Dauphine | PSL
UNIVERSITÉ PARIS

Projet Méthode Agiles d'ingénierie logicielle

Tutoriel pour l'apprentissage par l'expérimentation des concepts orientés objet en Java

Gestion de Livres

Auteurs :

Laye Kemo DIARSO
Michel NGUYEN

Fonctionnalités ou éléments traités	Auteurs	Version	Date de modification	Commentaires
Initiations à BleuJ	Laye Kemo DIARSO Michel NGUYEN	1.0	28/04/2020	
Eclipse et jUnit	Laye Kemo DIARSO Michel NGUYEN	1.1	29/04/2020	
Cucumber – BDD	Laye Kemo DIARSO Michel NGUYEN	1.2	24/05/2020	

Table des matières

Description du projet :

Ce projet consiste à la mise en place d'un tutoriel pour l'apprentissage des concepts orientés objets en java.

Pour cela, nous nous baserons sur l'exemple d'un projet permettant de gérer les livres en mode agile.

I. Premiers Pas avec BleuJ

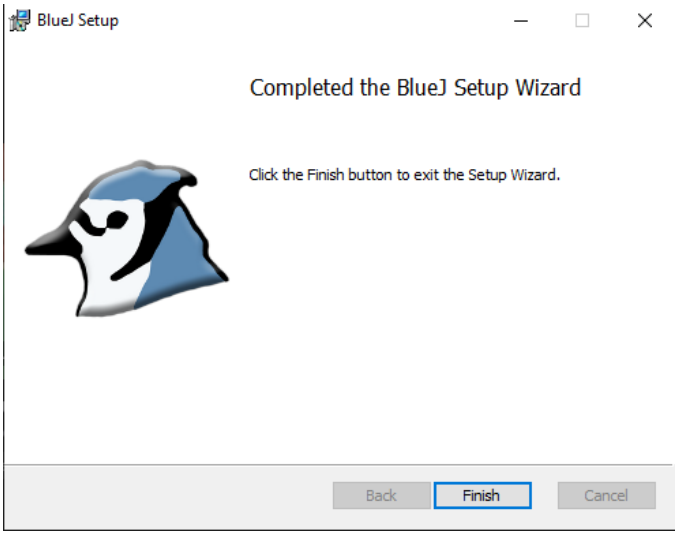
Afin de se familiariser avec l'environnement BleuJ, nous allons commencer par nous initier à travers l'exemple des livres.

1. *Téléchargez BlueJ*

Pour télécharger BlueJ il faut se rendre sur le site officiel de BlueJ http://www.bluej.org/ en choisissant la version de votre système d'exploitation.

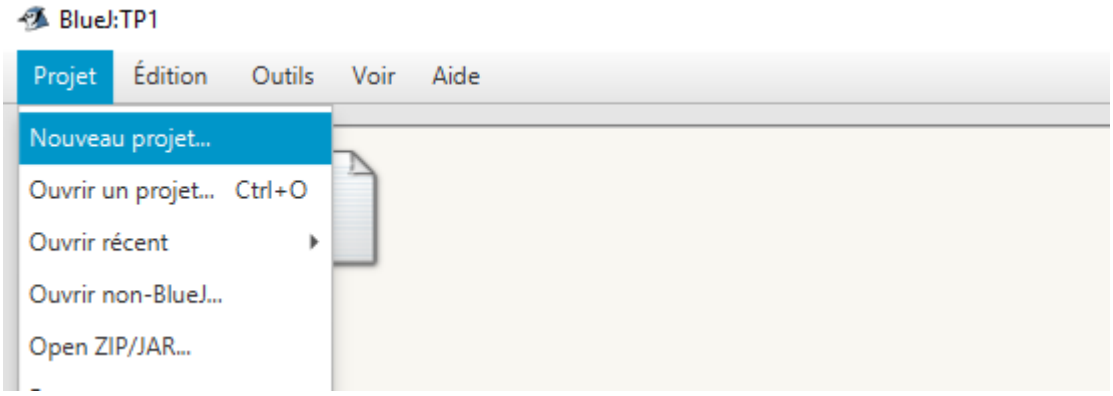
2. *Installez-le sur votre machine*

Double-cliquez sur le setup BlueJ précédemment téléchargé, cliquez sur next en choisissant les différentes options relatives à votre besoin. Après l'installation, cliquer sur finish.

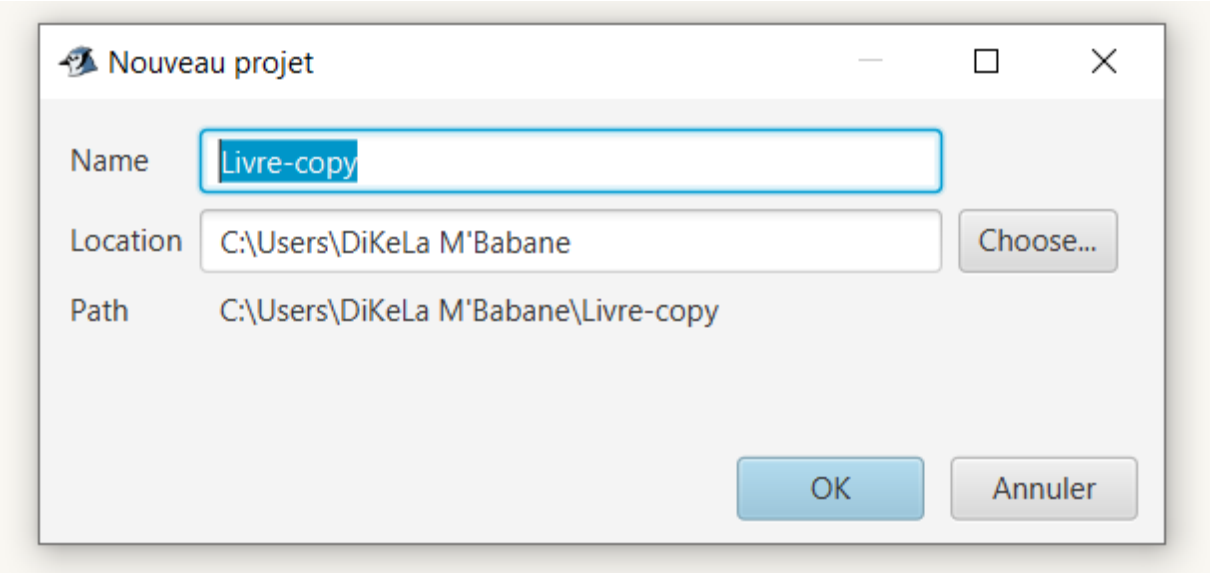


3. *Créez un nouveau projet*

Lancez l'application BlueJ après l'installation. Sur l'interface de l'environnement BlueJ, cliquez sur Projet puis Nouveau Projet.

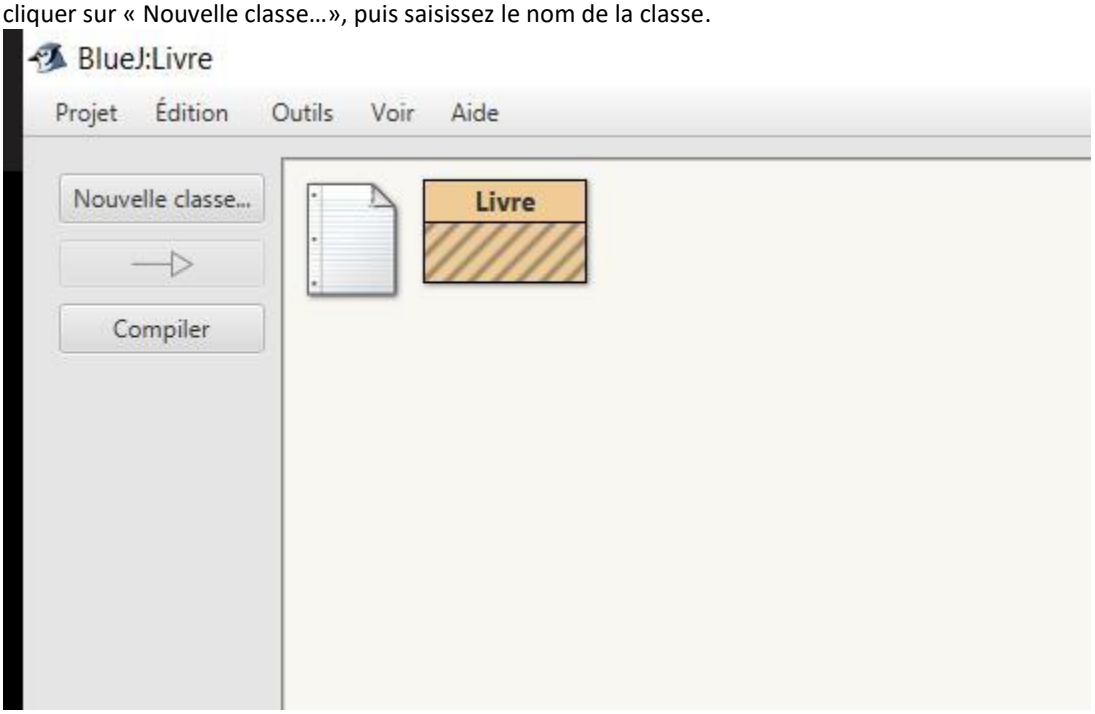


Ajoutez le Nom de votre projet, et cliquez sur Ok.



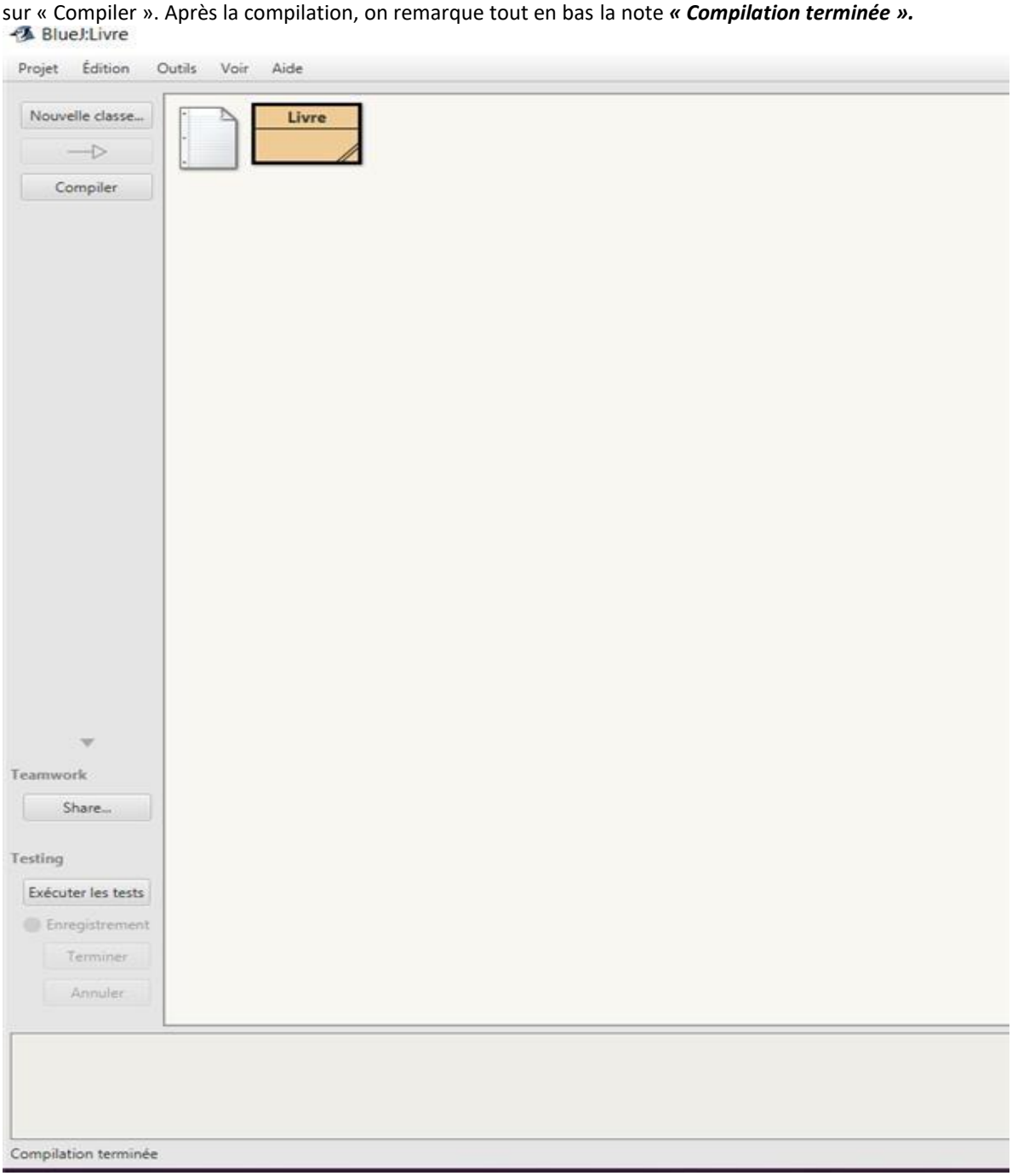
4. *Création de classe :*

Nous allons maintenant créer une classe. Dans notre cas, notre classe s'appellera « Livre ». Pour cela, cliquer sur « Nouvelle classe… », puis saisissez le nom de la classe.



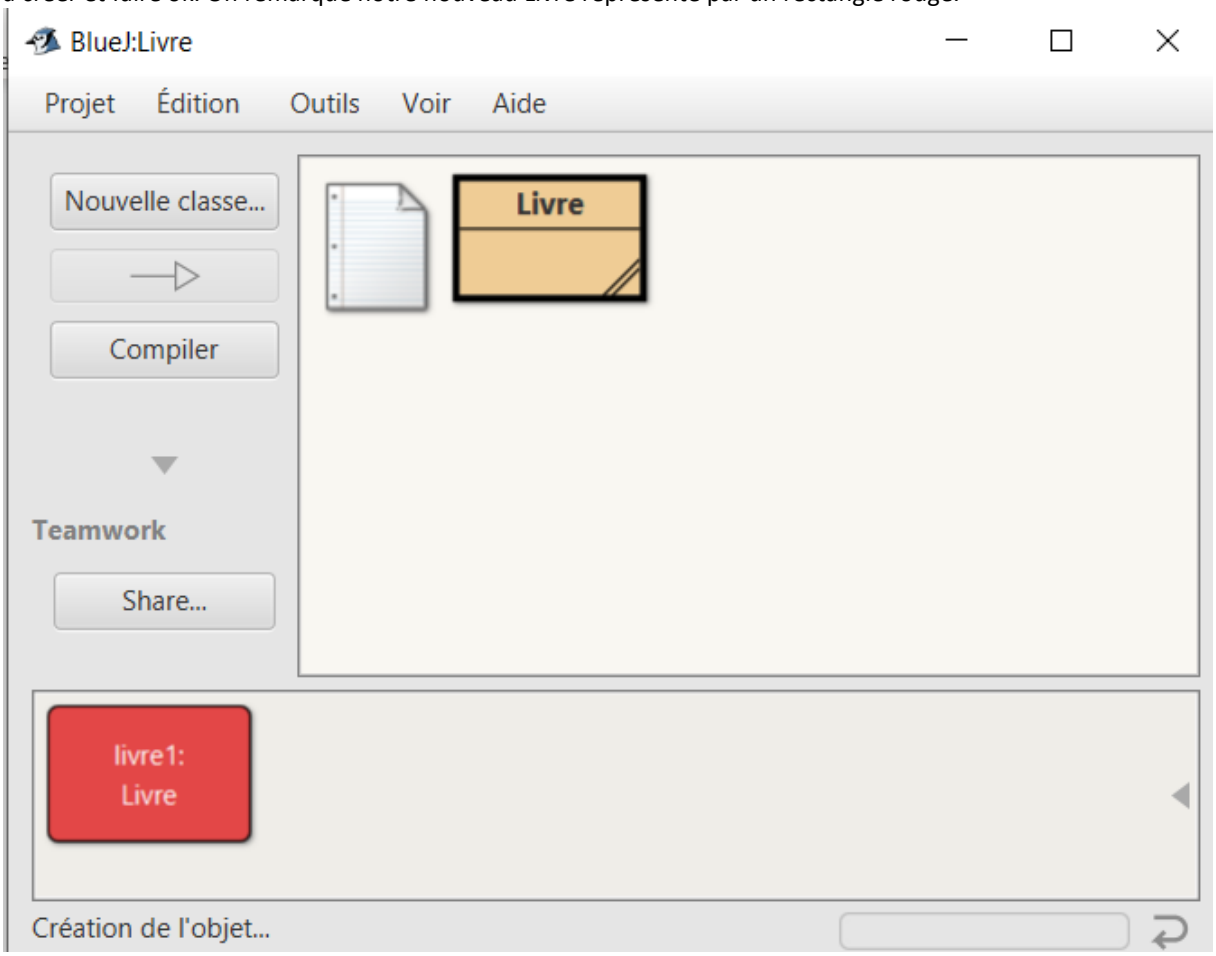
5. *Compilez la classe*

Pour compiler notre classe Livre précédemment créée, il faire clic droit sur le rectangle Livre, puis cliquer sur « Compiler ». Après la compilation, on remarque tout en bas la note ***« Compilation terminée ».***



6. *Instanciation de la classe*

Pour donner vie à notre classe livre, il faudra instancier notre classe Livre afin de créer un objet livre. Pour cela, il faut faire clic droit sur la classe livre, puis cliquer sur « new Livre ()», saisir le nom de l'objet à créer et faire ok. On remarque notre nouveau Livre représenté par un rectangle rouge.



7. *Ajout des attributs*

Pour éditer notre classe Livre, il faudra double cliquer sur la classe « Livre » ou faire clic droit puis éditer. Pour gérer nos livres, et ainsi pouvoir les vendre par la suite, nous attribuerons un titre à chaque livre, ainsi qu'un prix de vente d'origine.
On ajoute donc 2 attributs qui correspondront au titre du livre et à son prix, avec les getters et setters (qui servent à obtenir et modifier les attributs d'une classe)

```
public class Livre
{
    // variables d'instance - remplacez l'exemple qui suit par le vôtre
    private String titre = "Histoire de l'Art";
    private double prix = 50;

    /**
     * Constructeur d'objets de classe Livre
     */
    public Livre(){ /* initialisation des variables d'instance */ }


    /** Getters & setters
     *
     */
    public String getTitre()
    {
        // Insérez votre code ici
        return this.titre;
    }

    public void setTitre(String titre) {
        this.titre = titre;
    }


    public double getPrix()
    {
        return this.prix;
    }

    public void setPrix(int prix)
    {
        this.prix = prix;
    }
}
```

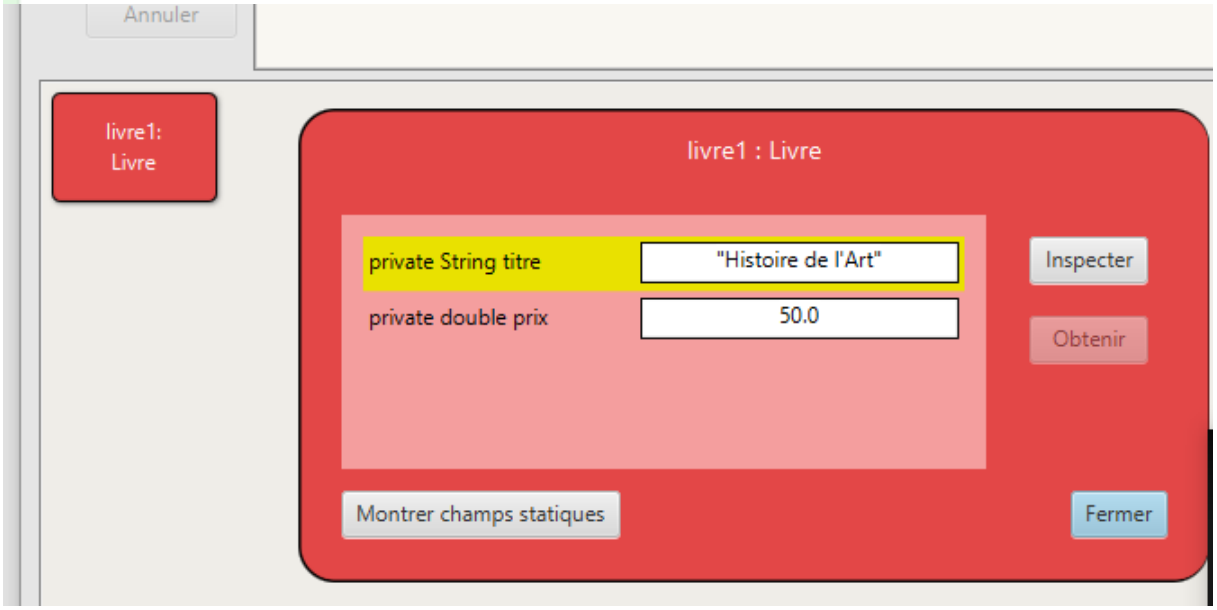
Comment attirer la clientèle ? Une promotion, un prix réduit est une des façons les plus courantes de faire. Pour cela, créons une méthode promotion, qui à partir d'une promotion, déterminera le nouveau prix d'un livre.

```
/**
 * Méthode qui calcule le nouveau prix d'un livre à partir d'une promotion
 * @param  promotion le pourcentage de la promotion (ex 0.3)
 */

public void promotion(double promotion)
{
    this.prix = (1 - promotion) * this.prix;
}
```

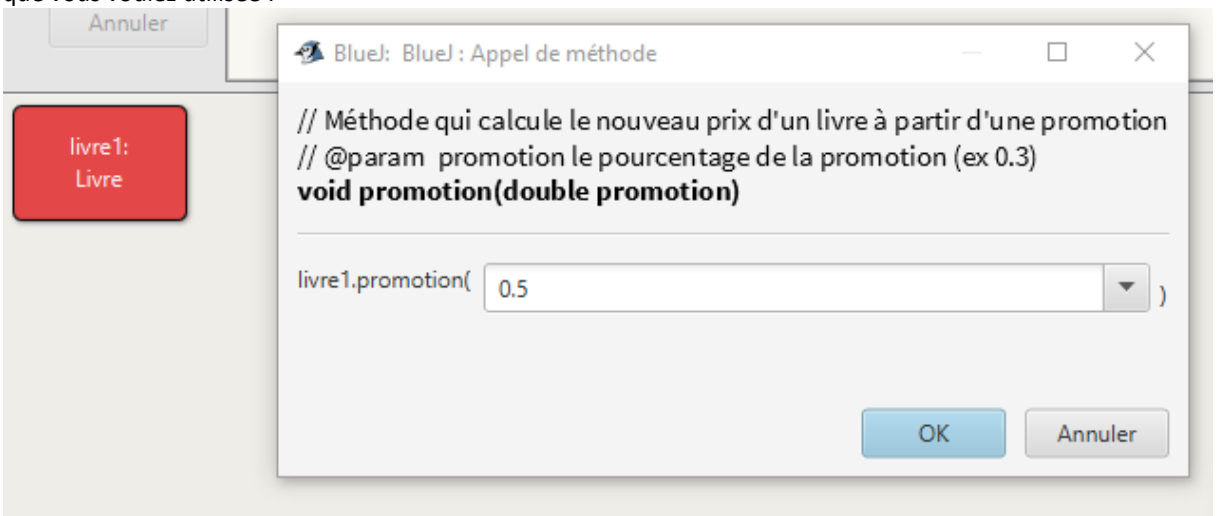
8. *Création d'objet après modification*

Donnez un titre et un prix au livre dans les attributs d'instance puis instanciez de nouveau un livre, en exécutant interactivement la méthode et en visualisant son effet sur l'état de l'objet créé.
Dans notre cas, notre premier Livre s'appellera « Histoire de l'Art », et est issu d'une rare collection, son prix de base sera donc de 50€.

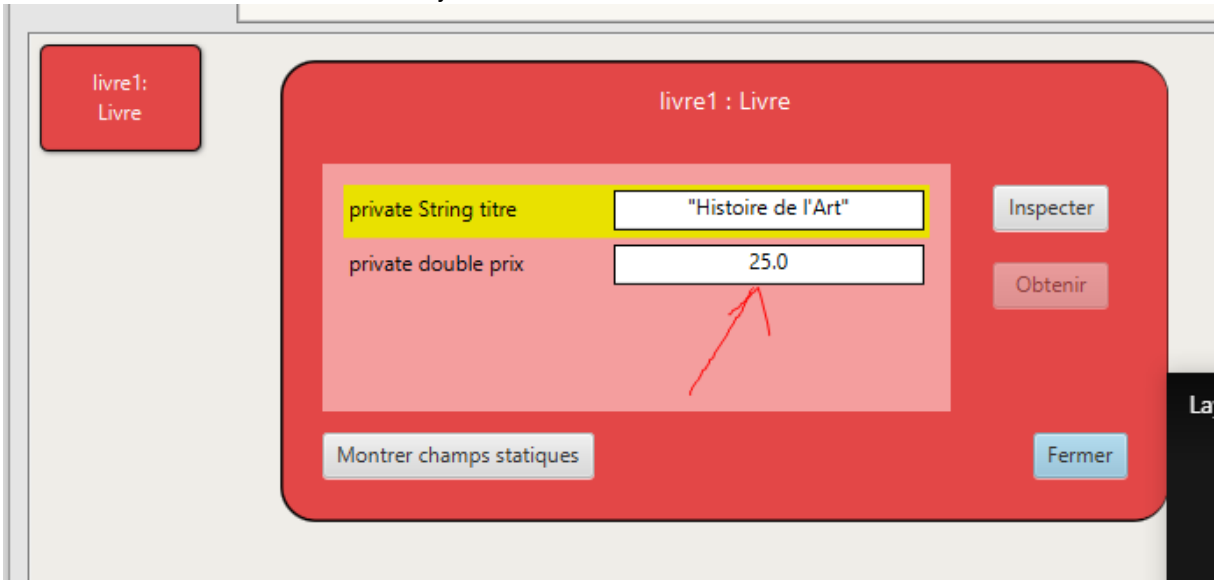
```
private String titre = "Histoire de l'Art";
private double prix = 50;
```



Essayons donc la méthode codée ! Effectuez un clic droit sur l'instance livre1, puis cliquez sur la méthode que vous voulez utilisée :



Observez à nouveau les attributs de l'objet.



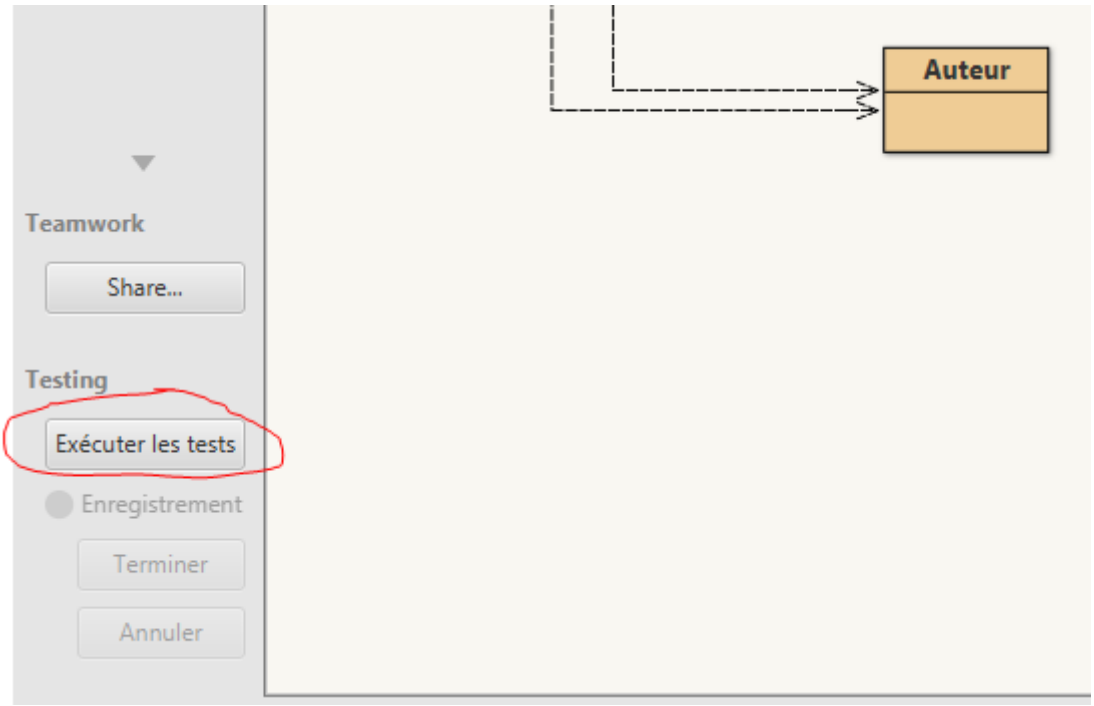
9. *Tests unitaires*

Les tests unitaires sont primordiaux pour vérifier le bon déroulement de parties ciblées dans notre projet. Idéalement, et éventuellement, nous devrons couvrir l'ensemble du projet à 100% grâce à ces fameux tests unitaires. Pour cela, nous allons créer un programme de tests adapté au projet.

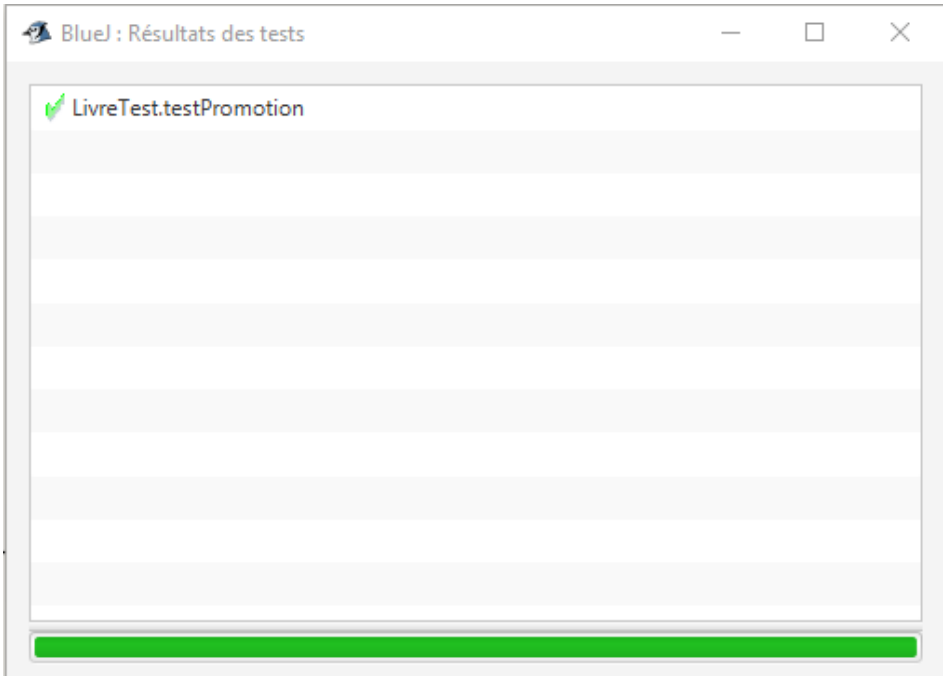
Rajoutez la méthode de test pour promotion en faisant un clic droit sur la classe de Test, puis écrire le test unitaire : en utilisant assertEquals, nous pouvons comparer le résultat attendu et le résultat obtenu, et retourner un booléen qui déterminera la bonne exécution (ou non du test).

```java
@Test
public void testPromotion()
{
    Livre livre2 = new Livre();
    livre2.promotion(0.5);
    assertEquals(25, livre2.getPrix(), 0.001);
}
```

Compilez et exécutez les tests :



Vérifiez que le test est valide. Bravo ! Une barre verte s'affiche qui confirme le bon déroulement du test.



10. Rajout d'une autre classe

Mais vous nous demandez... Si deux livres ont le même titre, mais aussi le même prix... Comment les différencie-on ?
Ajoutons une seconde classe Auteur et associons-là (de façon unidirectionnelle) à la classe Livre, avec une multiplicité 0..1 à 0..1.

```
public class Livre
{
    // variables d'instance - remplacez l'e
    private String titre = "Histoire de l'A
    private double prix = 50;
    private Auteur auteur;
```

11. Instanciation des classes et liaison des objets

Puis, ajoutez-y une méthode qui collabore avec la méthode de la classe Livre pour obtenir un résultat basé sur la contribution des objets des deux classes :
Nous voudrons bien obtenir le nombre total d'exemplaires vendus pour un auteur.
Dans la classe Auteur, rajoutez cette méthode :

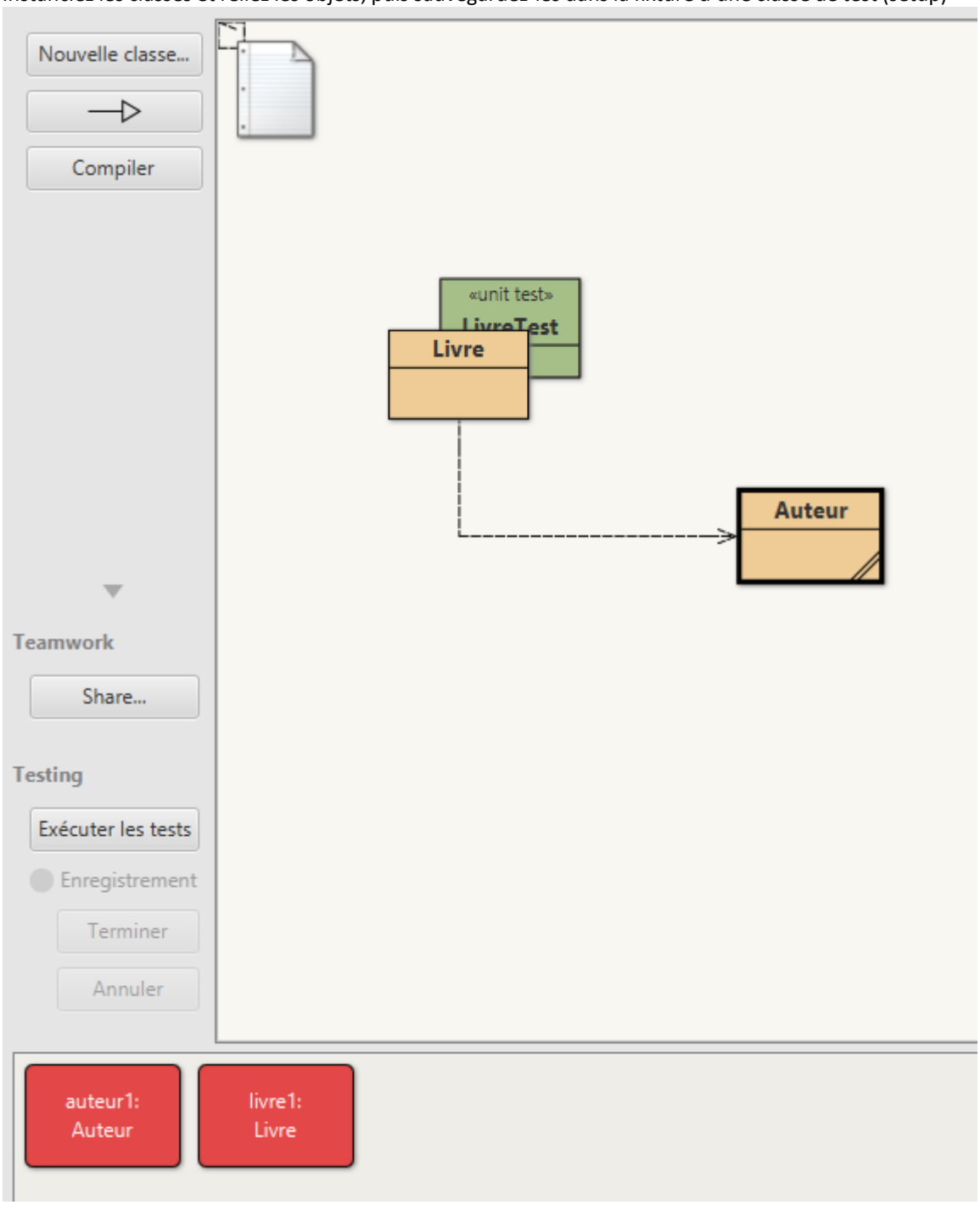
```
public int vente(int nbExemplairesVendus)
{
    this.nbVentes += nbExemplairesVendus;
    return this.nbVentes;
}
```

Dans la classe Livre, rajoutez un attribut auteur qui représentera l'auteur du livre puis un attribut nbVendus. Il faudra rajouter aussi la méthode vente qui est la méthode collaborative, puis un getter & setter pour avoir l'auteur du livre.

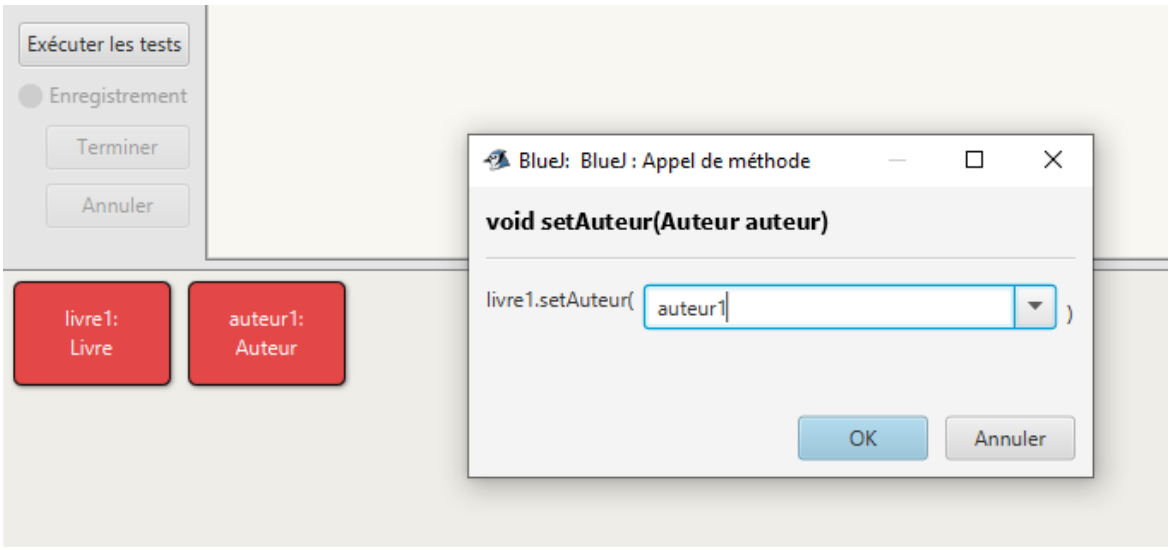
Voilà, vous pourrez maintenant attribuer un auteur à un livre !

```java
public class Livre
{
    // variables d'instance - remplacez l'exemple qui suit par le vôtre
    private String titre = "Histoire de l'Art";
    private double prix = 50;
    private Auteur auteur = new Auteur();
    private int nbVendus = 0;
    /**
     * Constructeur d'objets de classe Livre
     */
    public Livre(){ /* initialisation des variables d'instance */ }

    /**
     * Méthode qui calcule le nouveau prix d'un livre à partir d'une promotion
     * @param  promotion le pourcentage de la promotion (ex 0.3)
     */

    public void promotion(double promotion)
    {
        this.prix = (1-promotion)*this.prix;
    }

    public String toString()
    {
        return "Le livre " + this.titre + " coûte " + this.prix + "€";
    }

    public int vente(int nbExemplairesVendus)
    {
        return this.nbVendus = this.auteur.vente(nbExemplairesVendus);
    }


    public void setAuteur(Auteur auteur) {
        this.auteur = auteur;
    }

    public Auteur getAuteur() {
        return this.auteur;
    }
```

Instanciez les classes et reliez les objets, puis sauvegardez-les dans la fixture d'une classe de test (setup)

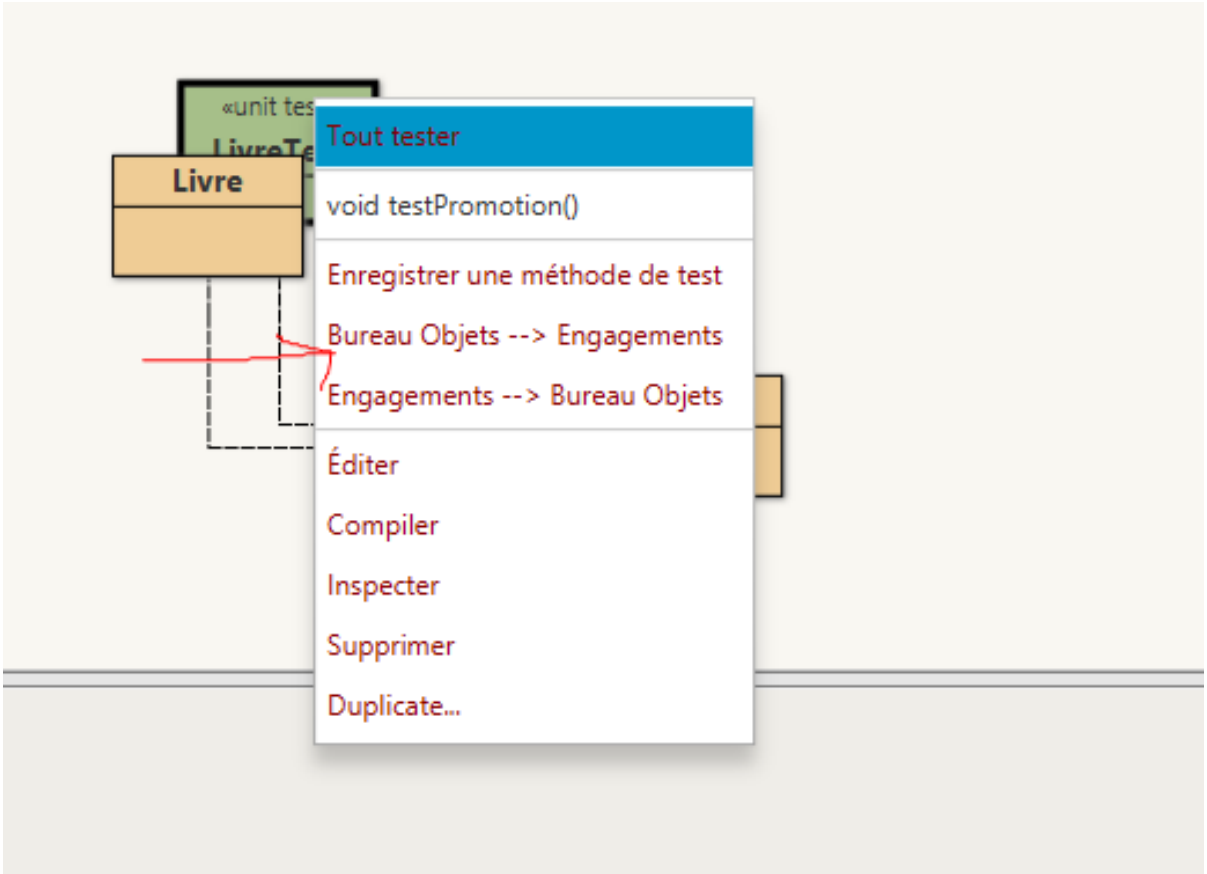


Effectuez la méthode setAuteur() de Livre et y mettre en input « auteur1 » pour associer les 2 instances.



Essayez d'utiliser la méthode vente d'auteur, et constater la nouvelle valeur dans livre.

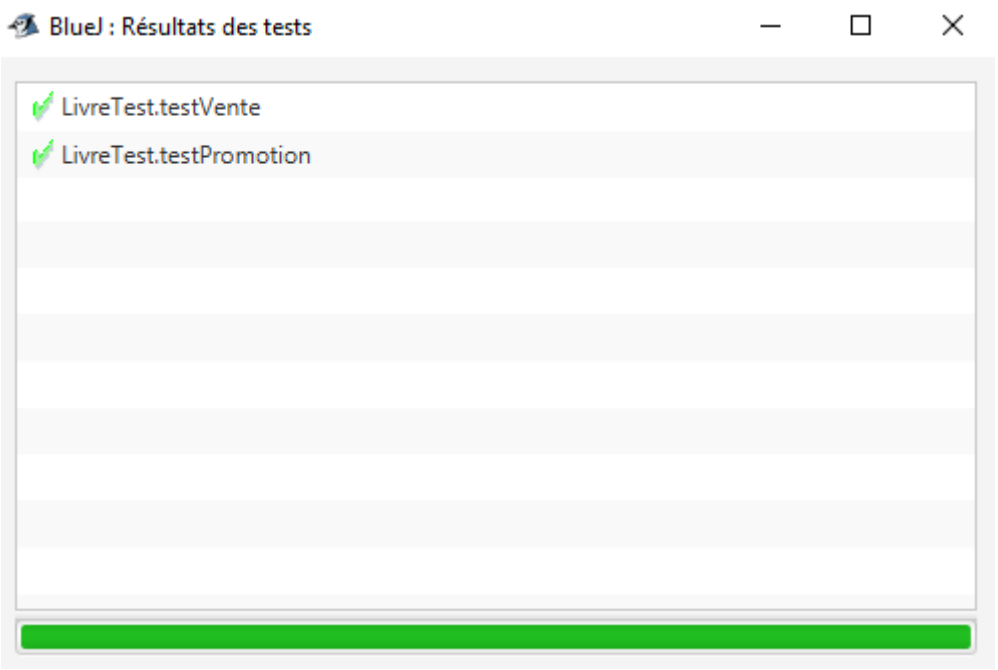
Enfin, sauvegardons cette configuration dans la fixture de la classe de test LivreTest grâce à ce bouton : (clic droit -> bureau objets → engagements)



12. Tests Unitaires

Créez interactivement une méthode de test qui utilise la fixture et montrez le résultat de son exécution et la couleur de la barre

```
@Test
public void testVente()
{
    livre1.setAuteur(auteur1);
    assertEquals(50, livre1.vente(50));
    assertEquals(65, livre1.vente(15));
}
```



Eclipse et jUnit

13. Installer Eclipse

Eclipse est un IDE, le plus couramment utilisée pour coder en Java. Le logiciel est universel, extensible et polyvalent ! Alors utilisons à notre avantage tous les atouts d'Eclipse !

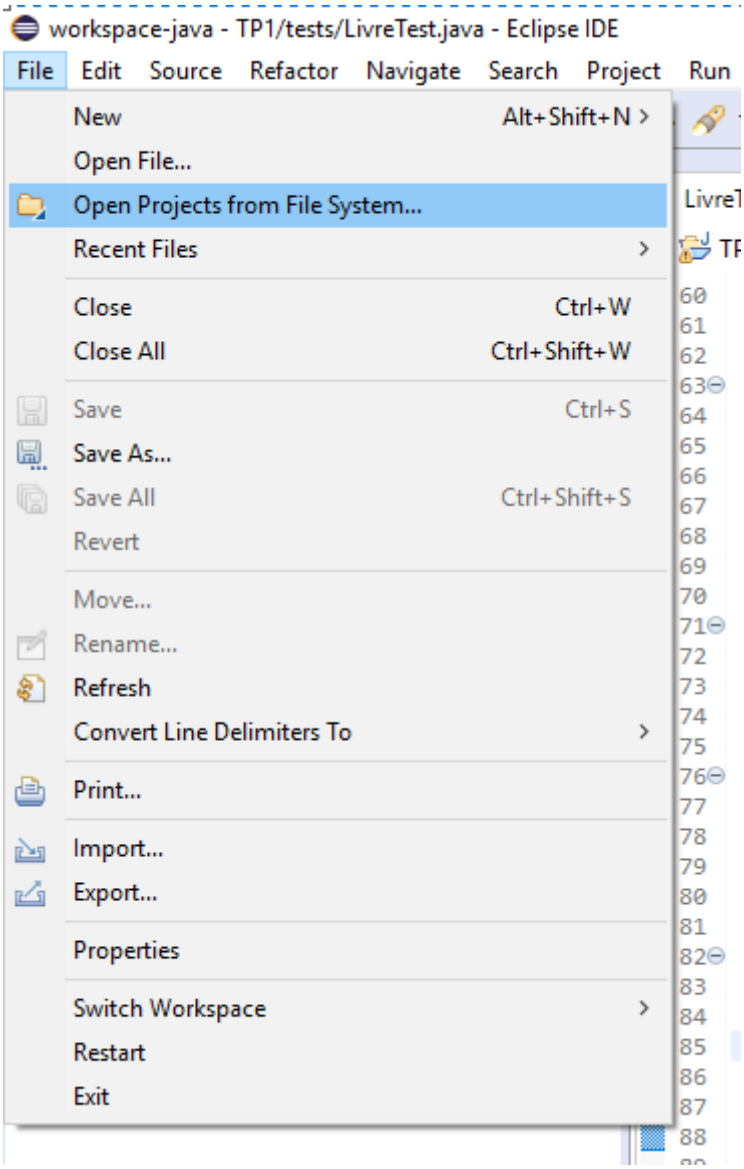
Allez sur le suite suivant et téléchargez la version correspondante à votre OS : https://www.eclipse.org/downloads/packages/

Installez Eclipse, puis lancez le logiciel.

14. Importez les classes élaborées en BlueJ et organisez-les dans un package dédié

Nous voulons évidemment utiliser le travail fait précédemment…

Allez dans File -> Open Projects from File System



Puis cliquez sur « Directory » et sélectionnez le dossier dans lequel vous avez écrit le code dans BlueJ.

*15. Implémentez une association bidirectionnelle « 0..1 à * » en l'encapsulant bien et testez unitairement sa robustesse.*

Nous voulons donc qu'un auteur soit automatiquement attribué à un Livre lorsqu'il est créé. Ca ne vous semblait pas bizarre d'avoir un Livre sans que personne ne l'ait écrit ?

Créons une association bidirectionnelle : un auteur peut avoir écrit 0 ou plusieurs livres, et un livre a été un écrit par un seul et même auteur.

Nous aurons donc une Liste de livres pour l'auteur et un seul Auteur pour un même livre.

Rajoutons donc cette liste de livres puis les méthodes addLivre et removeLivre, ainsi que son setter et son getter.

```
public class Auteur
{
    // variables d'instance - remplacez l'exemple qui suit par le vôtre
    private String nom = "Laye";
    private int nbVentes = 0;
    private List<Livre> livres = new ArrayList<Livre>();

    /**
     * Constructeur d'objets de classe Auteur
     */
    public Auteur()
    {
    }

    public List<Livre> getLivres() {
        return livres;
    }

    public void setLivres(List<Livre> livres) {
        this.livres = livres;
    }

    public void addLivre(Livre livre) {
        livre.setAuteur(this);
        this.livres.add(livre);
    }

    public void removeLivre(Livre livre) {
        this.livres.remove(livre);
    }
```

Ainsi, dans la fonction addLivre, nous avons un setAuteur qui va associer l'auteur au livre, avant d'ajouter ce dernier dans la liste de livres de l'auteur, puis rédigeons les tests unitaires pour vérifier si un livre a bien été ajouté, .

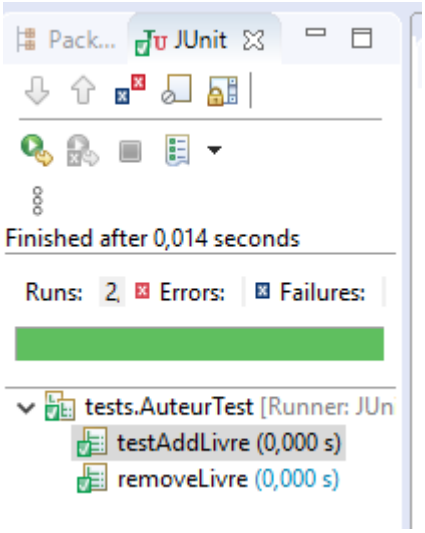
Vérifions nos méthodes avec les tests unitaires vus précédemment : rajoutez une classe AuteurTest.

Nous vérifions avec plusieurs tests :

- Si un livre a bien été ajouté avec addLivre() ;
- Si un livre a bien été enlevé avec removeLive() ;
- Si un auteur a bien été associé à un livre après un addLivre () ;

```
1  package tests;
2  import static org.junit.Assert.*;
3
4  import org.junit.Before;
5  import org.junit.Test;
6
7  import src.Auteur;
8  import src.Livre;
9
0  public class AuteurTest {
1
2      @Test
3      public void testAddLivre()
4      {
5          Auteur auteur = new Auteur();
6          Livre livre = new Livre();
7          auteur.addLivre(livre);
8          // vérifie si le livre a bien été ajouté
9          assertTrue(auteur.getLivres().contains(livre));
0          // vérifie l'encapsulation, c'est à dire si l'auteur a bien été associé au livre
1          assertEquals(auteur, livre.getAuteur());
2      }
3
4      @Test
5      public void removeAddLivre()
6      {
7          Auteur auteur = new Auteur();
8          Livre livre = new Livre();
9          auteur.removeLivre(livre);
0          // vérifie si le livre a bien été enlevé
1          assertFalse(auteur.getLivres().contains(livre));
2      }
3
4  }
5
```

Voici le résultat des tests unitaires :



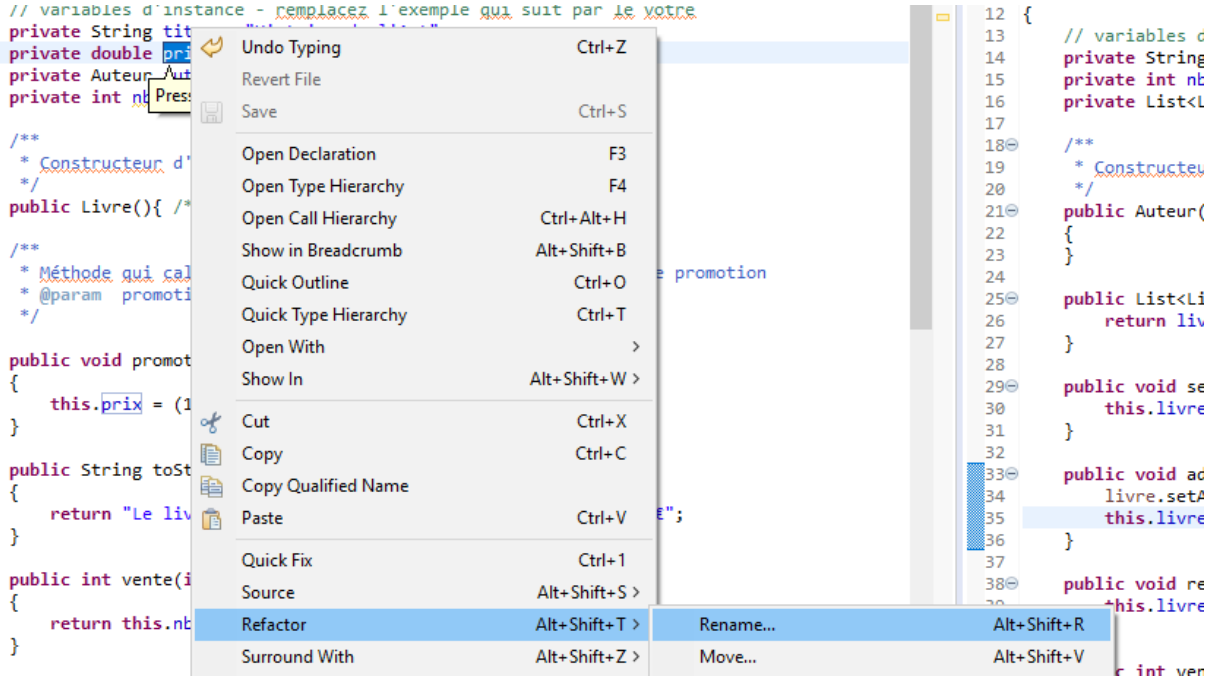
16. Illustrez l'usage de deux techniques de refactoring (ex : rename et extractMethod)

Rename : Peut-être qu'une méthode a été mal nommée dès le début - par exemple, quelqu'un a créé la méthode à la hâte et n'a pas pris soin de bien la nommer.

Ou peut-être que la méthode était bien nommée au début, mais qu'au fur et à mesure que sa fonctionnalité s'est développée, le nom de la méthode a cessé d'être un bon descripteur… La méthode de refactoring « rename » est votre ami dans ces cas là.

```
    // variables d'instance - remplacez l'exemple qui suit par le votre
    private String titre = "Histoire de l'Art";
    private double prix = 50;
    private Auteur auteur;
    private int nbVendus = 0;

    /**
     * Constructeur d'objets de classe Livre
     */
    public Livre(){ /* initialisation des variables d'instance */ }

    /**
     * Méthode qui calcule le nouveau prix d'un livre à partir d'une promotion
     * @param  promotion le pourcentage de la promotion (ex 0.3)
     */

    public void promotion(double promotion)
    {
        this.prix = (1-promotion)*this.prix;
    }

    public String toString()
    {
        return "Le livre " + this.titre + " coûte " + this.prix + "€";
    }
```

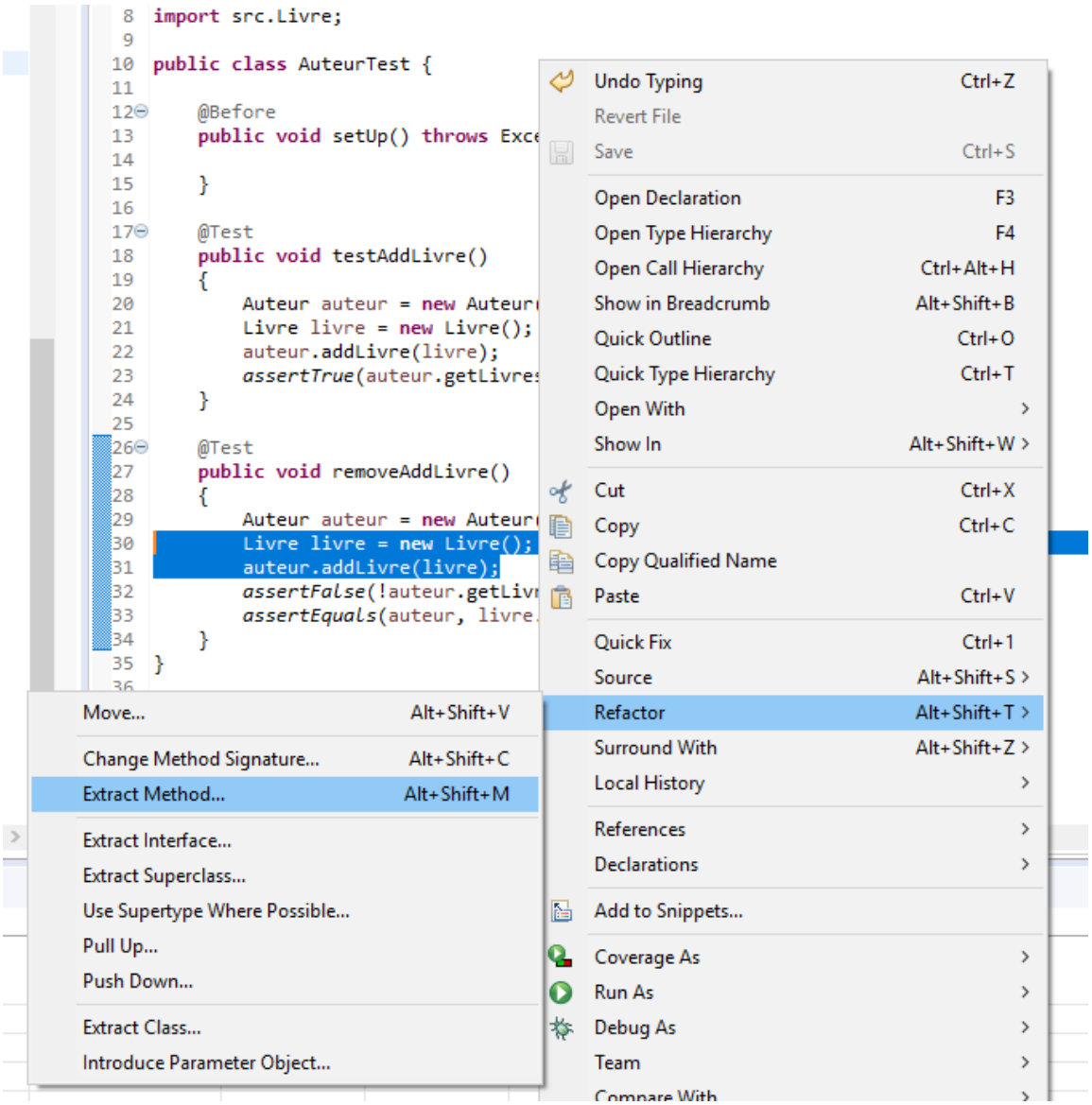
Pour rename : clic droit -> refactor -> rename



Mettre le nouveau nom de la variable (ou de la méthode), puis appuyer sur entrer. Dans ce cas, nous changeons « prix » en « prixVente »

```
    // add a un cote = set de l autre
    // variables d'instance - remplacez l'exemple qui suit par le vôtre
    private String titre = "Histoire de l'Art";
    private double prixVente = 50;
    private Auteur auteur;
    private int nbVendus = 0;

    /**
     * Constructeur d'objets de classe Livre
     */
    public Livre(){ /* initialisation des variables d'instance */ }

    /**
     * Méthode qui calcule le nouveau prix d'un livre à partir d'une promotion
     * @param  promotion le pourcentage de la promotion (ex 0.3)
     */

    public void promotion(double promotion)
    {
        this.prixVente = (1-promotion)*this.prixVente;
    }

    public String toString()
    {
        return "Le livre " + this.titre + " coûte " + this.prixVente + "€";
    }
```

ExtractMethod : Plus il y a de lignes dans une méthode, plus il est difficile de comprendre ce que fait la méthode. C'est la raison principale de ce remaniement.




```
    @Test
    public void removeAddLivre()
    {
        Auteur auteur = new Auteur();
        Livre livre = extracted(auteur);
        assertFalse(!auteur.getLivres().contains(livre));
        assertEquals(auteur, livre.getAuteur());
    }

    private Livre extracted(Auteur auteur) {
        Livre livre = new Livre();
        auteur.addLivre(livre);
        return livre;
    }
}
```

De plus, si on veut rappeler ces quelques lignes de code, on pourra juste appeler la fonction extraite.

17. Trouvez et parcourez le site officiel de jUnit. Lisez l'article « Test infected » et proposez une amélioration équivalente adaptée à votre exemple.

« Moins vous écrivez de tests, moins vous êtes productif et moins votre code est stable. Moins vous êtes productif et précis, plus vous ressentez de la pression. »

Source : http://junit.sourceforge.net/doc/testinfected/testing.htm

Prenons un exemple : dans notre code, plus précisément dans notre classe AuteurTest, nous avions mis dans le setup un auteur et un livre, mais nous ne l'avions pas utilisé, et au lieu de cela on devait instancier un nouvel auteur à chaque méthode de test. Rectifions cela :

```
public class AuteurTest {

    private Auteur auteur1;
    private Livre livre1;
    @Before
    public void setUp() throws Exception {
        auteur1 = new Auteur();
        livre1 = new Livre();
    }

    @Test
    public void testAddLivre()
    {
        // Auteur auteur = new Auteur();
        Livre livre = extracted(auteur1);
        assertTrue(auteur1.getLivres().contains(livre));
    }

    @Test
    public void testAuteur() {
        assertNotNull(auteur1);
    }

    @Test
    public void testRemoveAddLivre()
    {
        // Auteur auteur = new Auteur();
        Livre livre = extracted(auteur1);
        auteur1.removeLivre(livre);
        assertFalse(auteur1.getLivres().contains(livre));
    }

    @Test
    public void testBidirectionnel()
    {
        // Auteur auteur = new Auteur();
        Livre livre = extracted(auteur1);
        assertEquals(auteur1, livre.getAuteur());
    }
```

Ainsi, dans chaque fonction, au lieu de créer un nouvel auteur, on utilise simplement celui utilisé dans le setup. C'est le premier exemple dans l'article mentionné, on y gagne du temps et de la lisibilité.

18. Exécutez les tests en ligne de commande

Télécharger le jar du junit correspond à la version utilisée pour le projet : https://junit.org/junit4/

Ensuite, mettez les fichiers téléchargés (junit-4.13 et hamcrest-core-1.3.jar) dans votre projet.

Sous Windows, ouvrez le CMD et placez vous dans le dossier du projet où il y a les fichiers .java .

Pour exécuter une classe Java, il faut utiliser la commande javac qui va créer les fichiers .class.

```
               6 fichier(s)          434 184 octets
               2 Rép(s)  17 463 963 648 octets libres

C:\Users\DiKeLa M'Babane\Documents\M2IF DAUPHINE\Agilité\SimpleClasse>javac Auteur.java

C:\Users\DiKeLa M'Babane\Documents\M2IF DAUPHINE\Agilité\SimpleClasse>
```

En appelant la classe Auteur, la classe Livre est aussi appelée car elle est instanciée dans la classe Auteur. Après avoir compilé les deux classes principales, nous allons compiler les deux classes de Test.

Utilisons les deux fichiers téléchargés :

```
C:\Users\DiKeLa M'Babane\Documents\M2IF DAUPHINE\Agilité\SimpleClasse>javac -cp junit-4.13.jar;. AuteurTest.java

C:\Users\DiKeLa M'Babane\Documents\M2IF DAUPHINE\Agilité\SimpleClasse>
```

```
C:\Users\DiKeLa M'Babane\Documents\M2IF DAUPHINE\Agilité\SimpleClasse>javac -cp junit-4.13.jar;. LivreTest.java

C:\Users\DiKeLa M'Babane\Documents\M2IF DAUPHINE\Agilité\SimpleClasse>
```

Vous venez de compiler les tests, maintenant, lançons les !

Nous n'avons pas de classe main pour lancer l'application qui effectuera le test. Par substitution nous ajouterons l'option org.junit.runner.JUnitCore « nom de la classe de test »

```
C:\Users\DiKeLa M'Babane\Documents\M2IF DAUPHINE\Agilité\SimpleClasse>java -cp junit-4.13.jar;hamcrest-core-1.3.jar;
. org.junit.runner.JUnitCore LivreTest
JUnit version 4.13
...........
Time: 0,014

OK (11 tests)


C:\Users\DiKeLa M'Babane\Documents\M2IF DAUPHINE\Agilité\SimpleClasse>
```

```
C:\Users\DiKeLa M'Babane\Documents\M2IF DAUPHINE\Agilité\SimpleClasse>java -cp junit-4.13.jar;hamcrest-core-1.3.jar;
. org.junit.runner.JUnitCore AuteurTest
JUnit version 4.13
.........
Time: 0,011

OK (9 tests)


C:\Users\DiKeLa M'Babane\Documents\M2IF DAUPHINE\Agilité\SimpleClasse>
```

19. « Si le gars a la moindre possibilité de faire une erreur, il le fera »

En programmation, il faut toujours partir de la base que l'utilisateur qui rentrera les input dans l'IHM pourra faire des erreurs, car aucun être humain n'est parfait.

C'est pourquoi, il faut prendre des mesures et réfléchir aux possibilités qui pourraient rendre une fonctionnalité incompatible, voir incompréhensible.
Par exemple, un prix ne peut être négatif. On peut donc prendre des mesures de prévention et ne pas changer le prix lorsque ce dernier est négatif. Il peut en revanche être nul, donc ne pas oublier le signe « = »

```
public void setPrix(int prix)
{
    this.prixVente = prix;
}
```

====>

```
public void setPrix(int prix)
{
    if (prix >= 0 ) {
        this.prixVente = prix;
    }
}
```

Pareil pour la promotion : on part du fait que la promotion doit toujours être un DOUBLE positif. Si on reçoit par exemple un int, on doit pouvoir attraper l'erreur et mieux la repérer.

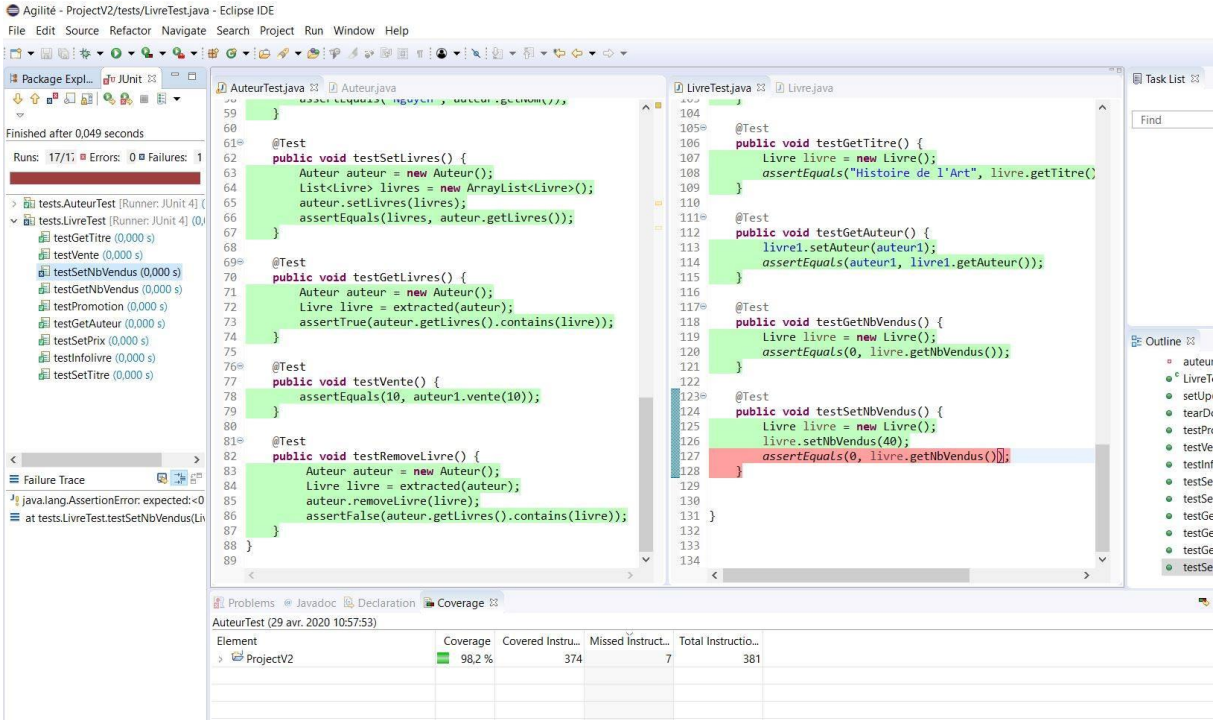
```
public void promotion(double promotion)
{
    // promotion = 0.5 => 50% de réduction
    if (promotion > 0 || promotion < 1) {
        try {
            this.prixVente  = (1 - promotion) * this.prixVente;
        } catch (Exception e) {
            // Code pour gérer l'exception
        }
    }
}
```

Cascade de tests

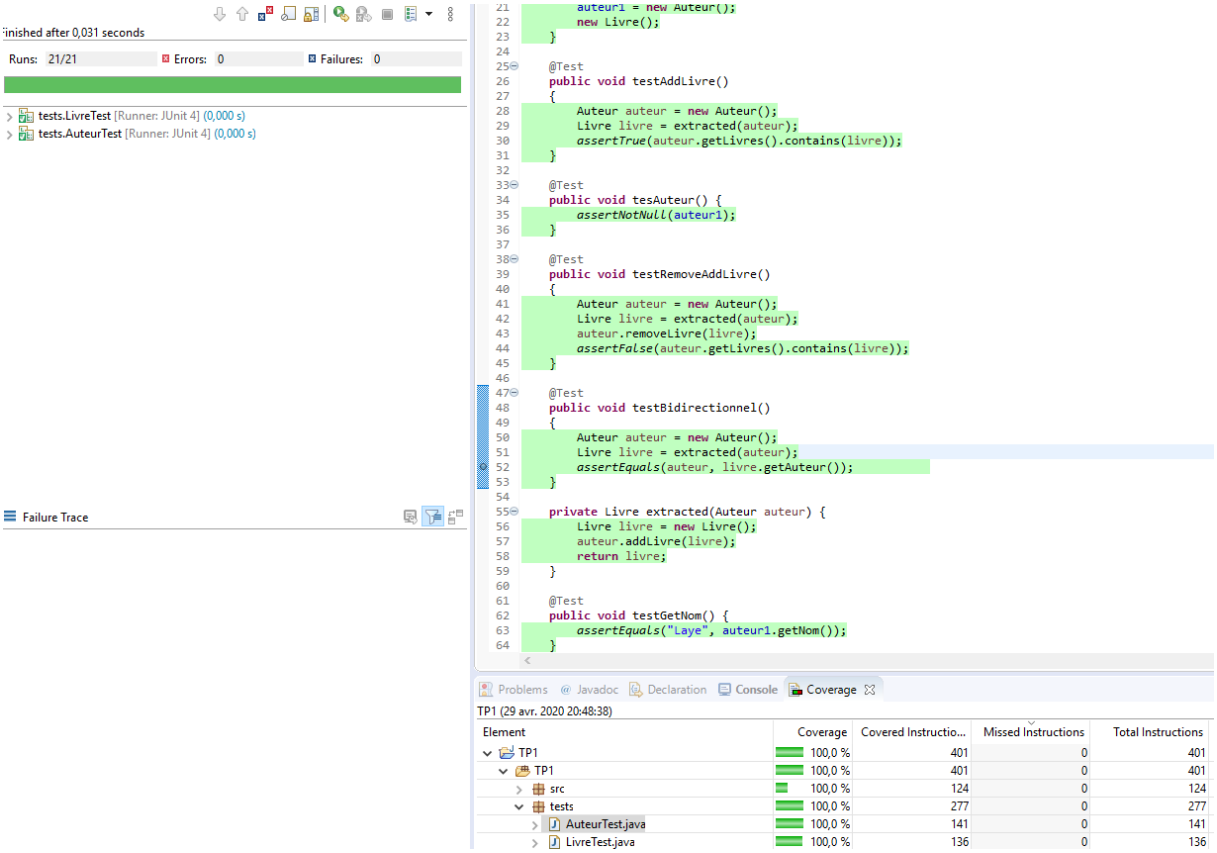
Pas	Remarque
Avancés	Bouchons, mutators …
Couverture du code	S'assurer que tout le code cible est couvert par les tests. Puis s'assurer que les tests sont exécutés de manière complète. Exemple : eclEmma.
Vérifier les domaines de validité des paramètres	S'assurer que les méthodes qui doivent vérifier le domaine de validité des paramètres le font bien. Par exemple, passer des paramètres licites, puis illicites, si besoin en écrivant des tests unitaires distincts
Ajouter les tests des exceptions	Avec try catch ou avec des annotations, pour s'assurer que des exceptions sont bien levées si besoin.
Tester les méthodes complexes	En s'inspirant des scénarii de test des UserStories
Tester les méthodes simples	Autres que les getters et setters. Tester aussi bien la valeur de retour de la méthode cible que l'impact sur l'état de l'objet testé. (attribut qui change de valeur suite à l'appel de la méthode). Usage de getters. Bien distinguer les commandes des query.
Tester les associations	Les assocations sont implémentées à l'aide d'attributs de type non primitif et des méthodes associées. Cas: multiplicité maxi 1 (attribut monovalué) ou * (collection typée, instanciable par ArrayList<>). Cas: bidirectionnel et unidirectionnel (voir exemple: Popey / Olive) . Possibilité d'utiliser des templates (aussi dit : code snipet)
Tester les setters	Sur la base des getters et du constructeur
Tester le constructeur	Sur la base des getters
Tester les getters	Sur la base des valeurs par défaut des attributs
Supprimer le SystemOut et le main du code cible	Ce n'est plus le main qui exécutera les classes cible, mais jUnit qui exécutera le code des classes de test qui feront exécuter les classes cible. Nous n'avons donc plus besoin de surveiller la console, ni de la polluer avec des sysout.
On teste JUnit une seule fois	Une toute première fois, produire une barre rouge (pour s'assurer que jUnit détecte les assertions qui ne sont pas respectées) puis la verdir indiquant le bon fonctionnement de nommage Java. Par la suite garder la convention que le vert indique le fonctionnement attendu.
Convention de nommage Java	Adopter un style de codage en citant sa référence, ou en proposer un nouveau style en l'explicitant. En phase avec la pratique "propriété collective du code"

Cette cascade de test nous permet de suivre pas à pas des indications pour aborder la démarche à suivre lorsqu'on effectue nos tests unitaires, et de manière générale, vérifier si notre code fait bien ce que l'on attend. On le lit de bas en haut.

1. Convention de nommage : sur ce projet, nous nous sommes basés sur la convention traditionnelle de Java. Par exemple, on peut s'inspirer du site d'oracle : https://www.oracle.com/technetwork/java/codeconventions-135099.html
2. On regarde si JUnit renvoie une barre rouge pour s'assurer du bon fonctionnement de JUnit



3. On a bien enlevé les System.out.println() et le main() pour pouvoir raccrocher notre projet à d'autres composants
4. On teste :
    a. Les getters
    b. Les constructeurs
    c. Les setters
    d. Les associations
    e. Les méthodes simples
    f. Les méthodes complexes
5. On rajoute ensuite les tests des exceptions (que nous n'avons pas dans notre projet car il utilisera un exemple très simple)
6. On vérifie les domaines de validité des paramètres
7. On s'assure de la couverture du code (il doit être optimalement proche de 100%, voire même à 100%) pour s'assurer qu'on couvre bien l'ensemble de nos tests.
   Pour cela, effectuez un clic droit sur le projet, et faites un « Run Coverage ».



Cucumber – Behavior Drived Development

Cucumber est un outil logiciel qui prend en charge le développement comportemental. L'élément central de l'approche BDD de Cucumber est son analyseur de langage ordinaire appelé Gherkin. Il permet de spécifier les comportements attendus du logiciel dans un langage logique que les clients peuvent comprendre. En tant que tel, Cucumber permet l'exécution de la documentation des fonctionnalités rédigée dans un texte orienté métier.

Pour l'installation de cucumber, nous vous recommandons de suivre ces instructions sur le site officiel :

https://cucumber.io/docs/installation/java/

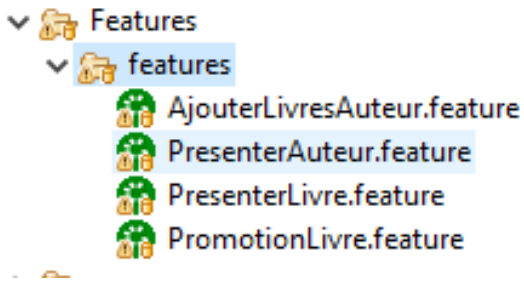
Grâce à Cucumber, on peut rédiger nos User Stories et ainsi les tester de manière simplifié et plus faccile à comprendre lorsque nous avons de nombreuses User Stories.

Voici les étapes à suivre :

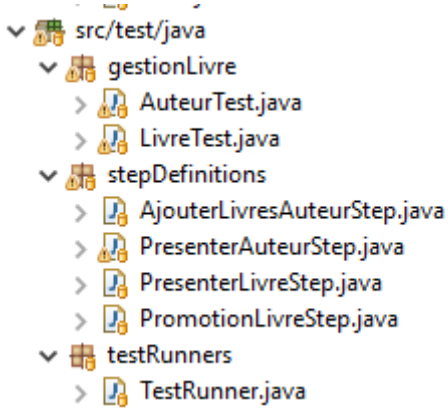
1. Réinventer des users stories qui auraient donné lieu au code qu'on a écrit (on remonte dans le temps !)
2. Planifier : comment vais-je faire cette user story plutôt qu'une autre ? en fonction de quoi et quand ? à vous de décider de ce que vous pourrez faire !
3. Ecrire le code... Mais cela est déjà fait ! On a déjà les tests unitaires ! En revanche, nous n'avons pas encore écrit les tests fonctionnel. Il faudra donc les implémenter en fonction des critères d'acceptance des user stories.
4. Archiver les implémentations (grâcea à un SVN tel que github)
5. Exécuter les tests automatiquement (user stories) avec cucumber puis obtenir des tests positifs

Commençons !

Créons un package « Features » dans lequel nous mettrons nos features, représentant nos user stories. Dans ce projet, elles sont au nombre de 4, nommées explicitement, mais vous pourrez bien sûr laisser libre cours à votre imagination pour en implémenter plus.



Ensuite, il faudra des classes pour tester ces features ainsi qu'un « test main » qui va tester l'ensemble de ces features. Créez donc un package stepDefinitions, où l'on définira les classes de test, puis un package « testRunners » où on aura notre lanceur de tests.



Sans plus tarder, voici nos users stories et nos classes de tests. Prenons l'exemple d'abord de la présentation d'un livre :

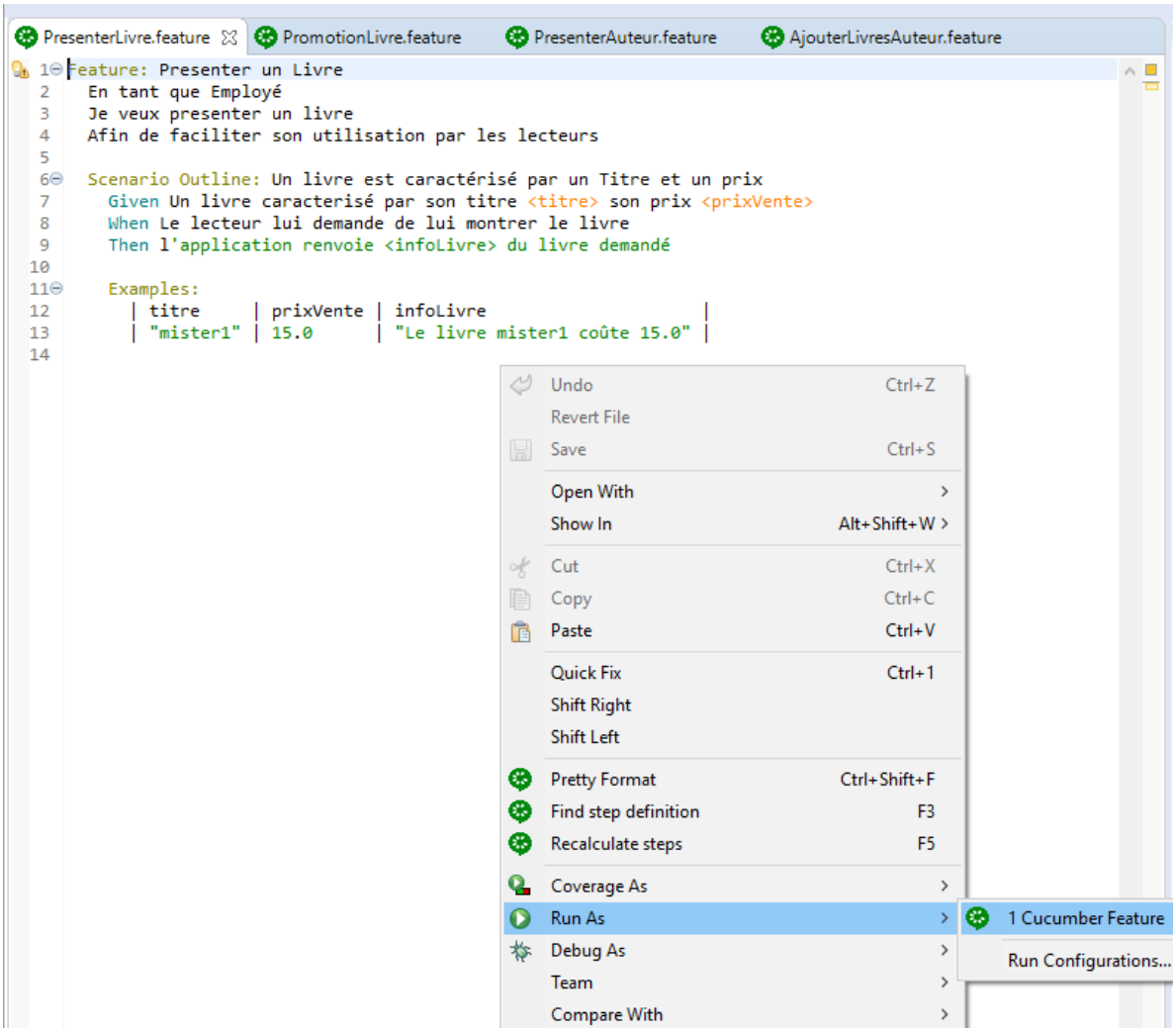
PresenterLivre.feature | PromotionLivre.feature | PresenterAuteur.feature | AjouterLivresAuteur.feature

```
1  Feature: Presenter un Livre
2    En tant que Employé
3    Je veux presenter un livre
4    Afin de faciliter son utilisation par les lecteurs
5
6    Scenario Outline: Un livre est caractérisé par un Titre et un prix
7      Given Un livre caracterisé par son titre <titre> son prix <prixVente>
8      When Le lecteur lui demande de lui montrer le livre
9      Then l'application renvoie <infoLivre> du livre demandé
10
11     Examples:
12       | titre     | prixVente | infoLivre                      |
13       | "mister1" | 15.0      | "Le livre mister1 coûte 15.0" |
14
```

On observe plusieurs mots clés :

- La partie « Given » décrit l'état du monde avant que vous ne commenciez le comportement que vous spécifiez dans ce scénario. Vous pouvez considérer cela comme les conditions préalables au test.
- La partie « When » est le comportement que vous spécifiez.
- Enfin, la section « Then » décrit les changements que vous attendez en raison du comportement spécifié.
- « Scenario Outline » nous permet de mettre un « Example » afin de vérifier plus rapidement nos tests.

Passons maintenant au test en lui-même. Où est le code ?



En effectuant un Run As -> Cucumber Feature, vous vous apercevrez que les tests ne sont pas encore définis.

Vous pouvez constater que le « texte en français » est traduit en code de test. A vous de jouer ! Dans le message d'erreur, vous aurez les « steps » qui n'auront pas été définis. Le snippet que vous devrez implémenter vous-même est donné. Copiez-le et collez-le dans la classe de test « PresenterLivreStep.java »

```
    Then l'application renvoie "Le livre misterI coute 15.0" du livre demande # null

Undefined scenarios:
/C:/Users/33652/Downloads/Cours 2019-2020/Méthodes Agiles d'Ingénierie Logicielle/Projet/Project_

1 Scenarios (1 undefined)
3 Steps (3 undefined)
0m0,189s


You can implement missing steps with the snippets below:

@Given("Un livre caracterisé par son titre {string} son prix {double}")
public void un_livre_caracterisé_par_son_titre_son_prix(String string, Double double1) {
    // Write code here that turns the phrase above into concrete actions
    throw new cucumber.api.PendingException();
}

@When("Le lecteur lui demande de lui montrer le livre")
public void le_lecteur_lui_demande_de_lui_montrer_le_livre() {
    // Write code here that turns the phrase above into concrete actions
    throw new cucumber.api.PendingException();
}

@Then("l'application renvoie {string} du livre demandé")
public void l_application_renvoie_du_livre_demandé(String string) {
    // Write code here that turns the phrase above into concrete actions
    throw new cucumber.api.PendingException();
}
```

A vous de jouer pour que la classe teste bien la présentation d'un livre !

PresenterLivreStep.java | PromotionLivreStep.java | PresenterAuteurStep.java | AjouterLivresAuteurStep

```java
package stepDefinitions;

import static org.junit.Assert.assertEquals;

import cucumber.api.java.en.Given;
import cucumber.api.java.en.Then;
import cucumber.api.java.en.When;
import gestionLivre.Auteur;
import gestionLivre.Livre;

public class PresenterLivreStep {

    private Livre livre;
    private String info;
    private Auteur auteur;

    @Given("Un livre caracterisé par son titre {string} son prix {double}")
    public void un_livre_caracterisé_par_son_titre_son_prix(String titre, Double prix) {
        livre = new Livre(titre, prix);
        auteur = new Auteur();
        livre.setAuteur(auteur);

    }

    @When("Le lecteur lui demande de lui montrer le livre")
    public void le_lecteur_lui_demande_de_lui_montrer_le_livre() {
        info = livre.infoLivre();

    }

    @Then("l'application renvoie {string} du livre demandé")
    public void l_application_renvoie_du_livre_demandé(String information) {
        assertEquals(information, info);

    }

}
```

En relançant le feature..

```
  Scenario Outline: Un livre est caractérisé par un Titre et un prix          # /C:/Users/33652/Downloads/Cours 2019-2020/Méthodes Agiles d'Ingénierie
    Given Un livre caracterisé par son titre "mister1" son prix 15.0          # PresenterLivreStep.un_livre_caracterisé_par_son_titre_son_prix(String,D
    When Le lecteur lui demande de lui montrer le livre                       # PresenterLivreStep.le_lecteur_lui_demande_de_lui_montrer_le_livre()
    Then l'application renvoie "Le livre mister1 coûte 15.0" du livre demandé # PresenterLivreStep.l_application_renvoie_du_livre_demandé(String)

1 Scenarios (1 passed)
3 Steps (3 passed)
0m0,163s
```

Les 3 steps (Given-When-Then) sont bien passés ! A vous de coder le reste des features.

PromotionLivre :

PresenterLivre.fe... | PromotionLivre.fe... | PresenterAuteur.f... | AjouterLivresAut...

```
Feature: Promotion d'un livre
  En tant que Auteur
  Je veux faire une promotion sur le livre
  Afin de réduire le prix d'un livre pour mieux le vendre

  Scenario Outline: calcul du nouveau prix du livre
    Given un livre et sa promotion <promotion>
    When L'auteur veut faire une promotion
    Then le <nouveauPrix> doit être calculé

    Examples:
      | promotion |
      | 0.5       |

  Scenario Outline: refus de promotion négatif
    Given un livre et sa promotion <promotion> négatif
    When L'auteur veut faire une promotion
    Then le syustème refuse avec le <messageErreur>

    Examples:
      | promotion | messageErreur  |
      | -0.1      | "prix négatif" |
```

PresenterLivreStep.java | PromotionLivreStep.java | PresenterAuteurStep.java

```
package stepDefinitions;
import cucumber.api.java.en.Given;


public class PromotionLivreStep {

    private Livre livre;
    private double nouveauPrix;
    private String msgError;

    @Given("un livre et sa promotion {double}")
    public void un_livre_et_sa_promotion(Double promotion) {
        livre = new Livre("Livre de la jungle", 15);
        livre.promotion(promotion);
    }


    @When("L'auteur veut faire une promotion")
    public void l_auteur_veut_faire_une_promotion() {
        nouveauPrix = livre.getPrix();
    }



    @Then("le <nouveauPrix> doit être calculé")
    public void le_nouveauPrix_doit_être_calculé() {
        Assert.assertEquals(7.5, nouveauPrix, 0.01);
    }



    @Given("un livre et sa promotion {double} négatif")
    public void un_livre_et_sa_promotion_négatif(Double promotion) {
        livre = new Livre("Livre de la jungle", 15);
        msgError = livre.promotion(promotion);
    }

    @Then("le syustème refuse avec le {string}")
    public void le_syustème_refuse_avec_le(String string) {
        Assert.assertEquals(msgError, "promotion négative");
    }

```

Problems | Javadoc | Declaration | Console

```
<terminated> PromotionLivre.feature [Cucumber Feature] C:\Program Files\Java\jre1.8.0_251\bin\javaw.exe  (24 mai 2020 à 17:10:25 – 17:10:26)
    Examples:

  Scenario Outline: refus de promotion négatif     # /C:/Users/33652/Downloads/Cours 2019-2020/Méthodes Agiles d'Ingénierie Logicielle/Projet/Project_Agility/Agilit
    Given un livre et sa promotion -0.1 négatif    # PromotionLivreStep.un_livre_et_sa_promotion_négatif(Double)
    When L'auteur veut faire une promotion         # PromotionLivreStep.l_auteur_veut_faire_une_promotion()
    Then le syustème refuse avec le "prix négatif" # PromotionLivreStep.le_syustème_refuse_avec_le(String)

2 Scenarios (2 passed)
6 Steps (6 passed)
0m0,166s
```

PresenterAuteur :

PresenterLivre.featu... | PromotionLivre.featu... | PresenterAuteur.feat... | AjouterLivresAuteu...

```
Feature: Présenter un auteur
  En tant que Employé
  Je veux presenter un auteur
  Afin de donner ses informations aux lecteurs

  Scenario Outline: Le nom de l'auteur et son nombre de ventes sont donnés
    Given Un auteur caracterisé par son nom <nom> et son nombre de ventes <nbVentes>
    When Un lecteur demande la présentation de l'auteur
    Then l'application renvoie <infoAuteur>

    Examples:
      | nom    | nbVentes | infoAuteur               |
      | "Mich" |       50 | "Mich a vendu 50 livres" |
```

PresenterLivreStep.java | PromotionLivreStep.java | PresenterAuteurStep.java | AjouterLivresAuteurStep.java

```
package stepDefinitions;


import org.junit.Assert;


public class PresenterAuteurStep {
    private Auteur auteur;
    private String infoAuteur;

    @Given("Un auteur caracterisé par son nom {string} et son nombre de ventes {int}")
    public void un_auteur_caracterisé_par_son_nom_et_son_nombre_de_ventes(String nom, Integer nbVentes) {
        auteur = new Auteur(nom, nbVentes);
    }

    @When("Un lecteur demande la présentation de l'auteur")
    public void un_lecteur_demande_la_présentation_de_l_auteur() {
        infoAuteur = auteur.infoAuteur();
    }

    @Then("l'application renvoie {string}")
    public void l_application_renvoie(String information) {
            Assert.assertEquals(information, infoAuteur);
    }

}
```

Problems | Javadoc | Declaration | Console

```
<terminated> PresenterAuteur.feature [Cucumber Feature] C:\Program Files\Java\jre1.8.0_251\bin\javaw.exe  (24 mai 2020 à 17:11:10 – 17:11:11)
    Examples:

  Scenario Outline: Le nom de l'auteur et son nombre de ventes sont donnés    # /C:/Users/33652/Downloads/Cours 2019-2020/Méthodes Agiles d'Ingénierie Logicielle/Projet/Project_Agility/AgilityProject/Features
    Given Un auteur caracterisé par son nom "Mich" et son nombre de ventes 50 # PresenterAuteurStep.un_auteur_caracterisé_par_son_nom_et_son_nombre_de_ventes(String,Integer)
    When Un lecteur demande la présentation de l'auteur                       # PresenterAuteurStep.un_lecteur_demande_la_présentation_de_l_auteur()
    Then l'application renvoie "Mich a vendu 50 livres"                       # PresenterAuteurStep.l_application_renvoie(String)

1 Scenarios (1 passed)
3 Steps (3 passed)
0m0,166s
```

AjouterLivresAuteur :

PresenterLivre.feature | PromotionLivre.feature | PresenterAuteur.feature | AjouterLivresAuteur.feature

```
Feature: Ajouter des livres d'un auteur
  En tant que Employé
  Je veux ajouter les livres d'un auteur
  Afin d'ajouter et afficher les livres d'un auteur

  Scenario Outline: Les livres de l'auteur sont affichés
    Given Un auteur
    And un livre <titre1> et son <prixVente1>
    And un livre <titre2> et son <prixVente2>
    When l employé ajoute des livres d'un auteur
    Then l application renvoie les <infoLivres>

    Examples:
      | titre1        | prixVente1 | titre2       | prixVente2 | infoLivres |
      | "Livre Agile" | 15.0       | "Livre Java" | 10.0       | "Les livres de Laye sont : Livre Agile,Livre Java"
```

PresenterLivreStep.ja... | PromotionLivreStep.ja... | PresenterAuteurStep.j... | AjouterLivresAuteurS...

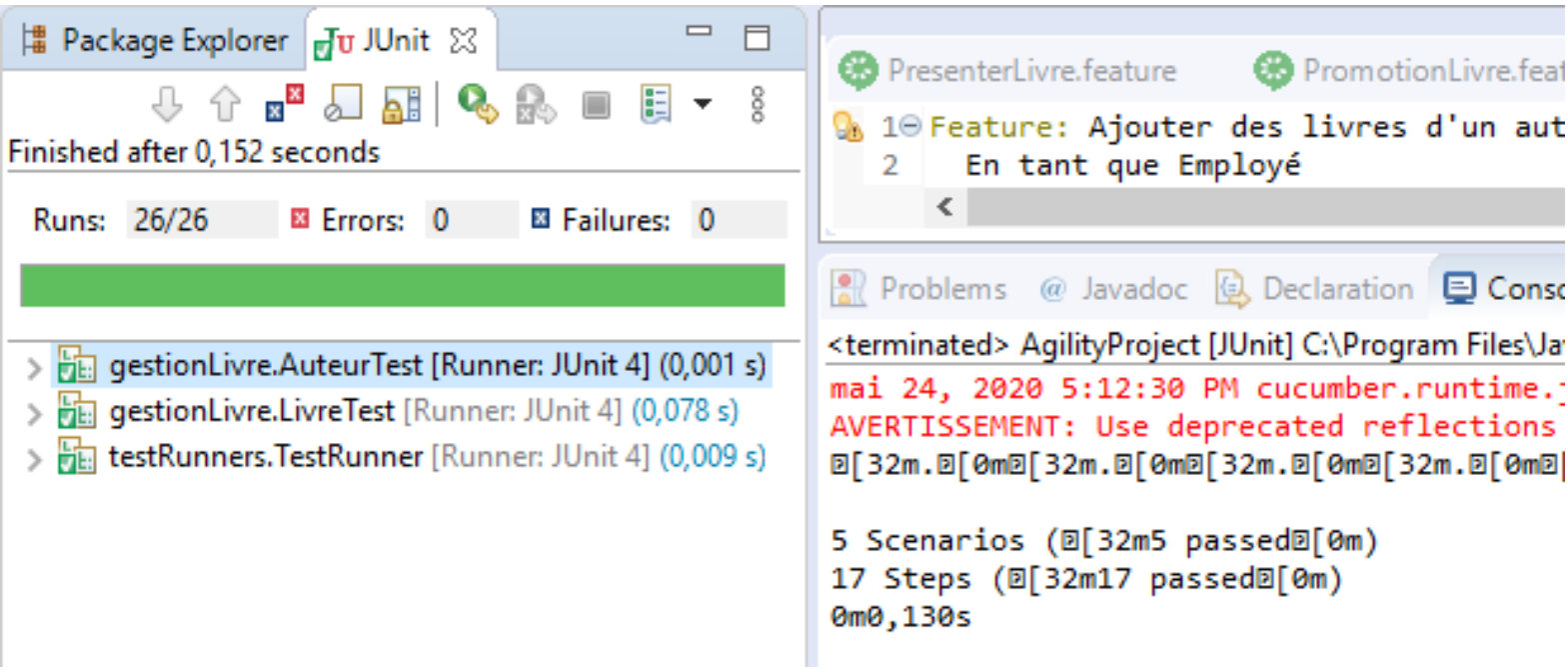
```
package stepDefinitions;
import org.junit.Assert;


public class AjouterLivresAuteurStep {
    private Auteur auteur;
    private Livre livreToAdd;
    private String infos;

    @Given("Un auteur")
    public void un_auteur() {
        auteur = new Auteur();
    }

    @Given("un livre {string} et son {double}")
    public void un_livre_et_son(String titre, Double prix) {
        livreToAdd = new Livre(titre, prix);
        auteur.addLivre(livreToAdd);
    }

    @When("l employé ajoute des livres d'un auteur")
    public void l_employé_ajoute_des_livres_d_un_auteur() {
        infos = auteur.afficherLivres();
    }


    @Then("l application renvoie les {string}")
    public void l_application_renvoie_les_livres_ajoutés_de_l_auteur(String infoLivres) {
        Assert.assertEquals(infoLivres, infos);
    }
```

Problems | Javadoc | Declaration | Console

<terminated> AjouterLivresAuteur.feature [Cucumber Feature] C:\Program Files\Java\jre1.8.0_251\bin\javaw.exe (24 mai 2020 à 17:24:33 – 17:24:34)

```
  Je veux ajouter les livres d'un auteur
  Afin d'ajouter et afficher les livres d'un auteur

  Scenario Outline: Les livres de l'auteur sont affichés # /C:/Users/33652/Downloads/Cours 2019-2020/Méthodes Agiles d'Ingénierie Logicielle/Projet/Project_Agility/AgilityProject/Features/features/AjouterLivresAuteur.feature
    Given Un auteur
    And un livre <titre1> et son <prixVente1>
    And un livre <titre2> et son <prixVente2>
    When l employé ajoute des livres d'un auteur
    Then l application renvoie les <infoLivres>

    Examples:

  Scenario Outline: Les livres de l'auteur sont affichés                          # /C:/Users/33652/Downloads/Cours 2019-2020/Méthodes Agiles d'Ingénierie Logicielle/Projet/Project_Agility/AgilityProject/Features/feature
    Given Un auteur                                                               # AjouterLivresAuteurStep.un_auteur()
    And un livre "Livre Agile" et son 15.0                                        # AjouterLivresAuteurStep.un_livre_et_son(String,Double)
    And un livre "Livre Java" et son 10.0                                         # AjouterLivresAuteurStep.un_livre_et_son(String,Double)
    When l employé ajoute des livres d'un auteur                                  # AjouterLivresAuteurStep.l_employé_ajoute_des_livres_d_un_auteur()
    Then l application renvoie les "Les livres de Laye sont : Livre Agile,Livre Java" # AjouterLivresAuteurStep.l_application_renvoie_les_livres_ajoutés_de_l_auteur(String)

1 Scenarios (1 passed)
5 Steps (5 passed)
0m0,167s
```

Enfin, en relançant le projet en tant que JUnit Test, nous avons bien tous les tests (jUnit et cucumber) qui se déroulent sans erreur.



Si vous en êtes arrivés jusqu'à là, félicitations, vous avez implémenté vos user stories avec brio !

Voici le lien du github du projet : https://github.com/Dikela/Project_Agility

Pour le faire fonctionner, faites un git clone du projet, modifiez le répertoire du jdk chez vous (dans le pom.xml), effectuez un mvn clean install en ligne de commandes dans le dossier où se trouve le pom.xml. Enfin, lancez le projet en tant que test junit.