A dark blue vertical bar runs down the left side of the page. A blue arrow points to the right from this bar, containing the date.

28/04/2020

SuperHero

La création de classe fait par les
grands

Several thin, curved lines in dark blue and light grey originate from the bottom left and sweep upwards and to the right.

Simona ARIZANOVA - Mickael PEEREN

Table des matières

Partie 1 - Bluej et Tests.....	2
1. Créer une classe SuperHero.....	2
2. Tester les méthodes d'une classe.....	5
Partie 2 – IntelliJ et JUnit.....	10
3. Liens entre deux classes	10
4. Implémentation d'un attribut de relation 0..1 - * - La bi-directionnalité	18
5. TDD.....	21
6. Refactoring.....	22
1. Rename Method.....	22
2. Extract Method.....	23
7. BDD	25
Feature	26
Partie III – Fusion de deux projets	30
Objectif.....	30
Diagramme de classe.....	30
Design Patterns	31
The End	38

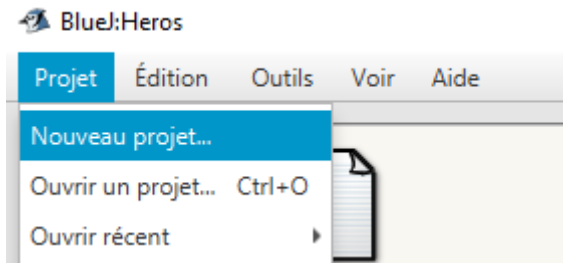
Partie 1 - BlueJ et Tests

Bienvenue à vous, super-héros ! Je suis Java-man, et je vais vous apprendre le Java et les concepts objets avec l'aide d'une application appelée BlueJ.

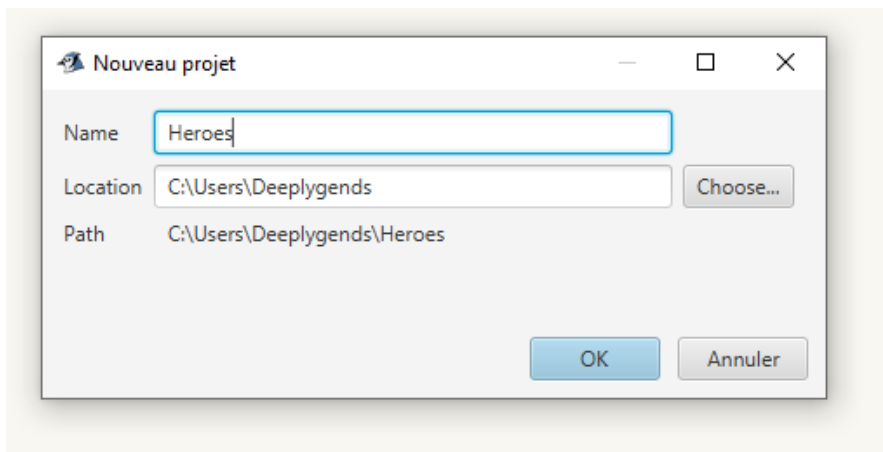
Vous allez m'aider à créer un super héros afin de combattre les méchants.

1. Créer une classe SuperHero

Créer un nouveau projet dans **BlueJ** via **Projet > Nouveau Projet ...**

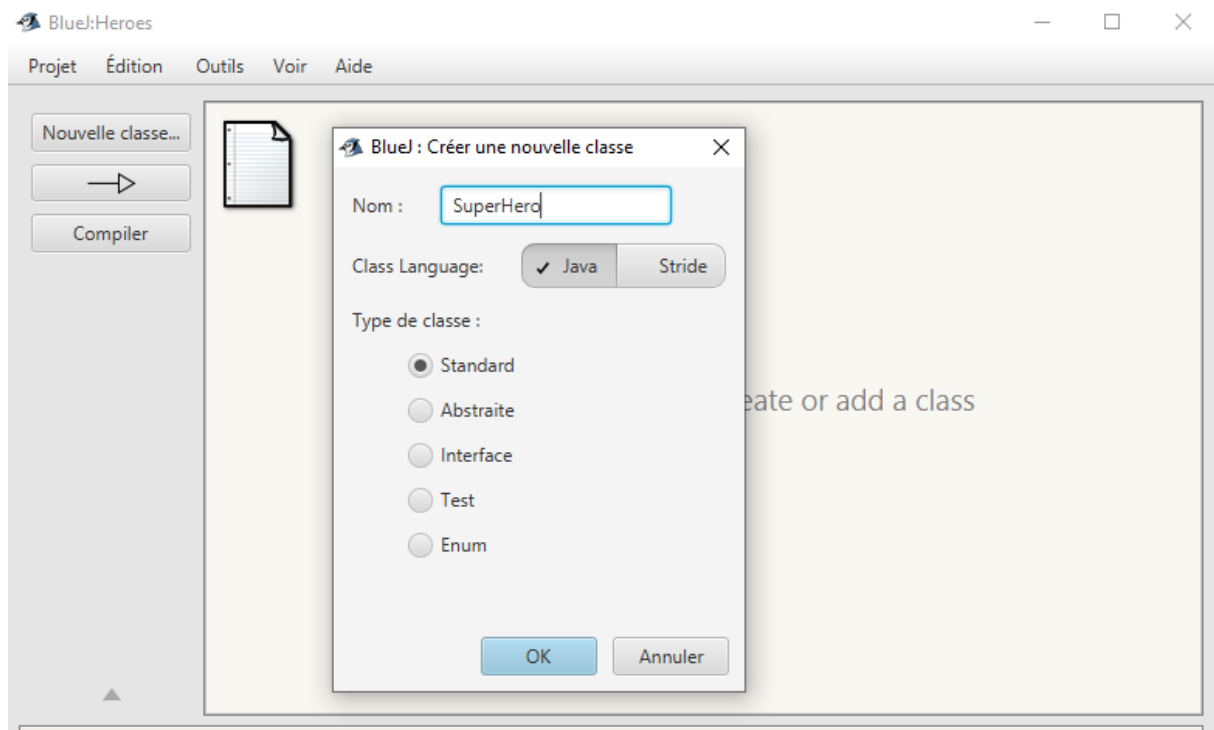


Entre le nom de projet que vous voulez (Avengers, Heroes, Rick et Morty, etc ...)

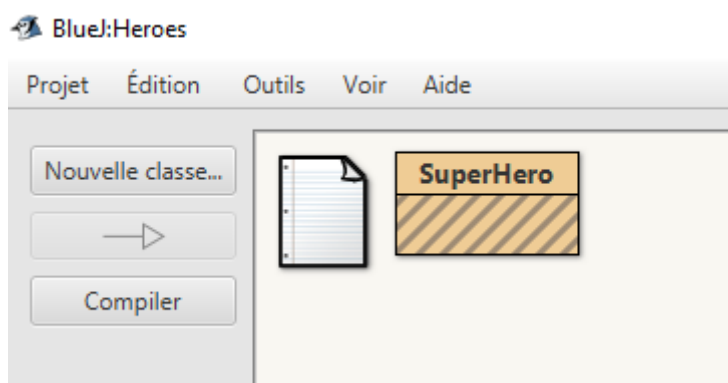


Ensuite, afin de créer notre équipe (constituée de super héros, je vous le rappelle)
, nous devons créer la classe « *SuperHero* ».

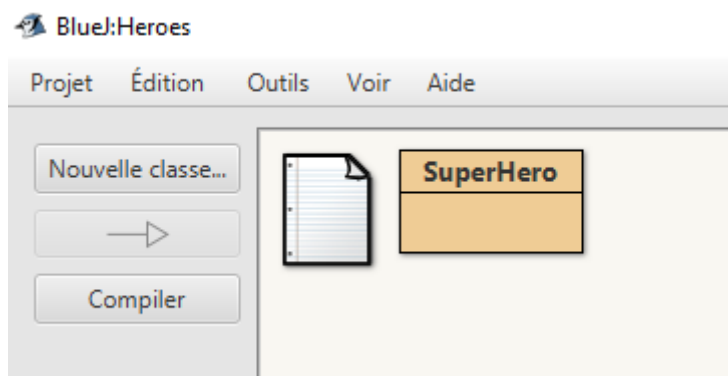




Félicitations, jeune padawan, la classe SuperHero vient d'être créée mais elle est toute barrée !



Pas de panique, Compile-man est ici pour vous aider. Cliquez sur **Compiler** et ... tadam !



Et si nous allons voir ce qu'il y a dans notre classe :

```

/**
 * Décrivez votre classe SuperHero ici.
 *
 * @author (votre nom)
 * @version (un numéro de version ou une date)
 */
public class SuperHero
{
    // variables d'instance - remplacez l'exemple qui suit par le vôtre
    private int x;

    /**
     * Constructeur d'objets de classe SuperHero
     */
    public SuperHero()
    {
        // initialisation des variables d'instance
        x = 0;
    }

    /**
     * Un exemple de méthode - remplacez ce commentaire par le vôtre
     *
     * @param y    le paramètre de la méthode
     * @return     la somme de x et de y
     */
    public int sampleMethod(int y)
    {
        // Insérez votre code ici
        return x + y;
    }
}

```

Cela ne représente pas vraiment un super-héros, apportons quelques modifications. Un Super-héros possède une identité et une certaine force. Ajoutons un champ « *identity* » de type string (chaîne de caractère) et un champ « *strenght* » de type int (entier). Ces deux champs seront privés : on ne veut pas que les méchants puissent changer la force de notre héros et la mettre à 0.

```

public class SuperHero
{
    // variables d'instance - remplacez l'exemple qui suit par le vôtre
    private String identity = "Spiderman";
    private int strength = 35;
}

```

Ensuite, nous allons ajouter les « *getter* » (méthodes d'accès pour récupérer une donnée) et les « *setter* » (pour initialiser une donnée privée) de nos deux champs.

```

public SuperHero()
{
    // initialisation des variables d'instance
}

public String getIdentity()
{
    return this.identity;
}

public void setIdentity(String identity)
{
    this.identity = identity;
}

public int getStrength()
{
    return this.strength;
}

public void setStrength(int strength)
{
    this.strength = strength;
}

```

Cela commence à ressembler à un super-héros. Mais pour rester un super-héros, il faut s'entraîner dur. Ajouter une méthode « *workOut* » qui prend en paramètre un entier (la force gagnée pendant l'entraînement) et qui va nous renvoyer la force totale du super-héros après son entraînement.

```

public int workOut(int addStrength)
{
    return this.strength + addStrength;
}

```

Très bien, notre super-héros est fort et entraîné. Mais on devrait vérifier s'il s'est bien entraîné non ? Même les super-héros trichent parfois ...

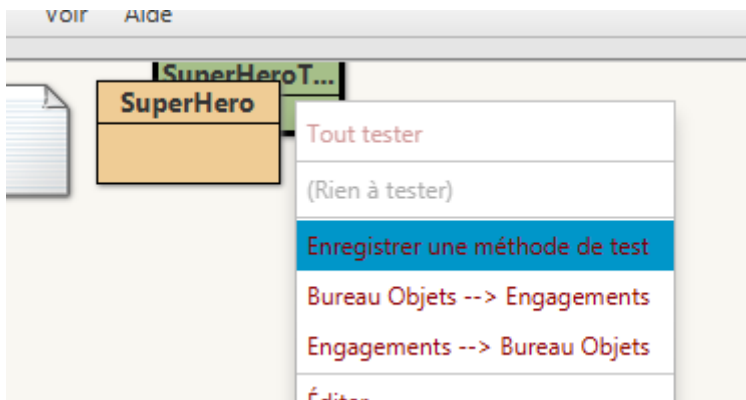
2. Tester les méthodes d'une classe

Créons une classe de test pour tester notre méthode *workOut* :

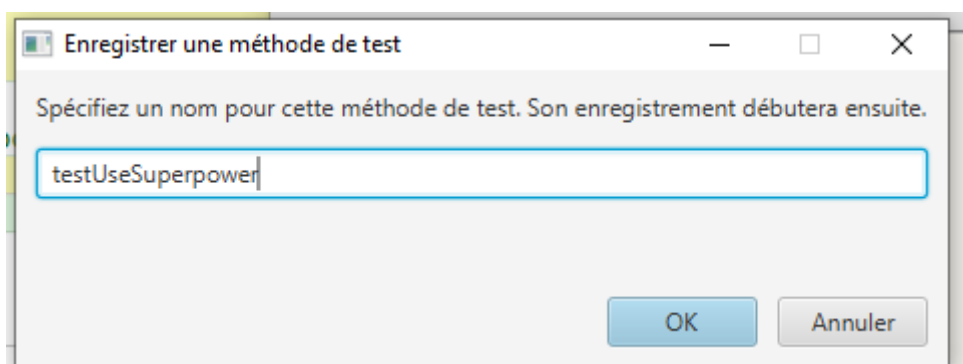
Clic droit sur la classe SuperHero > **Créer classe Test**



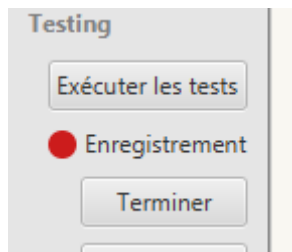
La classe de test apparait en vert, encore une fois : **Clic droit** dessus > **Enregistrer une méthode de test**.



Il faut faire attention à bien nommer notre méthode de test afin de bien savoir ce qu'elle teste, sinon on va oublier après.

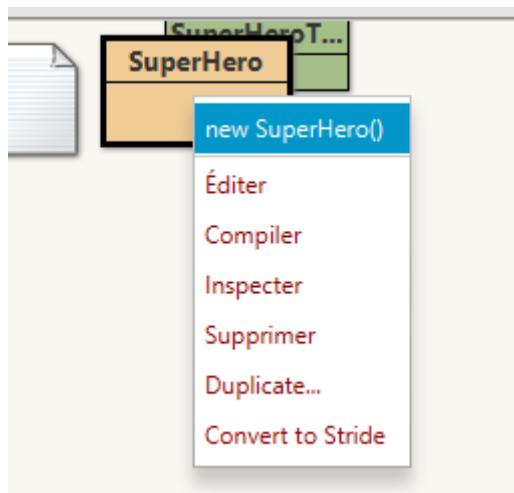


Tiens, tiens. L'enregistrement est lancé, à partir de maintenant tout ce que tu feras sera enregistré (Big Brother est là !).

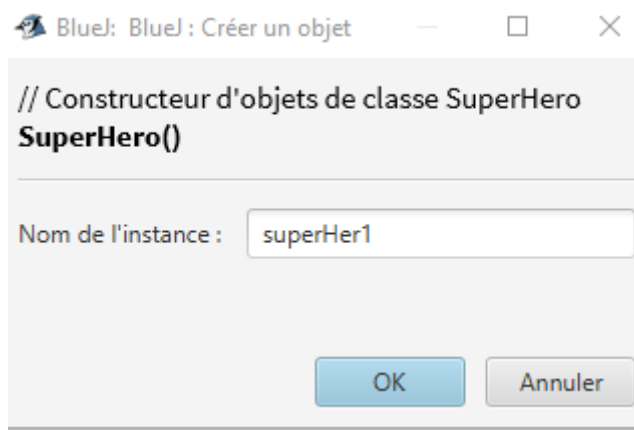


Pour tester l'entraînement d'un super-héros il faut créer une instance de « *SuperHero* » :

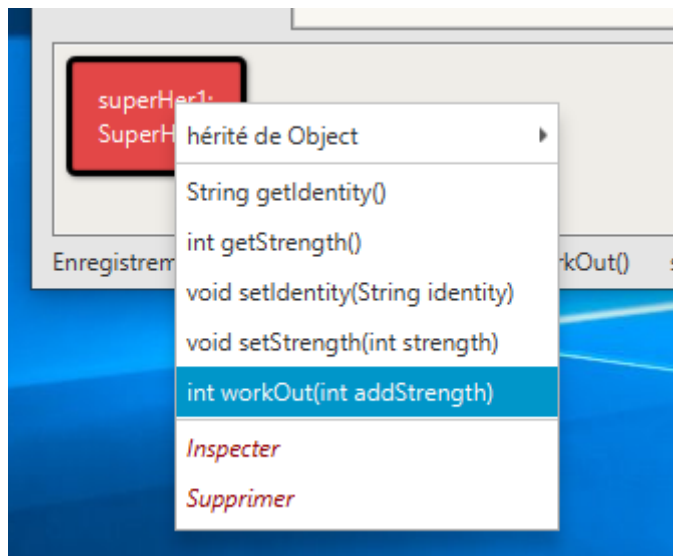
Clic droit sur la classe SuperHero > **new SuperHero()**



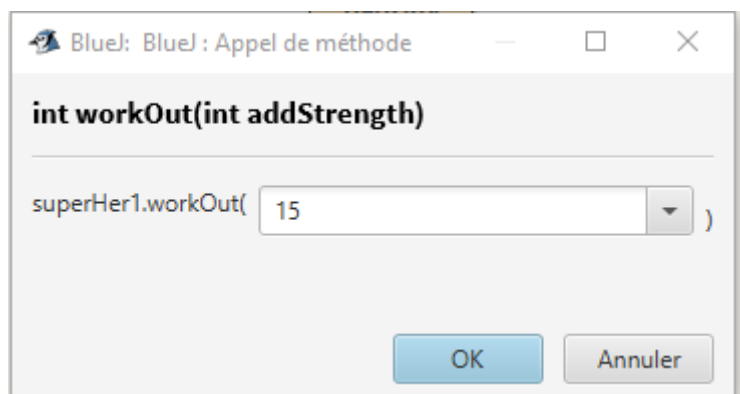
On nomme la variable comme on veut (ici superHer1). On peut aussi l'appeler Batman, Superman, Green Lantern si on veut être original.



Et enfin, on lance la méthode « *workOut* » de superHer1, notre premier super-héros. Il va faire un peu de corde à sauter, de course à pied, de natation mais aussi d'autres entraînements dignes d'un vrai super-héros !

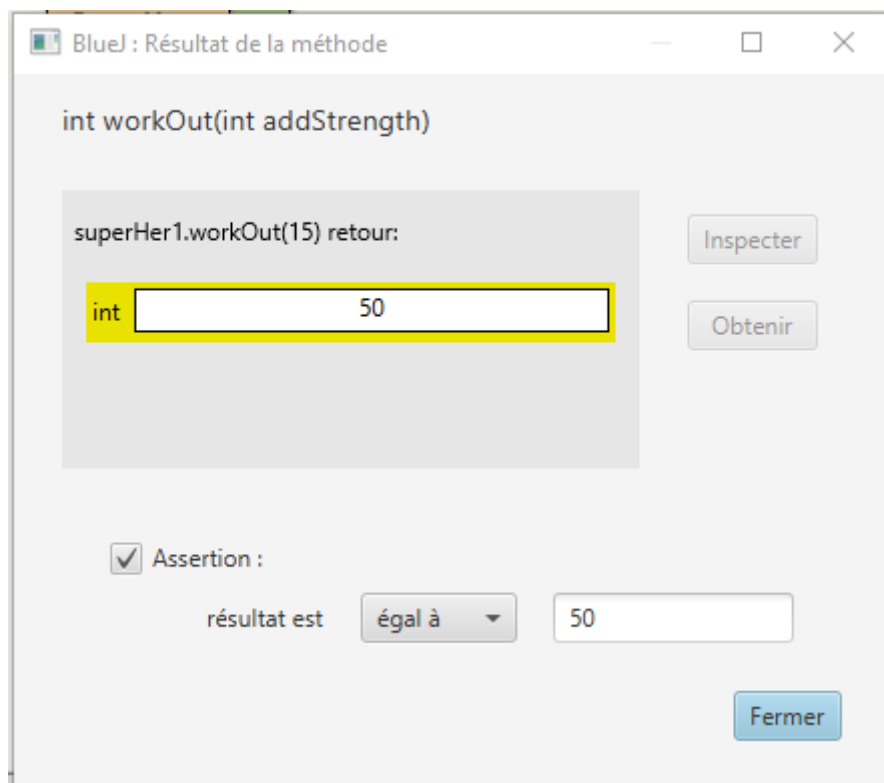


Après son entrainement, on imagine qu'il va gagner 15 de force :

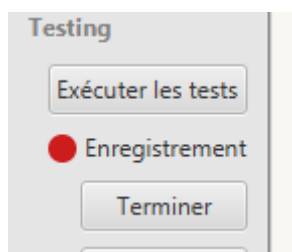


De base, un nouveau super-héros a 35 de force, donc on devrait obtenir 50. On écrit 50 à coté de « égal à ».

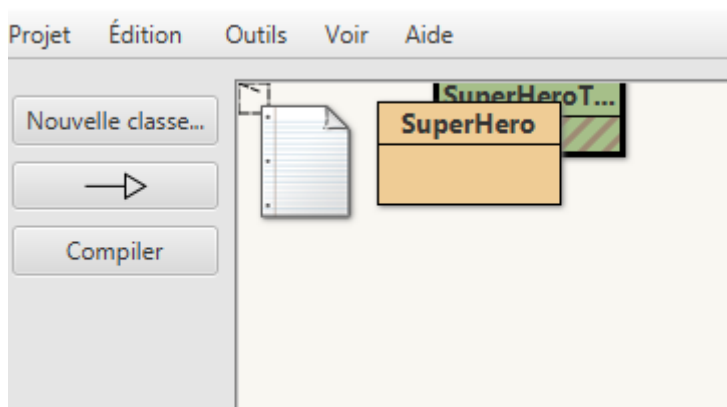
Encadré en jaune est le résultat de notre opération (ouf c'est le bon d'ailleurs)



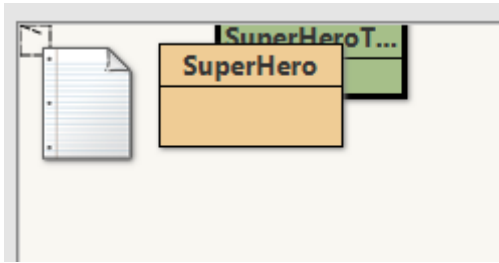
Notre test est terminé donc On clique sur **Terminer**



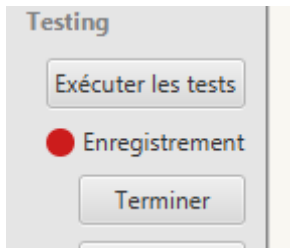
On n'oublie pas de compiler, sinon Compile-man ne va pas être content.



Ce qui nous enlève les rayures de notre joli carré vert.



Et on clique sur **Exécuter les tests** :



Et on obtient le résultat de notre test :

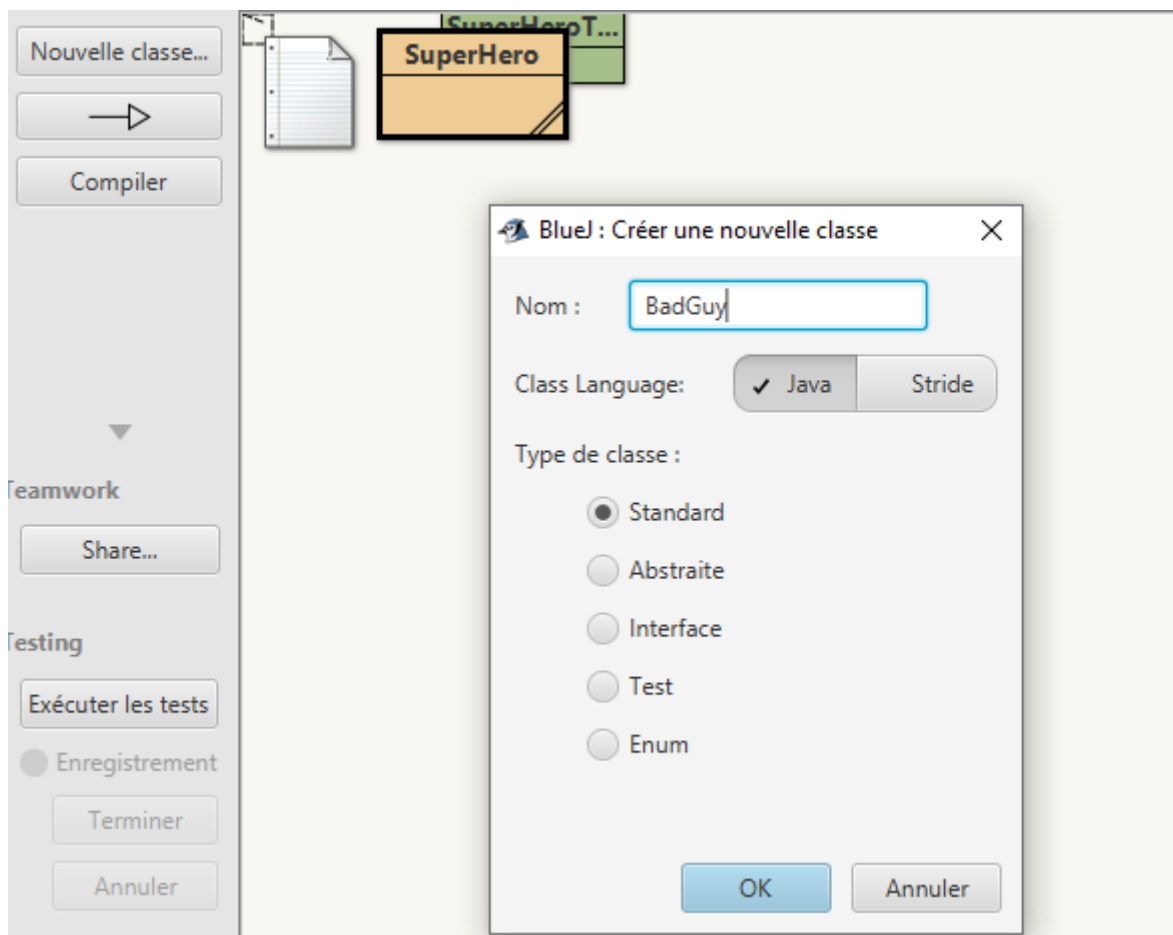


Yyyeeaaaaahhhh !! Première victoire de notre super-héros. C'est en grande partie grâce à toi !

Partie 2 – IntelliJ et JUnit

3. Liens entre deux classes

Mais passons aux choses sérieuses : un grand méchant arrive pour mettre des bâtons dans les roues de notre super-héros (Mouhahaha). Créons une classe BadGuy.



Le méchant, il n'est jamais très malin (donc on ne va pas lui mettre trop de champs). En général, c'est une brute donc il n'aura qu'un champ « *strenght* » (avec son accesseur `getStrength`)

```

/**
 * Décrivez votre classe BadGuy ici.
 *
 * @author (votre nom)
 * @version (un numéro de version ou une date)
 */
public class BadGuy
{
    // variables d'instance - remplacez l'exemple qui suit
    private int strength = 10;

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

    public int getStrength()
    {
        // Insérez votre code ici
        return this.strength;
    }
}

```

La ville n'est plus sûre depuis que notre méchant est arrivé. Notre super-héros va devoir lui apprendre les bonnes manières. Dans notre classe « *SuperHero* », nous allons ajouter un champ « *BadGuy* ». En effet, il vient sans vergogne dans la ville de notre héros.

```

public class SuperHero
{
    // variables d'instance - remplacez l'exemple
    private String identity = "Spiderman";
    private int strength = 35;

    private BadGuy badGuy = new BadGuy();
}

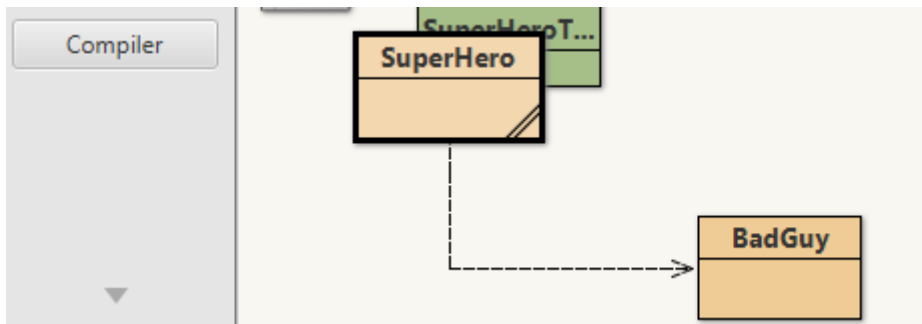
```

Nous venons d'associer un méchant à notre héros. Maintenant, place à la bagarre. Notre super-héros va essayer de se débarrasser de notre super-méchant. Nous allons créer une méthode « *fightBadGuy* » dans notre classe « *SuperHero* » pour qu'il combatte son méchant.

La méthode compare la force des deux opposants, et renvoie « *True* » si notre super-héros a une force supérieure à son méchant (et il lui botte allégrement les fesses). Sinon, on renvoie false.

```
/**
 * Return true if superhero beat the badGuy
 * else false
 */
public boolean fightBadGuy()
{
    return this.strength >= badGuy.getStrength();
}
```

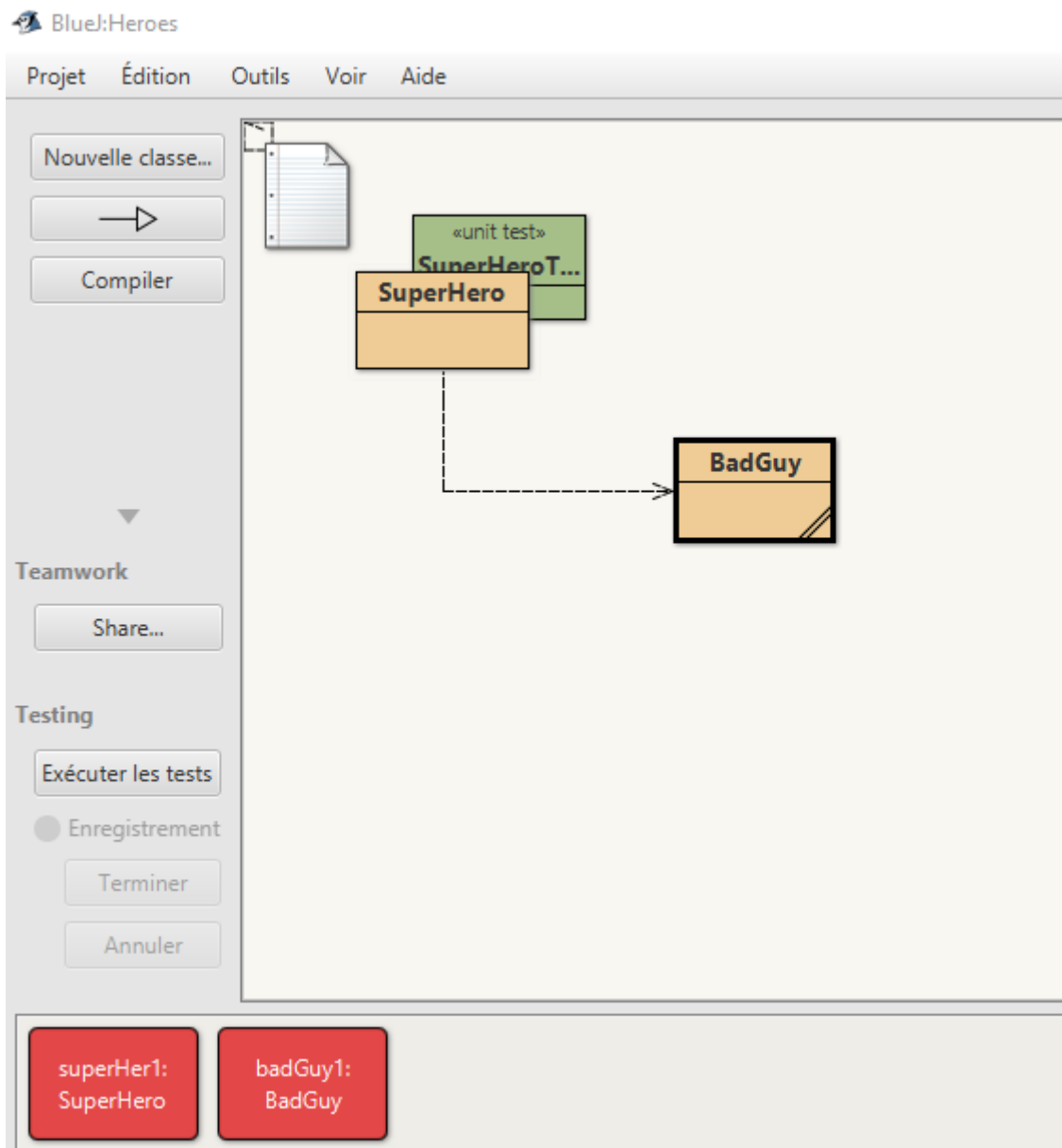
Compile-man vous demande de compiler. Exécution !



On avance bien. Essayons de sauvegarder une instance de « *Superhero* » avec une instance de « *BadGuy* » dans notre fichier de test.

Tout d’abord, sur la classe « *SuperHero* », **Clic Droit** sur la classe SuperHero > **new SuperHero()**

Ensuite, sur la classe BadGuy, **Clic Droit** sur la classe BadGuy > **new BadGuy()**



Comment faire pour dire à notre super-héros : « Tiens, ça c'est ton méchant, il est là pour t'embêter ? »

Nous sommes obligés de créer un « setter » dans notre classe « *SuperHero* » pour lui dire qui est son méchant. On supprime l'initialisation ci-dessous :

```
public class SuperHero
{
    // variables d'instance - remplacez l'exemple
    private String identity = "Spiderman";
    private int strength = 35;

    private BadGuy badGuy = new BadGuy();
}
```

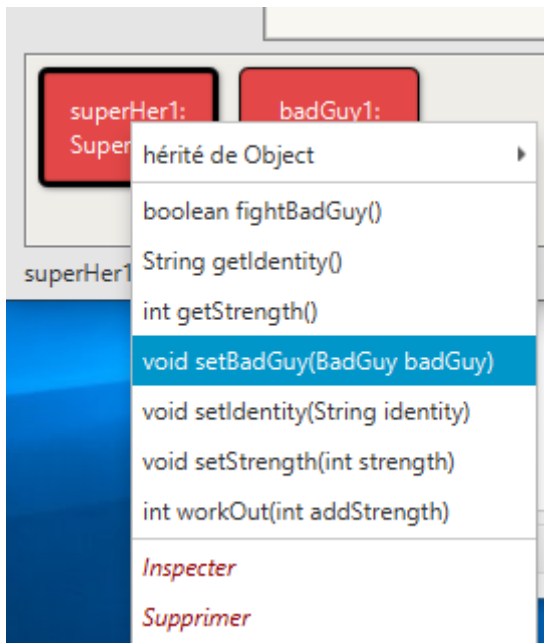
Par

```
private BadGuy badGuy;
```

Et on ajoute le setter « *setBadGuy* » pour la classe « *SuperHero* »

```
public void setBadGuy(BadGuy badGuy) {  
    this.badGuy = badGuy;  
}
```

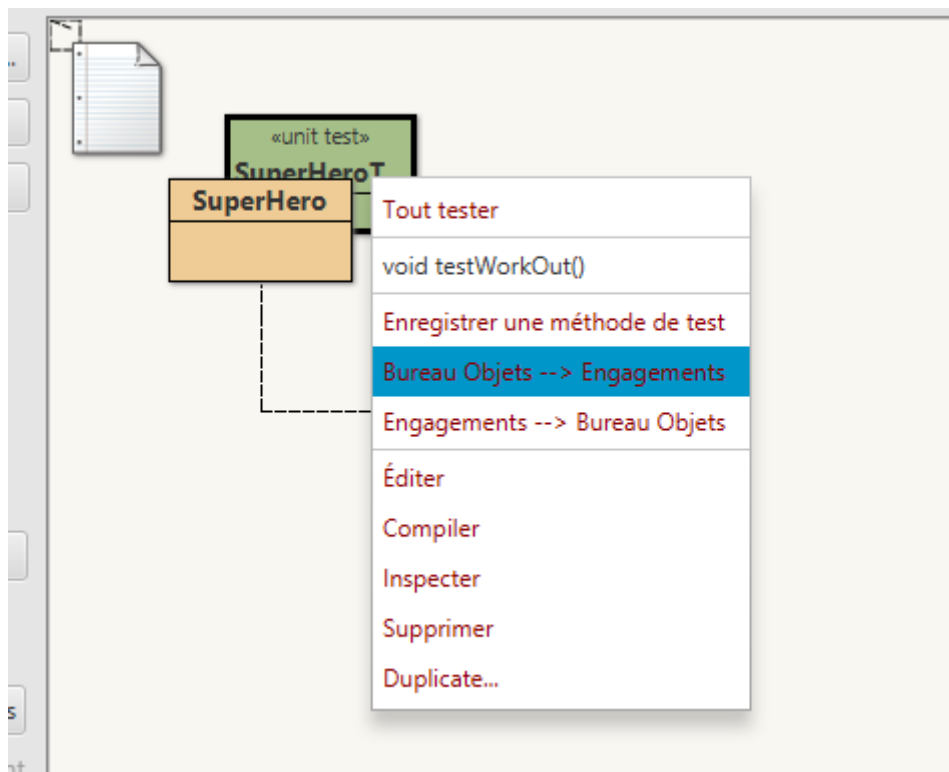
Maintenant, nous pouvons dire à notre SuperHero « *superHer1* » que « *badGuy1* » est son méchant via la méthode « *setBadGuy* ».



On précise en paramètre le nom de la variable du méchant (ici *badGuy1*)



Et, pour insérer ces deux instances dans le « *setUp* » de notre fichier test, nous faisons **ClicDroit** > « **Bureau Objets** → **Engagements** »



Tadam, Test-Man a encore frappé. Allons voir le code de « *SuperHeroTest* » en double cliquant dessus.

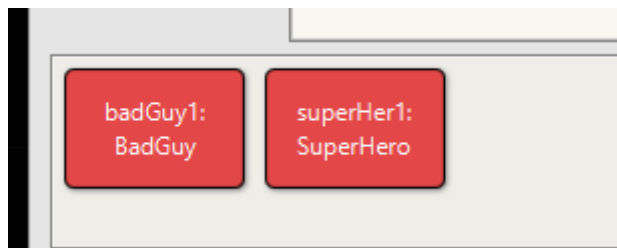
```
public class SuperHeroTest
{
    private SuperHero superHer1;
    private BadGuy badGuy1;
```

Deux champs sont apparus. Notre super-héros et son méchant. Et ci-dessous on a leur initialisation dans la méthode « *setUp* »

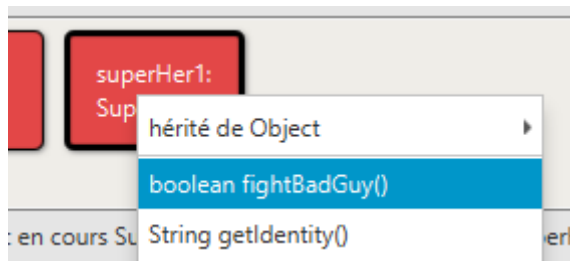
```
*/
@Before
public void setUp() // throws java.lang.Exception
{
    superHer1 = new SuperHero();
    badGuy1 = new BadGuy();
    superHer1.setBadGuy(badGuy1);
}
```

Enfin, la bataille finale !!! Pour tout combat, il y a des règles. Vérifions que les nôtres soient bien respectées. Nous allons tester la méthode « *fightBadGuy* ».

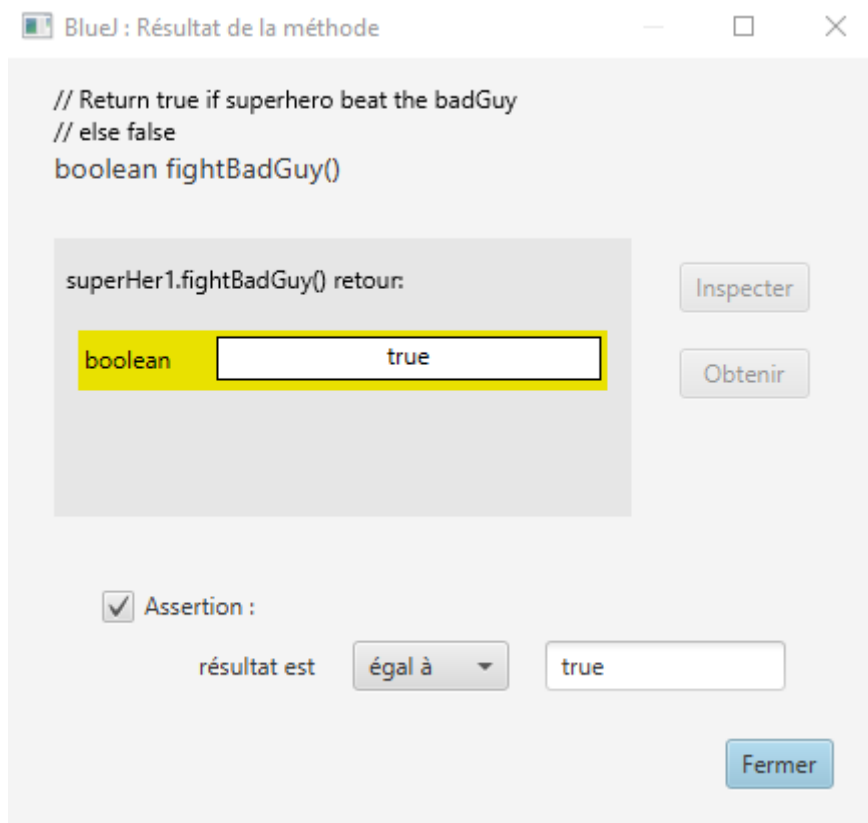
Pour cela, sur notre classe « *SuperHeroTest* », **Clic droit** > « **Enregistrer une méthode de test** ».



Comme par magie, nos deux objets sont réapparus. Testons la méthode « *fightBadGuy* »

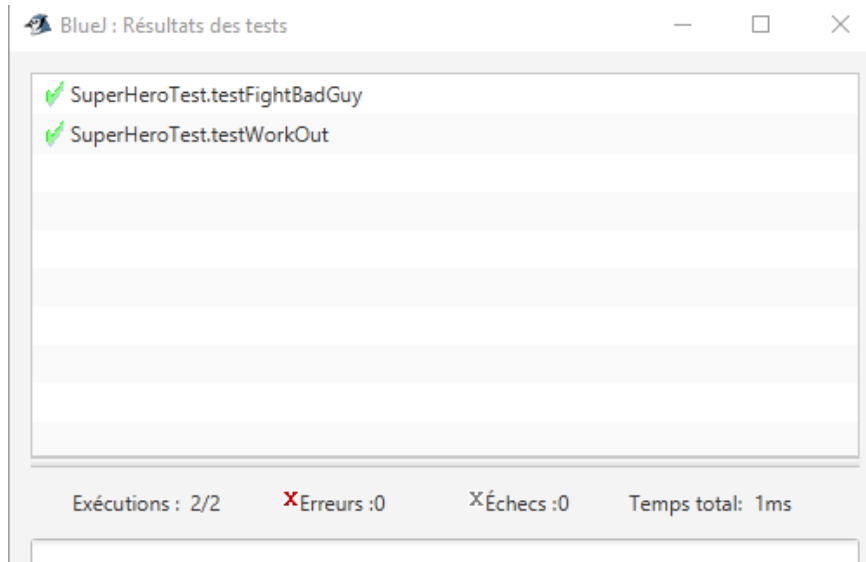


Notre super-héros a une force de 35 et le méchant une force de 10 donc on s'attend à ce que le résultat soit « *True* ».



On fait ce que Compile-man sait faire le mieux : on compile.

Et on fait ce que Test-man fait le mieux : On teste en exécutant les tests (tout simplement)



Bravo ! Grâce à toi la ville est enfin débarrassée du grand méchant ! Et notre super-héros peut continuer à veiller sur nos vies et nos espoirs.

4. Implémentation d'un attribut de relation 0..1 - * - La bi-directionnalité

Nous avons fini de travailler avec **BlueJ**, maintenant place aux outils de ceux qui jouent dans la cours des grands ; nous allons travailler avec **IntelliJ**.

Donc, où est ce qu'on s'était arrêtés ? Ah oui. J'ai remarqué quelque chose : on a un superhéros contre un seul méchant ? Cela ne tient pas trop la route. Notre super-héros est très fort et il peut se permettre de dérouiller plusieurs méchants.

Nous allons donc transformer l'attribut de type « *BadGuy* » dans « *SuperHero* » en `Collection<BadGuy>`

```
private Collection<BadGuy> badGuys;

/**
 * Constructeur d'objets de classe SuperHero
 */
public SuperHero()
{
    badGuys = new ArrayList<BadGuy>();
}
```

Sans oublier d'initialiser la collection dans le constructeur de « *SuperHero* » (sinon `NullPointerException` - Man se moquera de toi).

Voilà. Désormais on peut enfin combattre le crime comme il faut. Mais ... KESAKO ? Il y a des erreurs de partout.

```
public void setBadGuy(BadGuy badGuy){
    this.badGuy = badGuy;
}
```

```
public boolean fightBadGuy()
{
    return this.strength >= badGuys.getStrength();
}
```

Il va falloir corriger tout cela pour que notre super-héros puisse se battre correctement. Nous devons modifier la méthode « *setBadGuy* » pour qu'elle ajoute le méchant.

```
public void setBadGuy(BadGuy badGuy){
    badGuys.add(badGuy);
}
```

Et nous allons modifier notre méthode « *fightBadGuy* » pour que notre héros combatte toute sa liste de méchants.

```
/**
 * The hero fight every bad guy. If he have been defeat one time, the me
 * If he beats all his badGuy
 */
public boolean fightBadGuy()
{
    boolean win = true;
    for(BadGuy badguy : badGuys)
    {
        if(badguy.getStrength() > this.getStrength())
        {
            win = false;
            break;
        }
    }
    return win;
}
```

Ah ouf ! Il n'y a plus d'erreurs. Mais nous sommes confrontés à un problème ... tu ne vois pas de quoi je parle ? Aller tu me fais marcher ! Nous n'avons pas de relation bidirectionnelle. Le héros connaît ses méchants mais le méchant ne connaît pas le héros contre qui il se bat. Rectifions cela !

```

private SuperHero nemesis;
/**
 * Constructeur d'objets de classe BadGuy
 */
public BadGuy()
{
    // initialisation des variables d'instance
    nemesis = new SuperHero();
}

```

Il faut pouvoir affecter le bon super-héros au méchant. Créons les méthodes d'accès dans la classe « *BadGuy* ».

```

public void setSuperHero(SuperHero nemesis)
{
    this.nemesis = nemesis;
}
public SuperHero getSuperHero()
{
    return this.nemesis;
}

```

Et maintenant on écrit le test.

5. TDD

Les tests appelés “Test Driven Development” proviennent d’une technique de développement logiciel qui consiste à écrire des tests unitaires individuels sur chaque fonction. Le logiciel est ainsi développé de manière incrémentale et itérative. On va donc rédiger ces tests à l’aide JUnit. On fait usage de la méthode `assertEquals` qui permet de s’assurer que la valeur obtenue correspond bien à la valeur attendue.

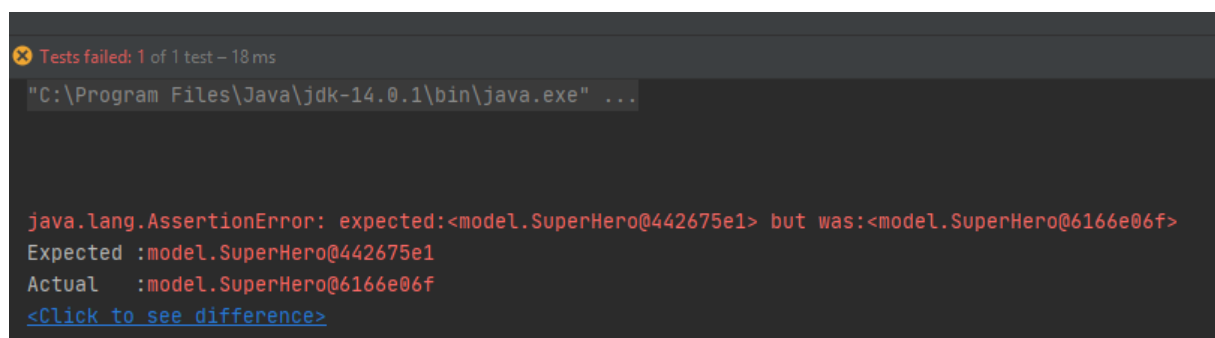
```
@BeforeEach
public void setUp()
{
    badGuy = new BadGuy();
    superhero = new SuperHero();
}

@Test
public void testGetStrength(){
    assertEquals( expected: 10, badGuy.getStrength());
}

@Test
public void testGetSuperHero()
{
    | assertEquals(superhero, badGuy.getSuperHero());
}

@Test
public void testSetSuperHero()
{
    badGuy.setSuperHero(superhero);
    assertEquals(superhero, badGuy.getSuperHero());
}
```

Et on lance les tests et Mais « Qué ça dire que ceci » ?



```
✖ Tests failed: 1 of 1 test - 18 ms
"C:\Program Files\Java\jdk-14.0.1\bin\java.exe" ...

java.lang.AssertionError: expected:<model.SuperHero@442675e1> but was:<model.SuperHero@6166e06f>
Expected :model.SuperHero@442675e1
Actual   :model.SuperHero@6166e06f
<Click to see difference>
```

« `testGetSuperHero` » a échoué : pourtant quand on crée un nouveau super héros il s’appelle « Spider-man » et possède 35 de force alors pourquoi ?

C’est parce qu’un objet créé, même s’il possède les mêmes valeurs dans ses attributs qu’un autre objet ne possède pas forcément la même adresse en mémoire. Donc si deux objets possèdent les mêmes

caractéristiques mais pas la même adresse mémoire, est ce que pour autant on doit les considérer comme différents ?

Nous allons donc utiliser et définir la méthode « *equals* » afin de rendre deux objets « *SuperHero* », possédant les mêmes données, égaux.

```
@Override
public boolean equals(Object o)
{
    /**
     * si l'objet passé en parametre est null ou n'est pas un objet SuperHero alors ils ne
     * peuvent pas être égaux
     */
    if(o == null || !(o instanceof SuperHero)) return false;
    /**
     * Si la référence (l'adresse mémoire est la même) ils sont forcément égaux
     */
    if(o == this) return true;
    /**
     * Sinon on transforme o en objet SuperHero et on compare les champs identity et strength
     */
    SuperHero superhero = (SuperHero)o;
    return superhero.getStrength() == this.strength && superhero.getIdentity() == this.identity;
}
```

On peut relancer notre test et C'est un succès !!!

Je vous laisse ajouter la méthode « *getSuperHero* » et son test 😊

6. Refactoring

Le refactoring est utilisé lorsque nous modifions la structure d'un objet, du programme et qui va impliquer de devoir modifier la logique visuelle et de compréhension du code. Nous allons chercher à adapter le code que nous avons écrit aux modifications qui ont été faites. Et je vais te parler de deux méthodes aujourd'hui : le **renommage** et l'**extraction**.

1. Rename Method

Cette méthode consiste à renommer les variables, attributs ou méthodes pour les rendre cohérents avec leur fonction. Par exemple, avant d'implémenter la Collection de « *BadGuy* », nous avons ceci :

```
public class SuperHero
{
    // variables d'instance - remplacez l'exemple
    private String identity = "Spiderman";
    private int strength = 35;

    private BadGuy badGuy = new BadGuy();
}
```

Et, subtilement, nous l'avons modifié de la manière suivante :

```
private Collection<BadGuy> badGuys;
```

En effet, « *badGuy* » au singulier est devenu « *badGuys* ». Cet exemple est subtil, mais nous allons faire une modification qui n'a pas été encore faite : la méthode « *fightBadGuy* ». En effet, son nom implique que notre héros va se battre contre seulement un seul méchant. Or notre méthode le fait combattre tous les méchants. On va devoir la renommer :

```
public boolean fightAllBadGuys()
```

Comme ça, toute personne qui verra ton code va tout comprendre, d'un seul coup.

2. Extract Method

La méthode d'extraction permet d'atomiser le code suivant la responsabilité. Imaginons que nous ajoutons ce code dans la méthode « *workOut* » de *SuperHero* :

```
public int workOut(int addStrength)
{
    this.strength += addStrength;
    //Print details about the hero after the work out
    log.info( msg: "name : "+identity);
    log.info( msg: "force : " + strength);
    return this.strength;
}
```

L'affichage des détails est effectué grâce à un champ de type **Logger** nommé « *log* » dans la classe *SuperHero*.

L'affichage des détails n'est pas vraiment lié à la méthode « *workOut* » on va donc l'extraire et en faire une méthode :

```
public int workOut(int addStrength)
{
    this.strength += addStrength;
    printDetailsHero();
    return this.strength;
}

private void printDetailsHero()
{
    //Print details about the hero after the work out
    log.info( msg: "name : "+identity);
    log.info( msg: "force : " + strength);
}
```

Voilà, nous avons séparé la responsabilité et cela permet de réutiliser l'affichage des détails d'un héros dans le journal de log.

Tu as vraiment beaucoup appris aujourd'hui ! Tu as pu créer ta propre BD à toi avec tes propres héros et méchants. Et sans t'en rendre compte, tu as appris à coder correctement en Java.

Et maintenant je vais te poser une question. Pourquoi on a commencé à faire des tests tout à l'heure avec JUnit ? Ce n'est pas plus simple de vérifier si nos méthodes fonctionnent en faisant des petits tests nous-même rapidement ? Beaucoup de programmeurs, (comme toi désormais), pensent que c'est une perte de temps. Ils veulent créer le meilleur jeu le plus vite possible. Et du coup quand ils doivent vérifier si quelque chose marche, ils vont juste créer un super-héros, l'entraîner un peu, des fois le faire combattre et ainsi de suite...

Mais à chaque ligne de code que tu rajoutes, il y a un risque que ton code ne marche plus. C'est pour ça, qu'en fait, il est plus rapide au final de rédiger des petites fonctions de test, « d'automatiser » nos tests, comme nous l'avons fait, pour nous assurer que chaque méthode, chaque champ est vérifié. Une seule ligne de code peut tout faire planter !

Alors la prochaine fois que tu diras ça « Oh mais ce n'est qu'une petite méthode, je suis sûr qu'elle va marcher, pas besoin de tester », rappelle-toi cela : J'ai rencontré un jour Splinter, un grand personnage qui pour vocation d'entraîner des tortues à devenir des ninjas... mais aussi, à coder très, très bien pendant son temps libre. Et c'est lui qui m'a expliqué ce que je viens de te raconter : « Prend toujours le temps de faire des tests, sinon tu te retrouveras noyé par des ' Pointers Exception ' ».

```
D:\testAgile>javac -cp junit-4.13.jar;. BadGuy.java SuperHero.java testBadGuy.java testSuperHero.java testJUNIT.java
D:\testAgile>java -cp junit-4.13.jar;hamcrest-core-1.3.jar;. org.junit.runner.JUnitCore testSuperHero testJUNIT testBadGuy
JUnit version 4.13
.avr. 29, 2020 9:07:21 PM testSuperHero setUp
INFO: Before statement
avr. 29, 2020 9:07:21 PM testSuperHero testFightBadGuy
INFO: Test FightBadGuy
.avr. 29, 2020 9:07:21 PM testSuperHero setUp
INFO: Before statement
.avr. 29, 2020 9:07:21 PM testSuperHero setUp
INFO: Before statement
.avr. 29, 2020 9:07:21 PM testSuperHero setUp
INFO: Before statement
.avr. 29, 2020 9:07:21 PM testSuperHero setUp
INFO: Before statement
avr. 29, 2020 9:07:21 PM testSuperHero testWorkOut
INFO: Test WorkOut Start
avr. 29, 2020 9:07:21 PM SuperHero printDetailsHero
INFO: name : Spiderman
avr. 29, 2020 9:07:21 PM SuperHero printDetailsHero
INFO: force : 50
.avr. 29, 2020 9:07:21 PM testSuperHero setUp
INFO: Before statement
.avr. 29, 2020 9:07:21 PM testSuperHero setUp
INFO: Before statement
.....
Time: 0,046
OK (12 tests)
```

Pour finir, je vais te citer un adage : « *Si tout semble bien marcher, vous avez forcément négligé quelque chose* ». C'est une variante de la **loi de Murphy**.

En effet, si vos tests sont valides, est-ce que cela signifie pour autant qu'ils contrôlent tous les aspects de votre code ? ou bien qu'une seule partie ?

Parfois les tests de méthodes complexes cachent des fonctionnalités non testées mais qui fonctionnent à un instant t. Cela peut être le constructeur d'une classe qui oublie d'instancier un de ces champs de type Collection par exemple, et vos tests valident cet oubli quand soudain. POUF, un NullPointerException sauvage apparait dans les hautes herbes.

7. BDD

Nous avons réalisé précédemment des TDD, Test Driven Development. De plus en plus en vogue, la méthode agile a permis l'émergence d'une nouvelle méthodologie de tests : les Behavior Driven Development. Les tests sont centrés autour d'une « User Story ». Plus simplement, une « User Story » est un cas d'utilisation de notre programme.

Ainsi dans le monde des super héros et des méchants, nous allons essayer d'imaginer des situations auxquels nos héros pourraient être confrontés. Pour cela, nous allons travailler avec un nouvel outil appelé « Cucumber ». C'est un monsieur qui aime beaucoup les concombres qui a inventé cet outil. J'espère que tu ne pas cru !

Je vais arrêter de te raconter des salades et je te propose donc de commencer. Voici les user-story que j'ai pu te concocter :

User-Story 1 : Combat entre les super héros et les méchants

Lorsque notre super héros réussit à vaincre le méchant, il absorbe la moitié de la force du méchant. Le méchant est affaibli et perd la moitié de sa force.

Dans les histoires pour enfants, les super héros sont toujours victorieux. Dans la vraie vie, ce n'est pas aussi simple. Des fois, les super héros ne sont pas au top de leur forme et perdent contre leur nemesis.

Quand cela arrive, le super héros perd la moitié de sa force et le méchant lui se retrouve revigoré. Il gagne la moitié de la force du super héros !

User-Story 2 : Entraînement du super-héros

Pour que notre super-héros devienne plus fort et réussis à écraser tous les méchants, il doit bien s'entraîner.

Dans cette user-story, le super héros gagne 15 de force lorsqu'il s'entraîne. Mais un super héros n'est pas invisible, sa force ne peut pas monter au-delà de 250...

User-Story 3 : Destruction du BadGuy

Notre super héros se bat contre des méchants sans cesse et malgré ça, ils continuent à terroriser la ville ! Notre super héros en a ras-le-bol. Dans cette User-Story, dès qu'un méchant est vaincu, il disparaît et quitte la ville pour toujours !

User-Story 4 : Les Avengers

Quand un méchant est très fort, il faut s'unir pour vaincre ! Et nos super héros ont compris cela. Ainsi on peut désormais créer des équipes formées des meilleurs super héros pour vaincre ce grand méchant. L'union fait la force : la force de tous les super-héros est unis en une grosse boule de force qui est envoyé sur le grand méchant.

Toutes ces histoires ont l'air si intéressantes mais on ne va coder que les 2 premières pour s'entraîner, je te laisserai faire les 2 dernières ;))

Feature

En BDD, on peut décrire dans un langage assez simple, ce que l'on souhaite faire. Ainsi toutes nos belles « user-story » vont être décrites dans le fichier « Feature » suivant :

```
Feature: Superhero fight one bad guy

Scenario Outline: fight a bad guy and the bad guy wins
  Given a fight between a superhero of strength <hero_strength> and a BadGuy of strength <badguy_strength>
  When the bad guy wins against the super hero
  Then the superhero strength decrease to <hero_newstrength>
  And the bad guy strength grow up to <badguy_newstrength>

Examples:
  | hero_strength | badguy_strength | hero_newstrength | badguy_newstrength |
  | 50 | 60 | 25 | 85 |
  | 250 | 500 | 125 | 625 |

Scenario Outline: fight a bad guys and the super hero wins
  Given a fight between a superhero of strength <hero_strength> and a Bad Guy of strength <badguy_strength>
  When the super hero wins against the bad guy
  Then the SuperHero strength grows up to <hero_newstrength>
  And the bad guy strength decrease to <badguy_newstrength>

Examples:
  | hero_strength | badguy_strength | hero_newstrength | badguy_newstrength |
  | 200 | 50 | 225 | 25 |
  | 250 | 100 | 250 | 50 |
```

Voici un exemple des deux premiers scénarios. On décrit ce qui se passe dans ce scénario dans « Scénario Outline » et on donne des exemples dans « Examples » pour notre programme vérifie que tout fonctionne bien. Le dossier « Feature » ne fait pas tout, on doit coder les fonctions dans un autre fichier Java. On s'assure d'avoir bien codé les instance « Given », « When » et « Then ».

Dans « Given », on décrit ce que l'on va tester.

Dans « When », on dit dans quel cas notre user-story se déclenche.

Dans « Then », on explique ce qui se passe quand l'user-story est en route, et on vérifie que nos résultats correspondent bien à ce à quoi l'on s'attendait. (voir « Examples »).

```

public void aFightBetweenASuperheroOfStrengthP1AndABadGuyOfStrengthP2(int p1, int p2) {
    System.out.println(p1);
    superHero = new SuperHero( identity: "Batman", p1);
    badguy = new BadGuy(p2);
}

@When("^the bad guy wins against the super hero$")
public void theBadGuyWinsAgainstTheSuperHero() {
    Assert.assertFalse(superHero.fightBadGuy(badguy));
}

@Then("^the superhero strength decrease to (\\d+)$")
public void theSuperheroStrengthDecreaseToP3(int p3) {
    assertEquals(superHero.getStrength(), p3);
}

@Then("^the bad guy strength grow up to (\\d+)$")
public void theBadGuyStrengthGrowUpToP4(int p4) {
    assertEquals(badguy.getStrength(), p4);
}

@Given("^a fight between a superhero of strength (\\d+) and a Bad Guy of strength (\\d+)$")
public void aFightBetweenASuperheroOfStrengthP1AndABGOfStrengthP2(int p1, int p2) {
    superHero = new SuperHero( identity: "SpiderMan", p1);
    badguy = new BadGuy(p2);
}

@When("^the super hero wins against the bad guy$")
public void theSuperHeroWinsAgainstTheBadGuy() { Assert.assertTrue(superHero.fightBadGuy(badguy)); }

@Then("^the SuperHero strength grows up to (\\d+)$")
public void theSuperheroStrengthGrowsUpToP3(int p3) {
    assertEquals(superHero.getStrength(), p3);
}

@Then("^the bad guy strength decrease to (\\d+)$")
public void theBadGuyStrengthDecreaseToP4(int p4) {
    assertEquals(badguy.getStrength(), p4);
}

```

On crée une autre classe, qui permet de tester toutes ces user-story.

```

package cumcumbertest;

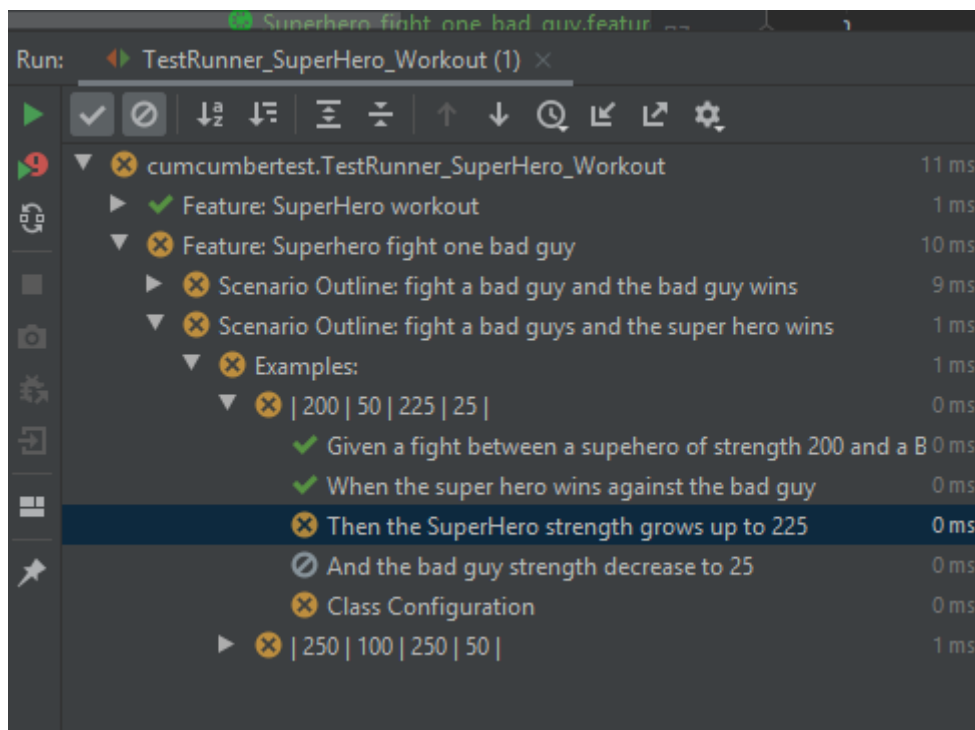
import cucumber.api.CucumberOptions;
import cucumber.api.junit.Cucumber;
import org.junit.runner.RunWith;

@RunWith(Cucumber.class)
@CucumberOptions(features="src/test/resources/features", glue="")
public class TestRunner_SuperHero_Workout {

}

```

Mais... ça n'a pas l'air de marcher très bien.



Pourquoi ... ah je sais ! Tu te rappelles la méthode qui permet de combattre des méchants ? On doit la corriger. Elle ressemble à ça pour l'instant.

```
public boolean fightBadGuy(BadGuy badguy)
{
    return badguy.getStrength() < this.strength;
}
```

Et pas seulement : la méthode setStrength doit s'assurer que les super héros ne peuvent pas devenir trop forts. Leur force ne doit jamais dépasser 250. On délègue donc à setStrength la vérification de la force du super héros.

```
public void setStrength(int strength)
{
    if(strength > 250)
        this.strength = 250;
    else
        this.strength = strength;
}
```

Et grâce à cette modification, on peut désormais baisser la force des héros qui ont perdu, et gagner en puissance lorsqu'un héros a gagné.

```

public boolean fightBadGuy(BadGuy badguy)
{
    if(badguy.getStrength() < this.strength)
    {
        setStrength(strength+badguy.getStrength()/2);
        badguy.setStrength(badguy.getStrength()/2);
        return true;
    }
    setStrength(this.strength /= 2);
    badguy.setStrength(badguy.getStrength() + this.strength);
    return false;
}

```

Et que voit-on ? Tout fonctionne !

▼ ✓ cumcumbertest.TestRunner_SuperHero_Workout	4 ms
▶ ✓ Feature: SuperHero workout	3 ms
▼ ✓ Feature: Superhero fight one bad guy	1 ms
▼ ✓ Scenario Outline: fight a bad guy and the bad guy wins	0 ms
▶ ✓ Examples:	0 ms
▼ ✓ Scenario Outline: fight a bad guys and the super hero wins	1 ms
▶ ✓ Examples:	1 ms

Partie 3 – Fusion de deux projets

Objectif

Alors, prêts pour cette nouvelle aventure ? On se languit d'avance de vous présenter ce qui fera de vous l'as de la conception d'un programme : LES DESIGN PATTERNS !!

Afin d'illustrer la modularité ainsi que l'efficacité des modèles de conception de logiciels, nous vous familiariserons aux patterns « Singleton » et « Décorateur ».

L'idée consiste à utiliser les points forts des patrons de conception afin de rendre la qualité du code meilleure.

Pour cela, nous nous sommes tout d'abord demandé : « Et si les NACs avaient la capacité de se battre » ? (« Oh non, Hamtaro va se blesser 😞 »).

Mais non voyons, nous voulons uniquement nous assurer qu'en l'absence de Laura, Hamtaro puisse se défendre du grand méchant Spiderman. Pour mêler l'utile à l'agréable, nous allons utiliser le pattern « Décorateur » pour modéliser la capacité des NACS et des grands méchants à se battre (et surtout se défendre !).

Pour ce faire, nous avons créé l'interface JAVA « IFeroce » avec la méthode « boolean fightBadGuy(BadGuy badguy) » en son sein.

Pour réduire la différence de poids entre le super (mauvais) héro et les NACS, nous avons fait évoluer ces derniers en super NAC ! (« C'est clair, trouver cette blague relève d'une intelligence supérieure »). Plus Hamtaro passera ses journées à dormir, plus il sera apte à repousser les méchants qui ne reviendront sans doute plus l'embêter !

Diagramme de classe

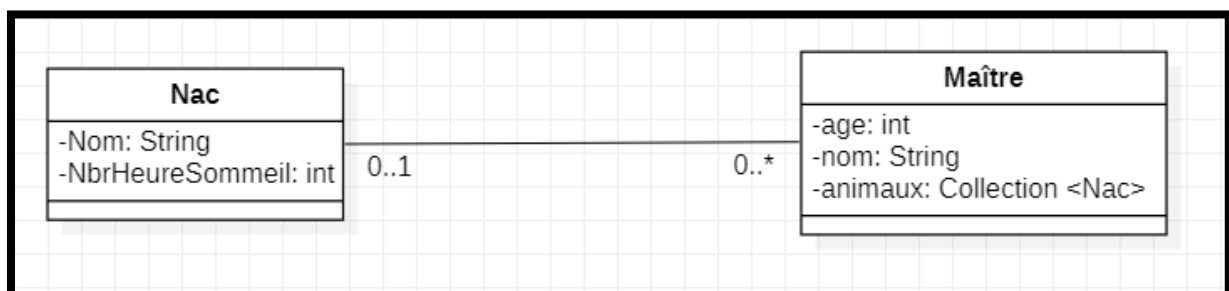


Diagramme de classe : 1 - Classe Nac avant fusion

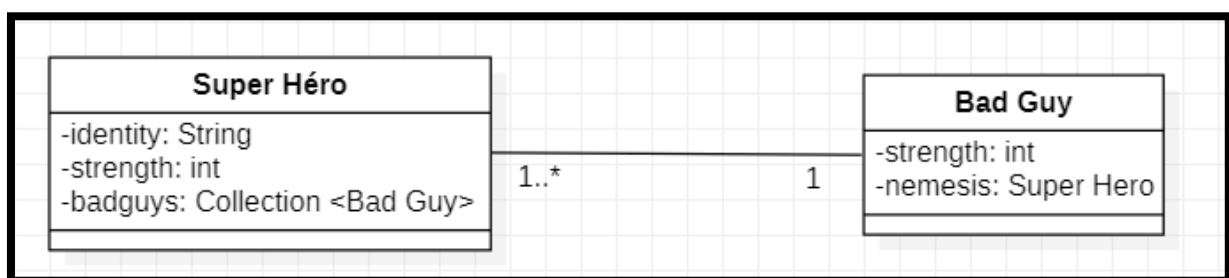
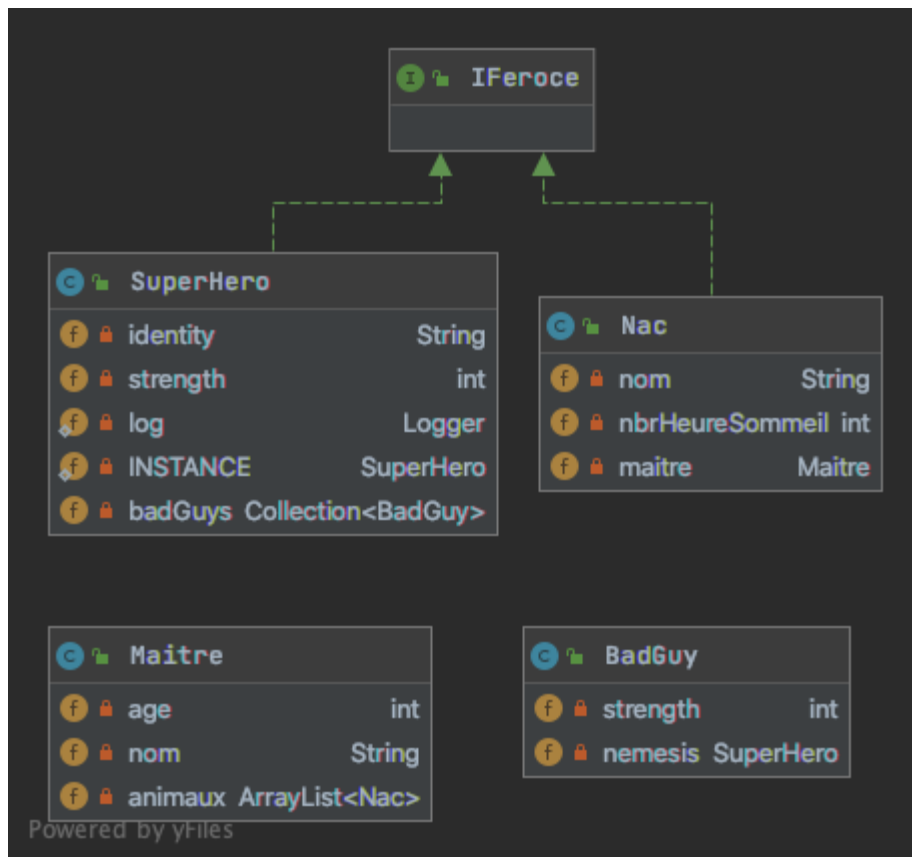


Diagramme de classe : 2 - Classe SuperHéro avant fusion



1 Diagramme de classe après fusion

Design Patterns

1) Design Pattern Singleton.

Les supers héros sont bien connus pour être (super) forts ! Pour cela, nous avons choisi qu'un unique Super Héros pourra veiller sur la ville d'Hamtaro. Pour ce faire, nous avons implémenté le design pattern « Singleton ».

Ce design pattern sert à restreindre l'instanciation d'une classe à un unique objet. En clair, un super Héro n'existera qu'une seule fois et sera le seul et l'unique super Héros de la ville.

Pour cela, quelques modifications sont à apporter à la classe Super héros initiale. Au lieu de donner des paramètres par défauts comme c'était le cas précédemment, on crée un attribut Super Héro **static**. Ce mot clé est essentiel pour gérer l'unicité de Spiderman.

```

public class SuperHero implements IFeroce {
    private String identity;
    private int strength;
    private static Logger log = Logger.getLogger("SuperHero");

    private static SuperHero INSTANCE;

    private Collection<BadGuy> badGuys;
  
```

Implémentation de l'interface IFeroce 1

Ce n'est pas tout, il faut pouvoir renvoyer ce bon super Héros sans en créer un autre qui empiéterait sur le territoire de notre Spiderman. Cela se fait par la création de la méthode suivante :

```
public static SuperHero getInstance(){
    if(INSTANCE == null){
        INSTANCE = new SuperHero( identity: "Spiderman", strength: 40);
    }
    return INSTANCE; // Lazy Loading
}
```

2 modifications de la classe SuperHéro

Ensuite, lorsque vous retentez les tests, une barre rouge devrait apparaître. C'est normal, pas de panique. Dès que nous créons un super Héros, il vaut mieux se référer à la méthode créée ci-dessus (et ce pour tous les tests) :

```
@Test
public void testEquals()
{
    assertTrue(!superHer1.equals(null));
    assertEquals(superHer1, superHer1);
    assertEquals(superHer1, SuperHero.getInstance());
    assertTrue(superHer1.getIdentity().equals("Spiderman"));
}
```

3 Modifications des classes de test

2) Design Pattern Décorateur

Pour ne pas surcharger le travail du Super Héro de la ville, les Nacs ont mutés pour développer des capacités au combat. En revanche, les Nacs naissent malheureusement sans force, donc sans pouvoir. Ils acquièrent de la force en dormant. Pour répondre à cette problématique, nous sommes passés par le Design Pattern Décorateur. En effet, sans ajouter de force à un Nac, le décorateur permet d'attacher dynamiquement une nouvelle responsabilité à notre objet qui peut donc désormais se battre.

Pour cela, il faut passer par l'implémentation de l'interface (caractéristique d'un comportement) suivante :

```
package model;

public interface IFeroce {
    boolean fightBadGuy(BadGuy badguy);
}
```

4 Interface IFéroce

Ensuite, comme les Super Héros et les Nacs possèdent désormais ces comportements, ils doivent tous les deux posséder la fonction fightBadGuy et être en relation avec IFéroce de la manière suivante :

```
public class Nac implements IFeroce{
```

```
public class SuperHero implements IFeroce {
```

Voici comment nous avons décidé de transformer les heures de sommeil en force :

```
@Override
public boolean fightBadGuy(BadGuy badguy) {
    double sommeilTransformeEnForce = (this.nbrHeureSommeil * 100) / Math.PI;
    if(badguy.getStrength() < sommeilTransformeEnForce)
    {
        setNbrDodo((int) (this.nbrHeureSommeil + badguy.getStrength()/2));
        badguy.setStrength(badguy.getStrength()/2);
        return true;
    }
    setNbrDodo(this.nbrHeureSommeil / 2);
    badguy.setStrength((int) (badguy.getStrength() + this.nbrHeureSommeil * 2));
    return false;
}
```

5 : Méthode FightBadGuy

Attention, on ne perd pas les bonnes habitudes, pas de méthodes nouvelles sans une phase de test !

Voici l'équivalent dans la classe NacTest :

```
@Test
public void fightBadGuyTest(){
    Nac hamtaro = new Nac( nom: "Hamtaro", nbrHeureSommeil: 12);
    BadGuy bg = new BadGuy( strength: 12);
    Nac hamtara = new Nac( nom: "Hamtara", nbrHeureSommeil: 0);

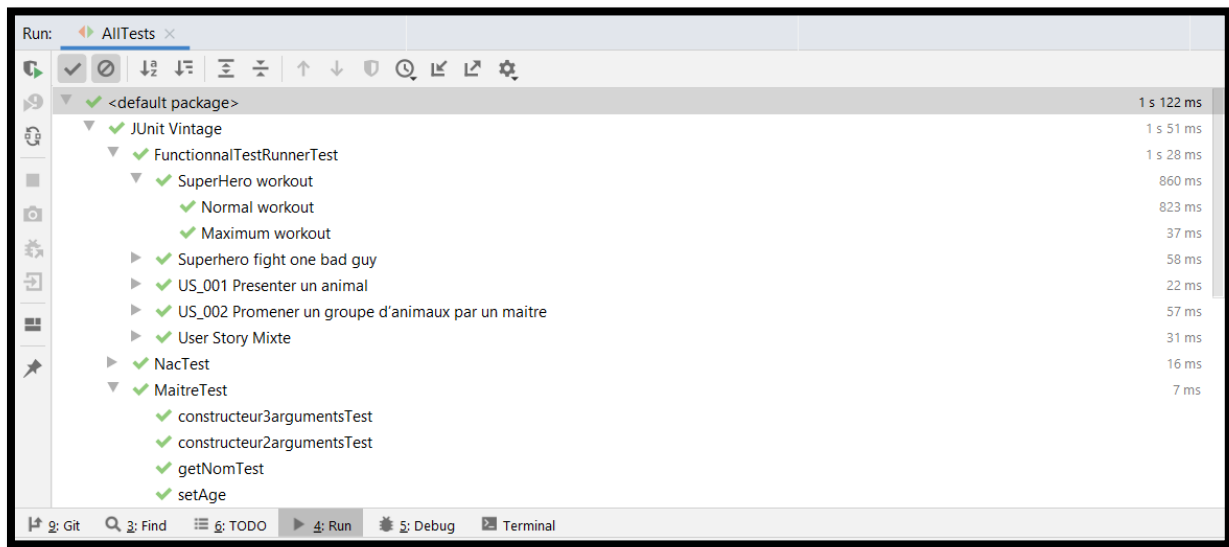
    assertTrue(hamtaro.fightBadGuy(bg));
    assertEquals(hamtaro.getNbrDodo(), actual: 18 );
    assertEquals(bg.getStrength(), actual: 6);
    assertFalse(hamtara.fightBadGuy(bg));
    assertEquals(hamtara.getNbrDodo(), actual: 0 );
    assertEquals(bg.getStrength(), actual: 6);
}
```

Les animaux peuvent désormais combattre les méchants !

100% classes, 98% lines covered in package 'model'			
Element	Class, %	Method, %	Line, %
BadGuy	100% (1/1)	100% (6/6)	100% (12/12)
Maitre	100% (1/1)	100% (15/15)	100% (62/62)
Nac	100% (1/1)	100% (13/13)	100% (46/46)
SuperHero	100% (1/1)	100% (13/13)	94% (35/37)

6 Couverture du code

Les Nouveaux Animaux de Compagnie sont désormais intégralement couverts par notre assurance multirisque enlèvement. Laura pourra enfin promener Hamtaro en toute sécurité !



7 Barre verte =)

Connaissant bien le vice des méchants, nous nous doutions bien qu'ils seraient prêts aux pires actes pour priver à un maître de voir son hamster à son retour le soir. Par conséquent, la meilleure assurance au niveau mondial pour la protection des NACs (et oui la nôtre) a simulé dernièrement les différents scénarios compromettant pour les NACs et leur verdict est sans appel : les grands méchants ne seront plus invincibles ... « dormez, dormez petits NACs, les méchants n'ont qu'à bien se tenir ! ».

Pour finir en beauté, que diriez-vous d'écrire vous-même la User Story « Apporter un moyen de défense aux NACs » .

« Bon c'est le moment de fermer ce tutoriel et de continuer mes recherches ailleurs, encore un qui veut se faire valoir ».

Pour ceux qui sont restés, voici la solution ! Je parie que des fidèles comme vous avez déjà tout vu, mais bon je tenais quand même à me faire valoir 😊 .

- **US MIXTE : Apporter un moyen de défense aux Nacs**

- *Critères d'acceptation :*

- As a : Player

- I want : Que mon Nac sache combattre

- So that : Pour se défendre contre les méchants

- *Description :*

- Given : Un combat entre un Nac et un ennemi

- When : Le Nac gagne

- Then : Le NAc voit son nombre d'heures de sommeil augmenter et le Bad Guy voit sa force diminuer

- Given : Un combat entre un Nac et un ennemi

- When : Le badguy gagne

- Then : Le NAc voit son nombre d'heures de sommeil diminuer et le Bad Guy voit sa force augmenter.

Pour ceux d'entre vous qui ont omis la seconde description (oui vous qui avez clairement pris partis pour les NACs), il faut envisager que la défaite du NAC est possible (sauf si vous trouvez le fameux cheat code permettant de rendre les NACs invincibles, bah oui, nous n'allions tout de même pas tout vous révéler).

Ces scénarios nous permettent d'établir le fichier feature dont le contenu vous est illustré ci-après :

```

Feature: User Story Mixte
  As a : Player
  I want : Que mon Nac sache combattre
  So that : Pour se défendre contre les méchants

Scenario Outline: Le nac gagne contre un badguy
  Given un Nac avec un <nbrHeureDodo> et un BadGuy avec une <force>
  When le Nac gagne
  Then le Nac voit son <evolutionNbrSommeil> augmenter et le badguy son <evolutionForce> diminuer

Examples:


| <nbrHeureDodo> | <force> | <evolutionNbrSommeil> | <evolutionForce> |
|----------------|---------|-----------------------|------------------|
| 12             | 12      | 1                     | 0                |



Scenario Outline: Le nac perd contre un badguy
  Given un Nac avec un <nbrHeureDodo> et un BadGuy avec une <force>
  When le badguy gagne
  Then le Nac voit son <evolutionNbrSommeil> diminuer et le badguy son <evolutionForce> augmenter

Examples:


| <nbrHeureDodo> | <force>    | <evolutionNbrSommeil> | <evolutionForce> |
|----------------|------------|-----------------------|------------------|
| 1              | 1000000000 | 0                     | 1                |


```

8 Tests fonctionnels mixtes

Naturellement, les tests fonctionnels nouvellement rédigés et déroulés devraient ressembler à ce qui suit. « il a employé le terme devrait, doute-t-il de ces facultés à inculquer des choses ? Ce qu'il nous explique est-il vrai »

```

public static class Us_Mixte {

    private Nac puppy;
    private BadGuy bg;
    private int force_before;
    private int nbrDodo_before;

    @Given("un Nac avec un {int} et un BadGuy avec une {int}")
    public void un_Nac_avec_un_et_un_BadGuy_avec_une(Integer int1, Integer int2) {
        puppy = new Nac( nom: "hamtaro", int1);
        bg = new BadGuy(int2);
        this.force_before = bg.getStrength();
        this.nbrDodo_before = puppy.getNbrDodo();
    }

    @When("le Nac gagne")
    public void le_Nac_gagne() { assertEquals(puppy.fightBadGuy(bg), actual: true); }

    @When("le badguy gagne")
    public void le_badguy_gagne() { assertEquals(puppy.fightBadGuy(bg), actual: false); }
}

```

```

@Then("le Nac voit son {int} augmenter et le badguy son {int} diminuer")
public void le_Nac Voit_son_augmenter_et_le_badguy_son_diminuer(Integer int1, Integer int2) {
    boolean b1 = false;
    if(int1 == 1){
        b1 = true;
    }
    assertEquals(b1, actual: puppy.getNbrDodo() > this.nbrDodo_before);
    if(int2 == 1){
        b1 = true;
    }
    else{
        b1 = false;
    }
    assertEquals(b1, actual: bg.getStrength() > this.force_before);
}

@Then("le Nac voit son {int} diminuer et le badguy son {int} augmenter")
public void le_Nac Voit_son_diminuer_et_le_badguy_son_augmenter(Integer int1, Integer int2) {
    boolean b1 = false;
    if(int1 == 1){
        b1 = true;
    }
    assertEquals(b1, actual: puppy.getNbrDodo() > this.nbrDodo_before);
    if(int2 == 1){
        b1 = true;
    }
    else{
        b1 = false;
    }
    assertEquals(b1, actual: bg.getStrength() >= this.force_before);
}

```

Arrêtez d'être rabat joie, vous possédez maintenant votre propre style d'écriture : façonner le et amusez-vous autant que vous le pourrez.

C'est le moment de faire apparaitre une dernière fois pour toute l'exécution ainsi que les tests (n'hésitez pas à verser toutes vos larmes, nous connaissons le manque que l'on procure aux gens).

Quelle expédition nous avons réalisé ! Vous rendez-vous compte ? Nous avons quand même eu l'audace de vouloir intégrer les NACs au sein du monde terrifiant des super-Héros (et des BadGuys en particulier). Après plusieurs péripéties (de votre part uniquement ça a été un jeu d'enfant de notre côté), nous avons réussi à mettre en relation deux programmes totalement indépendants les uns des autres à l'origine.

Cela n'aurait jamais été possible sans la méthodologie Agile (vous comprendrez après nous bien sûr) mise en place dès le départ. En effet, nous avons progressivement mis en place les fonctionnalités des programmes tout en nous assurant qu'une intégration ne porte pas préjudice au reste du programme. Par ailleurs, cela évite l'effet de tunnel puisqu'à tout moment, nous savons où nous en sommes !

C'est sur une larme (très) salée que nous refermons les pages de ce manifeste. Nous vous confions cet univers atypique qui aura besoin de vous pour perdurer !

Vous pourrez trouver cette solution sur le lien suivant :



FIN.

The End

Notre aventure a été très intéressante : nous avons pu voir l'émergence de grands super-héros. Des méchants ont croisé la route de nos super-héros qui ont dû donc s'entraîner très fort pour pouvoir les vaincre. Tu sais désormais utiliser BlueJ mais mieux encore, un logiciel de développement que les grands utilisent : IntelliJ. Tu as pu te rassurer que tes méthodes fonctionnent grâce aux tests unitaires (JUnit) mais aussi grâce aux tests fonctionnels (Cucumber). Toute cette aventure m'a fatigué, je pense que je vais me reposer un peu. Je te lègue les rennes de ce que nous avons créé.

L'avenir de ce monde est désormais entre tes mains. Que vas-tu en faire ?