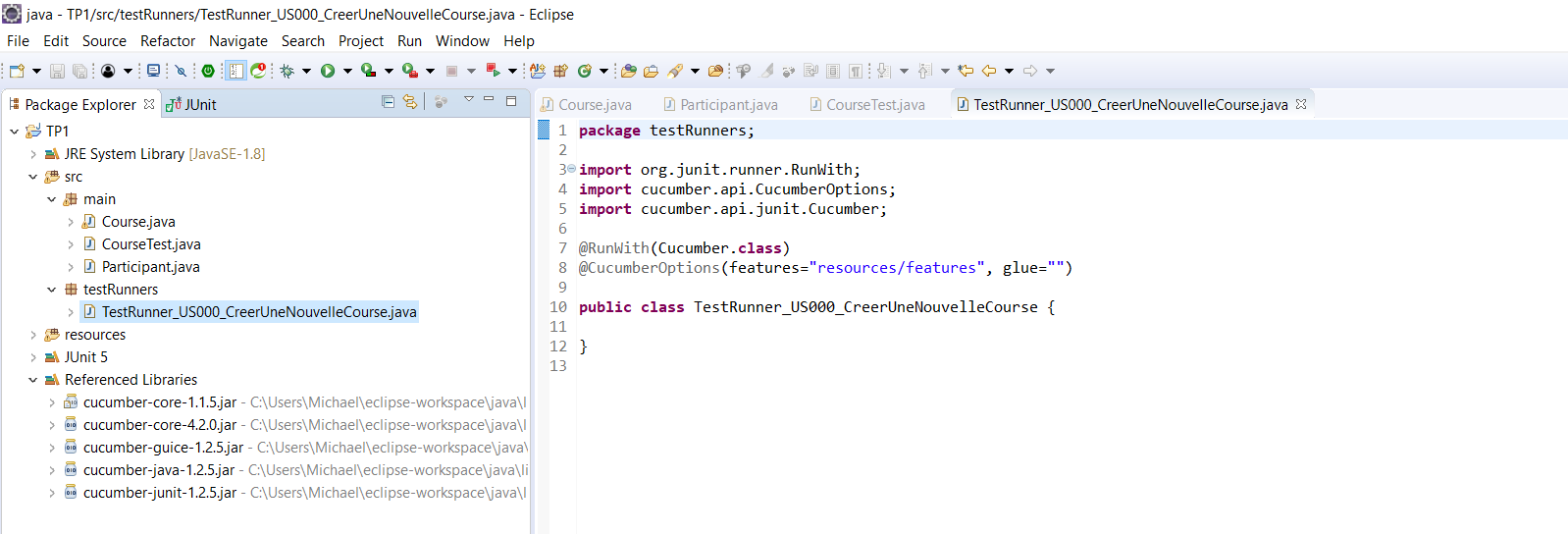
Installation de Cucumber



Ajout via le Build Path Eclipse :



* + 1. Configuration de Cucumber pour exécution des tests



* + 1. Création des fichiers pour description des Users Story

