# Agilité – Les tests unitaires

*HADI Ismaïl & JOLIMAN Iris*

*Classe Nac*

# Table des matières


Première partie : BlueJ 3

Installation de BlueJ 3

Création d'un nouveau projet 3

Création de la classe « fétiche » : NAC 6

Tu vas donner vie à un Hamster ! (c'est magique) 7

Hamtaro se présente-t-il vraiment bien ? 11

Encore des tests … toujours des tests 17

Seconde partie : Eclipse et jUnit 21

Création d'un projet avec IntelliJ 21

Oulala une avalanche de code ! 23

CLASSE MAÎTRE 24

CLASSE NAC 26

CLASSE MAÎTRE TEST 27

NAC TEST 30

Escalade de Tests 31

Refactoring 33

# Première partie : BlueJ

Q1. télécharger BlueJ

La programmation orientée objet est un outil très répandu aujourd'hui dans le développement logiciel. Elle permet de représenter des problématiques très variées à travers des **objets** et des **classes**.

Pour comprendre son fonctionnement, nous avons décidé de baser ce tutoriel pour illustrer la vie des NACs (Nouveaux Animaux de Compagnie).

L'outil BlueJ, que nous allons utiliser au sein de ce tutoriel, permettra d'allier la schématisation à l'implémentation technique.

## Installation de BlueJ

Q2. Installer BlueJ sur la machine

Comme indiqué précédemment, installer BlueJ est un prérequis pour ce tutoriel car car c'est l'outil qui nous permettra de visualiser et manipuler nos objets et nos classes.

Vous pourrez le télécharger et l'installer en cliquant sur le lien suivant : http://www.bluej.org/

## Création d'un nouveau projet

Q3. Créer un nouveau projet

En ouvrant BlueJ, il est nécessaire de créer un nouveau projet. Pour cela on réalise les actions suivantes : Projet > Nouveau projet…

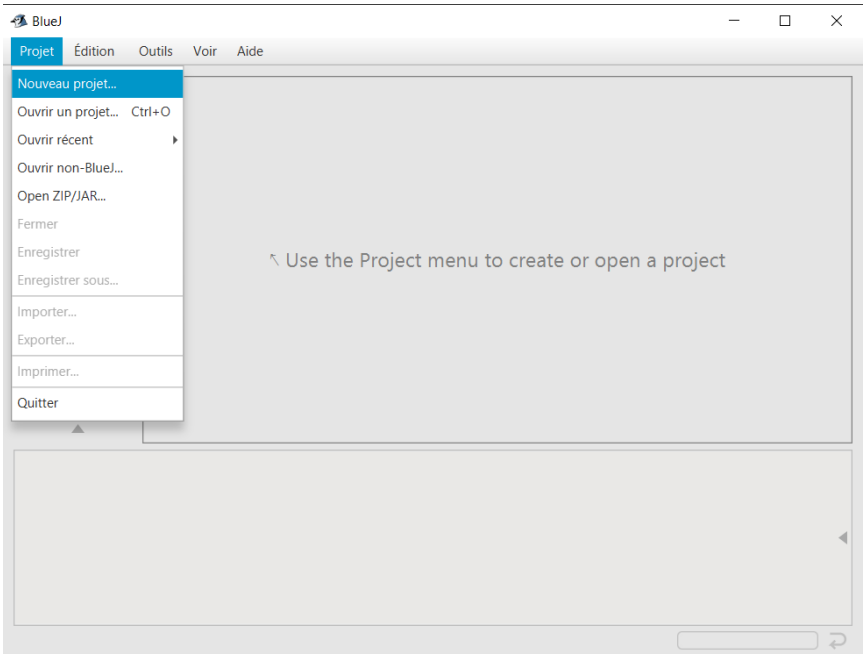



*Figure 1: Création d'un nouveau projet dans BlueJ*

Par la suite, vous pouvez choisir le nom de votre choix ainsi que le répertoire dans lequel vous voulez retrouver le projet.

Nous voulions garder le suspens pour l'instant concernant le thème de notre projet… Veuillez donc ne pas porter trop d'attention au nom choisi 😊.

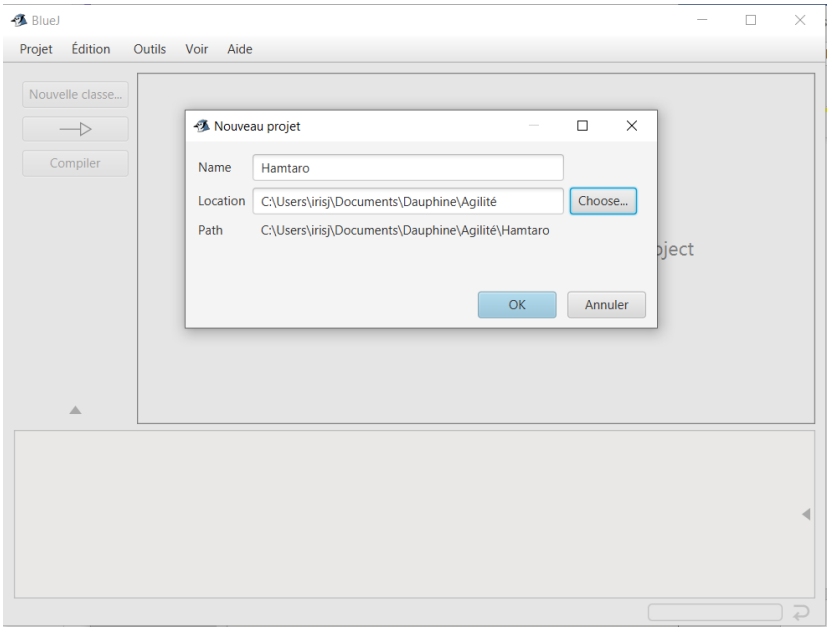



*Figure 2: Nommer le projet*

A l'issue de la création du projet, vous devriez obtenir une fenêtre similaire à celle ci-dessous avec le nom de votre projet en haut à gauche.

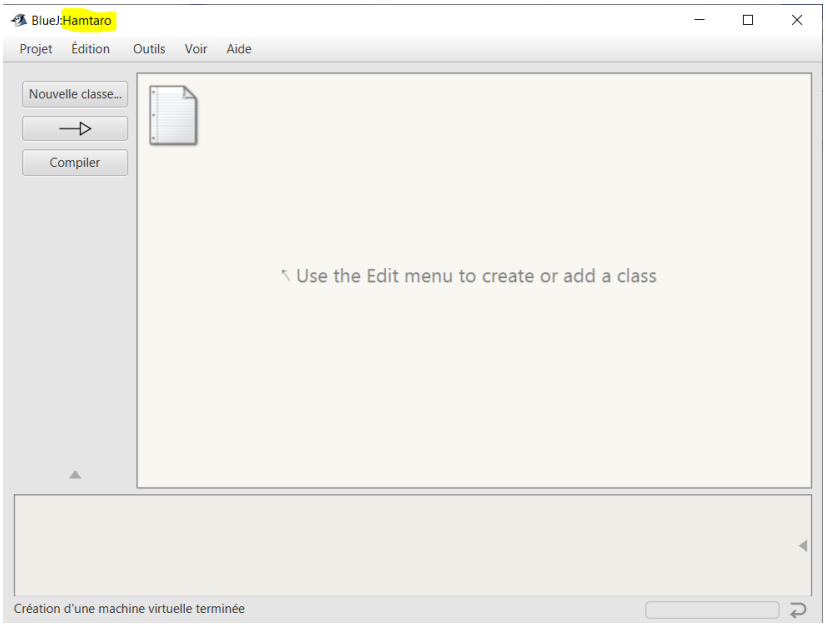



*Figure 3: Fenêtre après la création de projet*

Pour revenir à notre modélisation, quel est donc notre besoin ?

C'est l'heure de mettre fin au suspens qui vous tiraillait surement. Nous allons représenter la vie et les aventures d'un petit hamster très célèbre pour tous les enfants des années 2000 (ça commence à faire vieux…) : Hamtaro.

Pour ceux qui ne connaissent pas, voici pour la culture : https://www.youtube.com/watch?v=sWti4NjPTYo

Pour adopter cet animal de compagnie qui en fait craquer plus d'un, il faut pouvoir trouver un moyen de modéliser son espèce dans notre projet. En programmation orientée objet, cela se matérialise par la création d'une classe (description d'un type) que nous appellerons NAC.

Hamtaro sort tous les jours lorsque son maître est au travail ou à l'école pour retrouver tous ses amis Hamster dans le club des Ham-Hams où on retrouve Amiral, Bijou et Chapo par exemple. Tous ces petits hamsters y compris notre favori Hamtaro seront des instances de la classe NAC (donc des objets en programmation orientée objet). Nous pourrions également créer la pire angoisse d'Hamtaro Serpentar le serpent qui serait également une instance de la classe NAC.

## Création de la classe « fétiche » : NAC

Q4. Créer la classe fétiche

Pour créer notre nouvelle classe NAC, nous devons cliquer sur Nouvelle classe… et la fenêtre suivante s'affiche. Pour le moment, nous nous intéressons uniquement au type standard.

La compilation consiste à vérifier la cohérence du code présent au sein de la classe compilée. C'est au niveau de cette opération que les erreurs sont détectées.

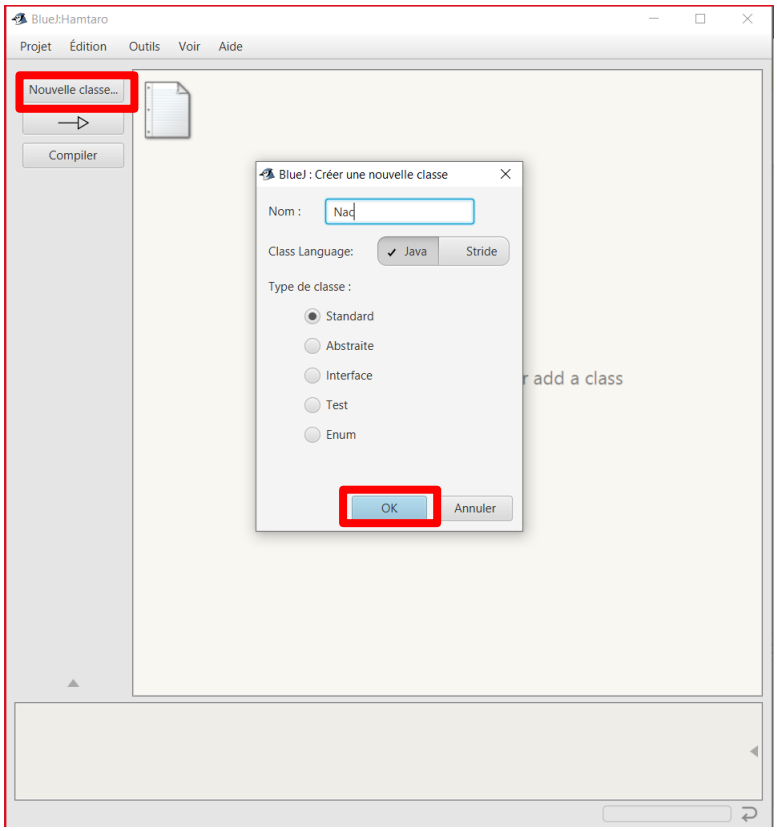



*Figure 4: Définition de notre nouvelle classe (type standard)*

Q5. Compiler la classe

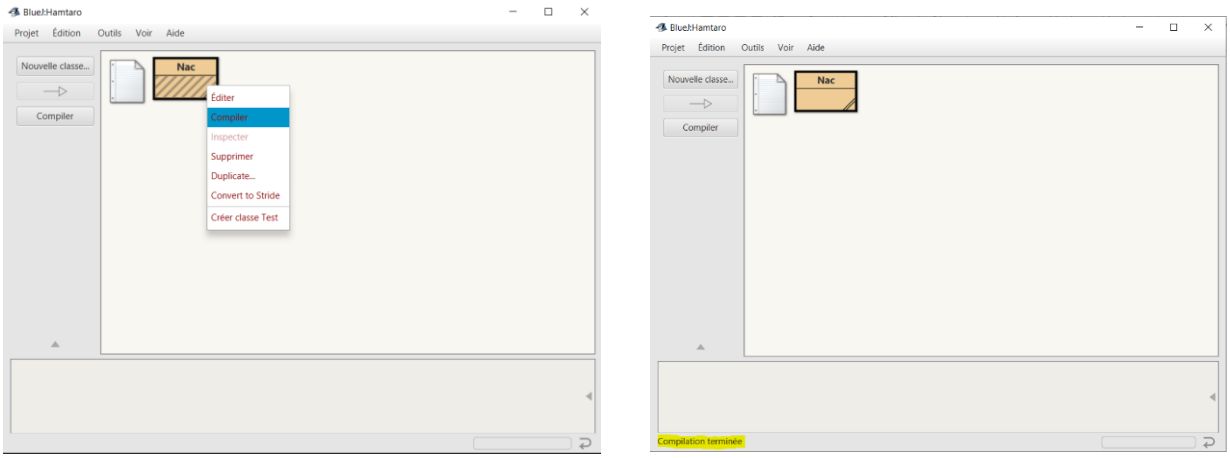



*Figure 5: Classe NAC avant et après compilation*

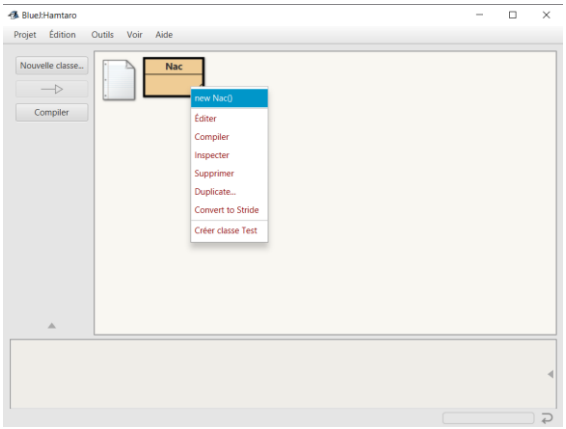
Avant compilation, on remarque que la classe est grisée. C'est cette modélisation sur BlueJ qui indique à l'utilisateur qu'une compilation est nécessaire pour pourvoir utiliser la classe (avant ça, on bloque la naissance de notre Hamtaro préféré … quel dommage).
Visuellement, vous pouvez constater que le quadrillage gris a disparu après la compilation. Voilà une bonne chose de faite.

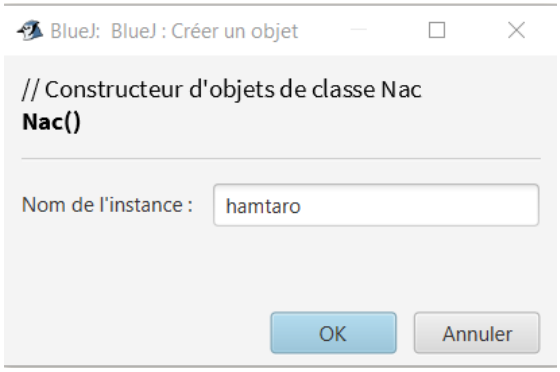
## Tu vas donner vie à un Hamster ! (c'est magique)

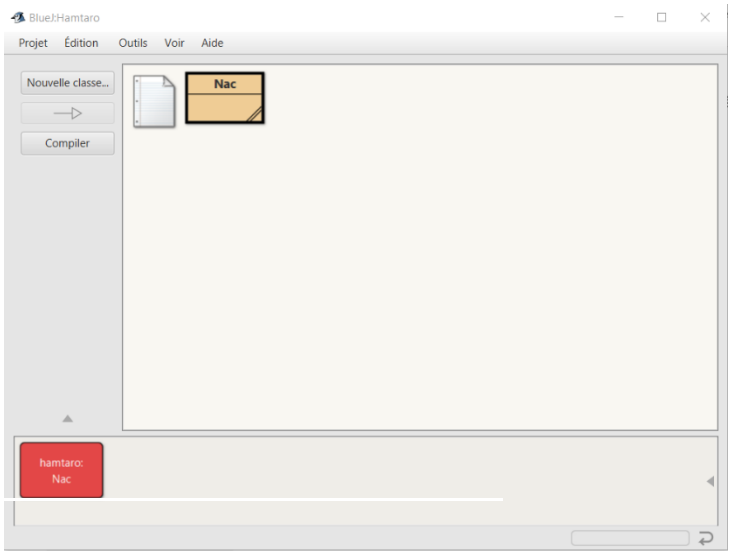
Q6. Instancier la classe

Maintenant que notre classe est prête à l'emploi, nous allons pouvoir créer autant d'animaux de compagnies (bon pas un zoo quand même) que voulus: commençons tout d'abord par notre hamster Hamtaro !

Pour cela, il suffit d'effectuer un clic droit sur la classe Nac (après compilation) et de sélectionner le champ New Nac(). Une pop-up apparait ensuite nous permettant de donner le nom souhaité à notre nouveau compagnon.
En tant que futur as de la programmation orientée objet, il est à noter que le nom d'une classe commence par une MAJUSCULE et celui d'une instance par une minuscule. Dans notre cas, hamtaro est une instance de la classe Nac (tant pis pour le français de toute façon on se rappelle plus de rien depuis le CP).




On peut voir ici qu'Hamtaro a bien été créé (dans la partie rouge)

Q7. Ajouter deux attributs avec accesseurs et une méthode qui les manipulent

Cependant, Hamtaro aimerait bien posséder un nom. Et même si il aime faire la Java (haha), il a besoin de beaucoup d'heure de sommeil. Ce sont des attributs que nous souhaiterions lui donner.

Pour se faire, nous ajoutons deux attributs dans la classe (**nom** et **nbHeureSommeil**). Afin de rendre le code robuste, nous créons des accesseurs (méthodes get et set) afin de récupérer ou modifier ces valeurs.

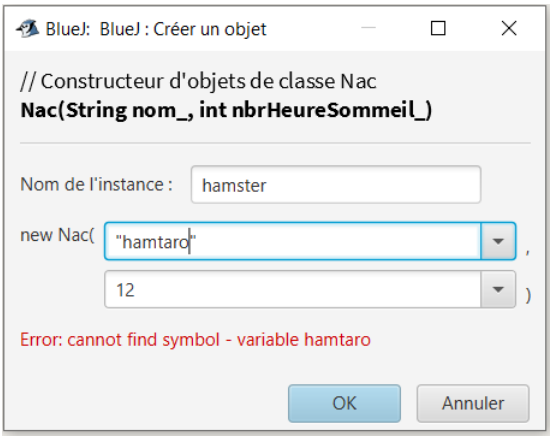
Voici le code implémentant ces modifications :

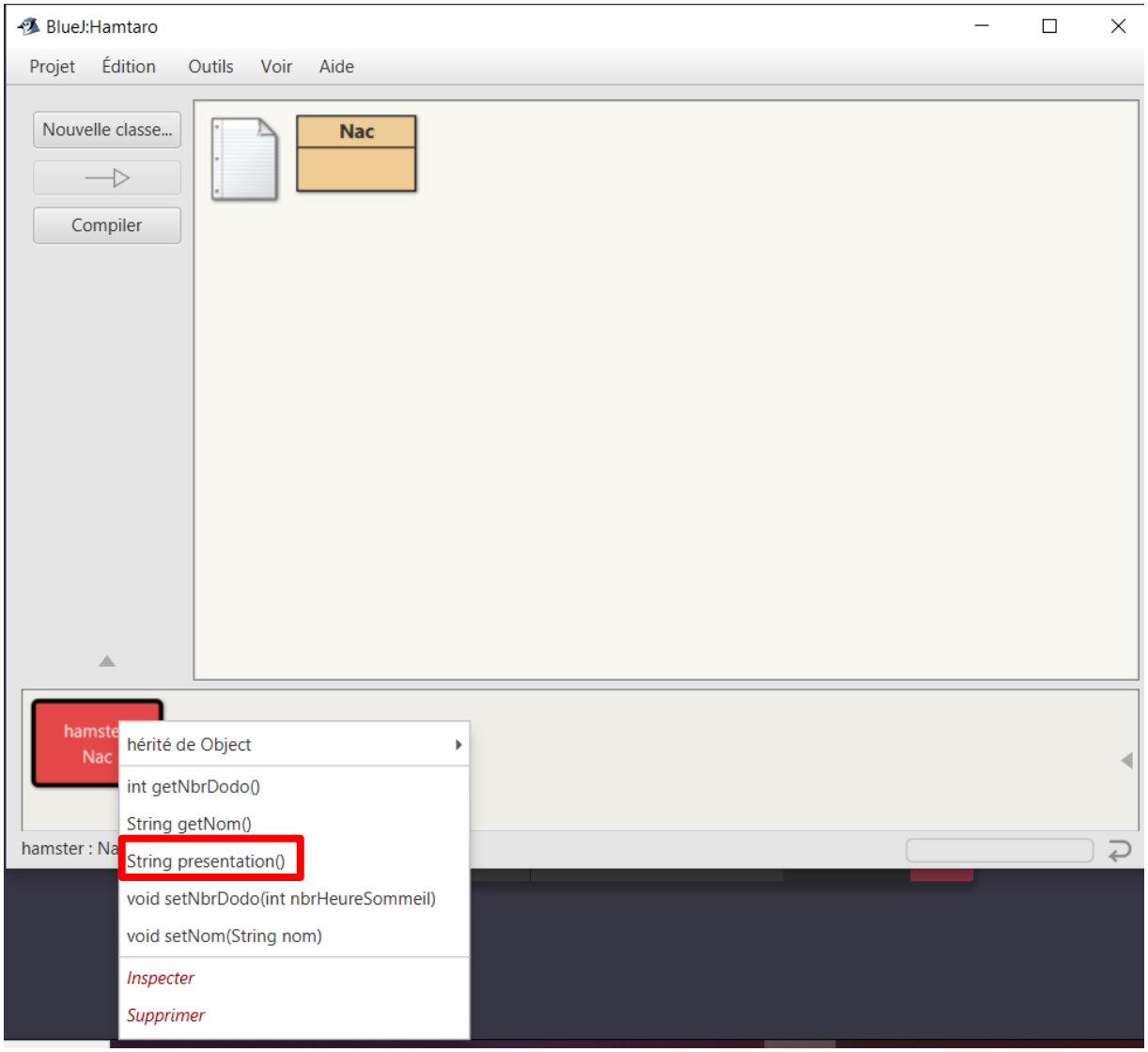
```
public class Nac{
    private String nom;
    private int nbrHeureSommeil;
    /**
     * Constructeur d'objets de classe Nac
     */
    public Nac(String nom, int nbrHeureSommeil){
        this.nom = nom;
        this.nbrHeureSommeil = nbrHeureSommeil;
    }
    public String getNom() {
        return(this.nom);
    }
    public void setNom(String nom){
        this.nom = nom;
    }
    public int getNbrDodo(){
        return(this.nbrHeureSommeil);
    }
    public void setNbrDodo(int nbrHeureSommeil){
        this.nbrHeureSommeil = nbrHeureSommeil;
    }
    public String presentation(){
        return "Je m'appelle " + this.nom + " et je dors " + this.nbrHeureSommeil + "heures.";
    }
}
```

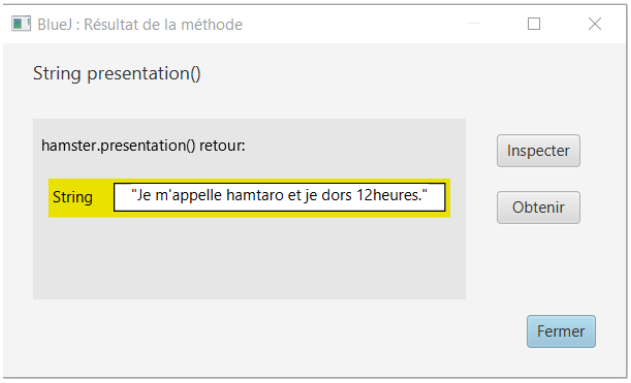
Par ailleurs, une méthode nouvellement créée permet désormais de présenter votre animal ! Et oui en programmation orientée objet on peut tout faire … Même faire parler les animaux.
Pour effectuer ces différentes méthodes, il suffit de faire un clic droit sur l'instance de la classe que vous devrez créer à nouveau grâce au constructeur (en bas à gauche en rouge) et le tour est joué! Vous n'avez qu'à entrer les valeurs souhaitées et vous obtiendrez le résultat escompté.

Q8. Instancier à nouveau

Nous avons par ailleurs modifié le constructeur de la classe (nous ne sommes pas Dieu mais pouvons tout de même mettre au monde nos animaux). Ainsi, notre animal peut être baptisé avant de voir le jour.





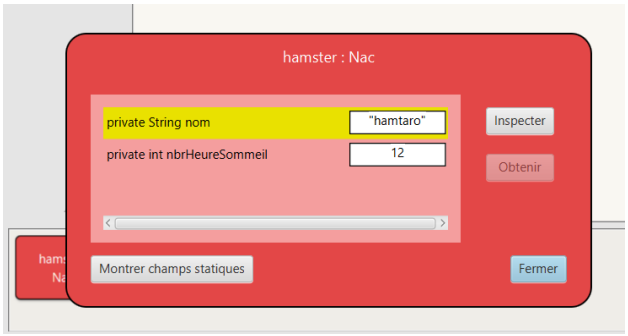




*Figure 10: Retour de la fonction présentation*

## Hamtaro se présente-t-il vraiment bien ?

Q9. Tester unitairement la classe et montrer la barre verte.

Même si nous pouvons faire énormément de choses en programmation orientée objet, la règle primordiale est de tester notre code. On ne voudrait quand même pas qu'Hamtaro se retrouve avec une queue de poisson…

Pour tester les nouvelles fonctionnalités de notre œuvre, il est primordial de créer une nouvelle classe, que l'on nommer NacTest. Pour cela, il suffit d'effectuer un clic droit sur la classe Nac et de sélectionner "Créer classe Test". Une classe verte, nommée "NacTest" apparaît alors derrière la première.

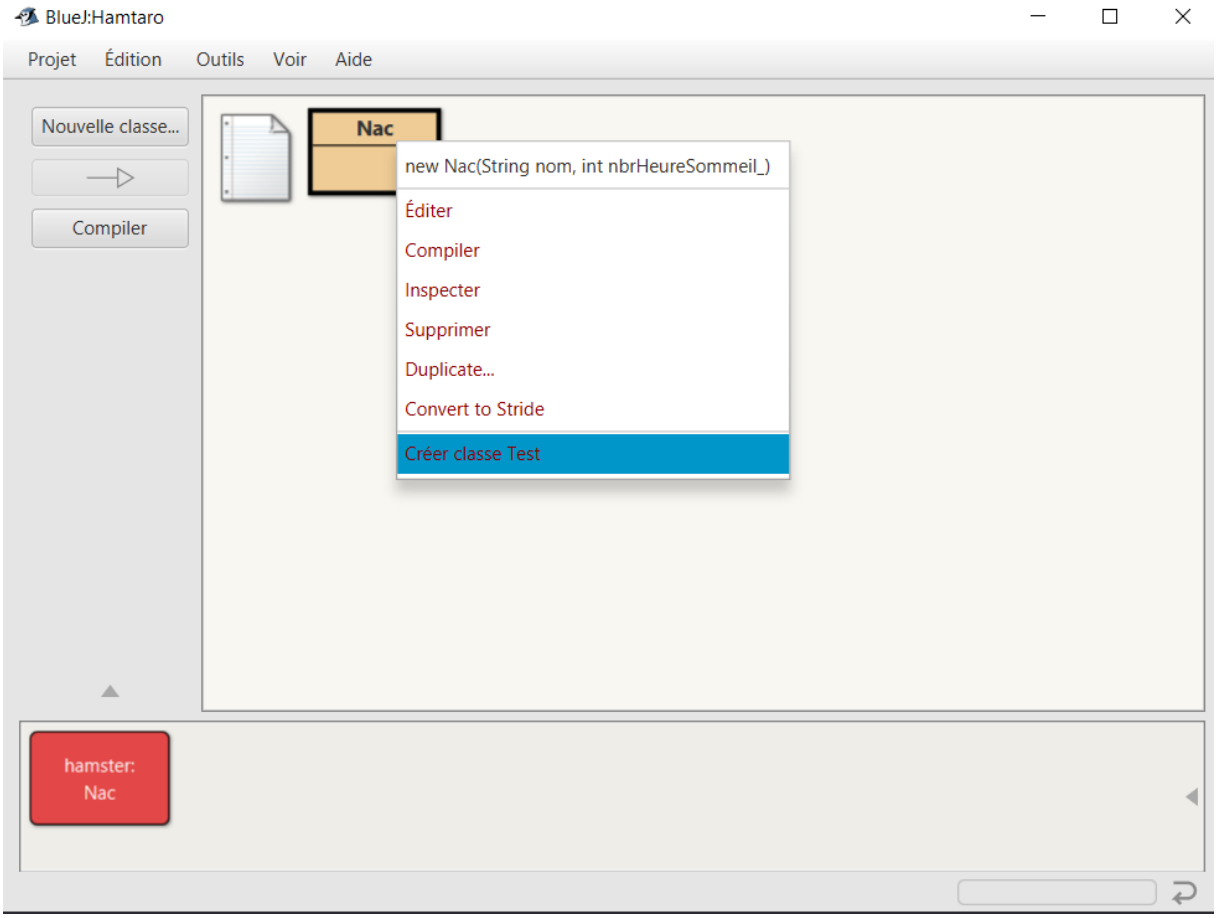



*Figure 11: Création de NacTest*

Vous l'aurez compris, coder en Java demande avant tout de la patience ! Pour tester une méthode créée (nous prendrons pour exemple la méthode présentation), un clic droit sur la classe NacTest vous permettra d'effectuer l'action "Enregistrer une méthode de test"

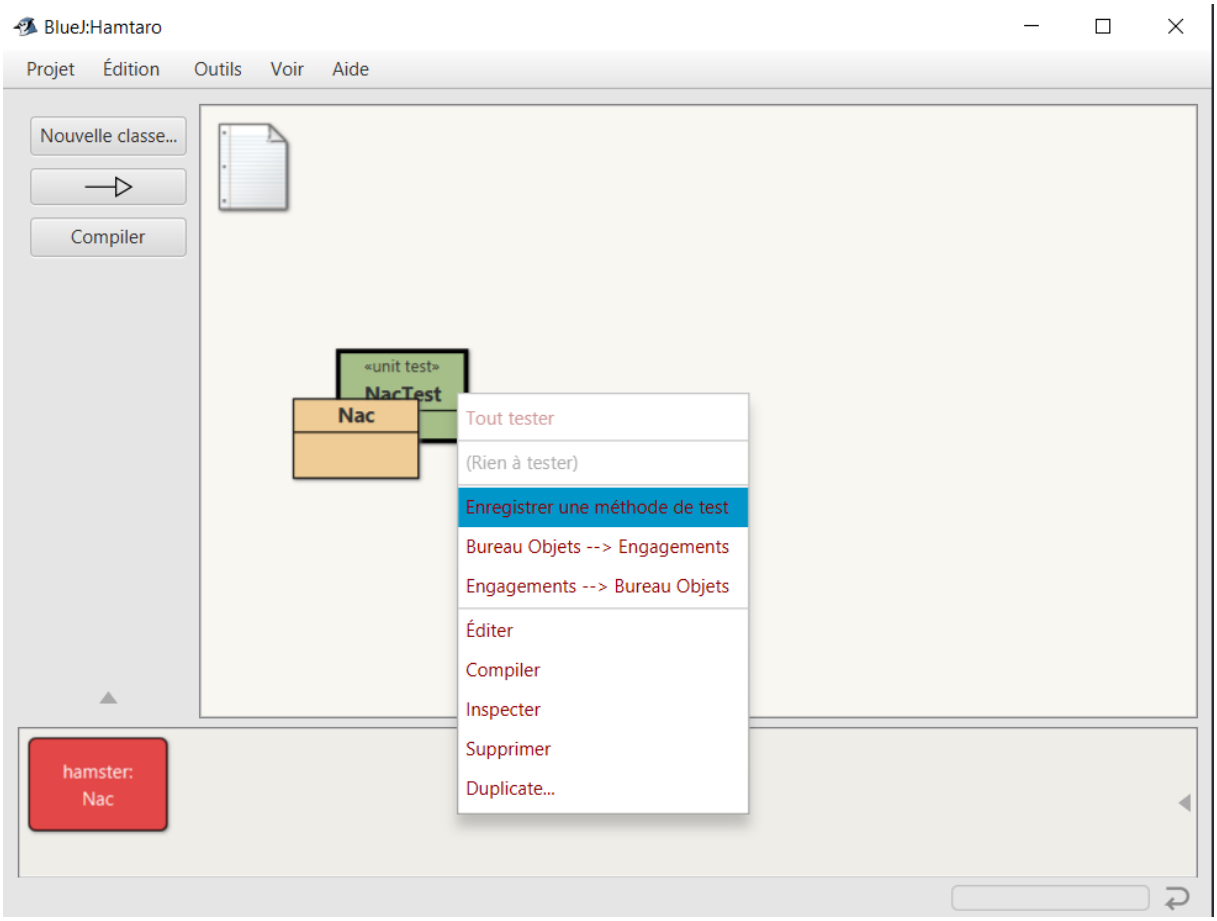



*Figure 12: Enregistrer une méthode de test*

Dès lors, une pop-up s'affiche vous demandant le nom de la méthode de test. Il est de coutume d'un codeur qui se respecte (oui toi lisant ce tutoriel) de nommer la méthode comme suit: nom de la méthode testée (ici présentation) + Test à la fin ce qui nous donne (fuusiioon) presentationTest (eh oui tout le monde n'en est pas capable soyez fiers !)

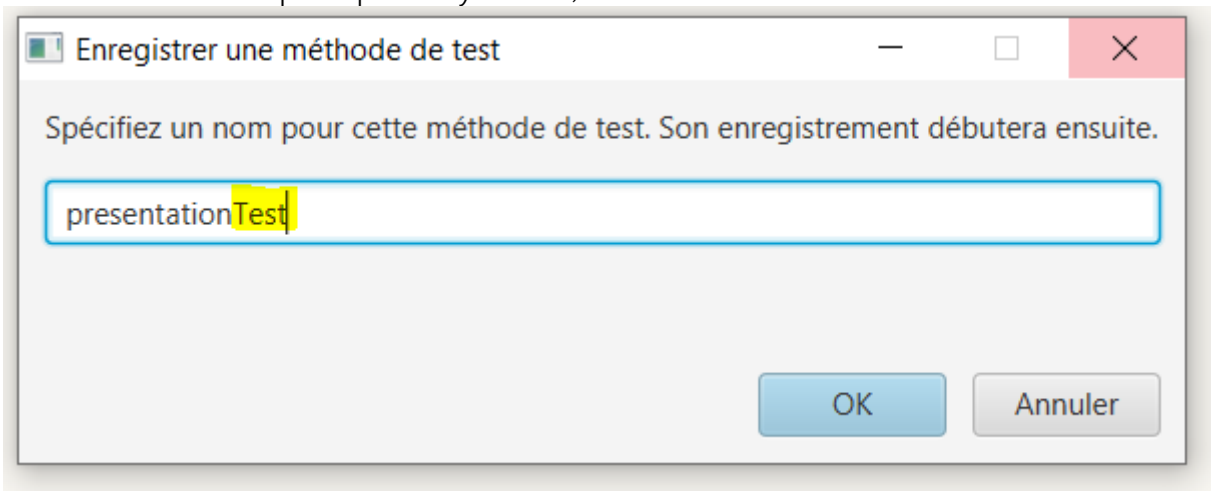
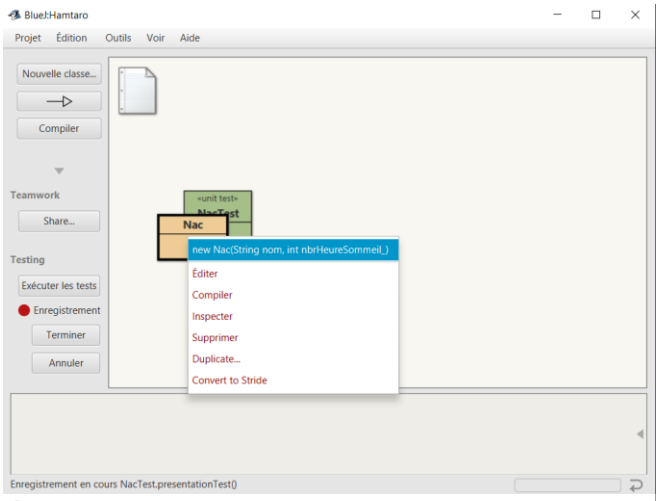
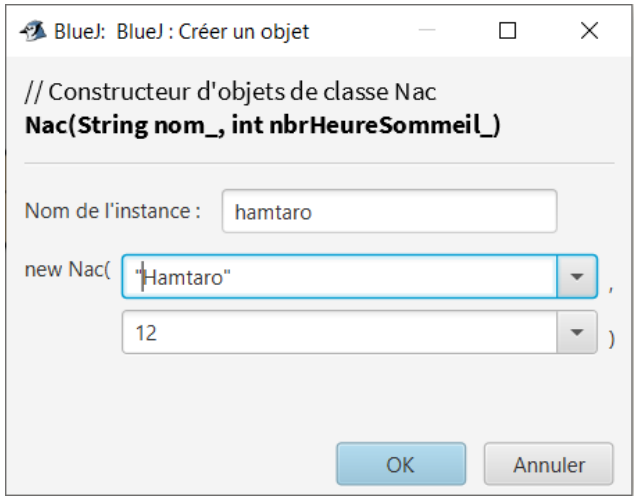


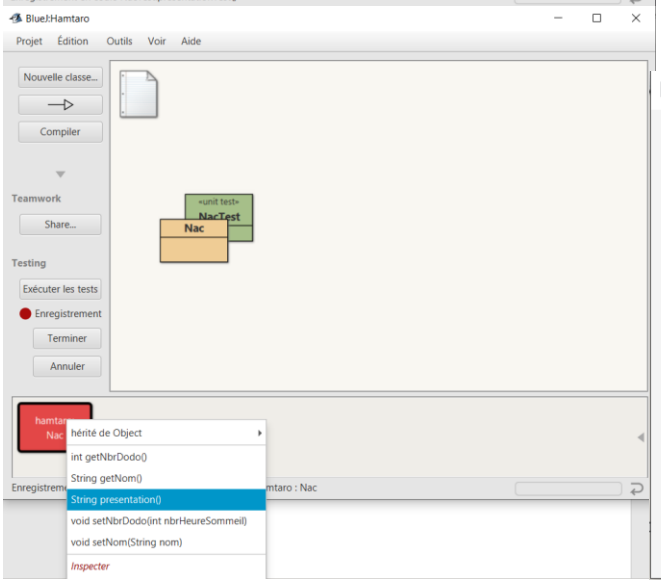

*Figure 13: PopUp intergalactique de fusion (bon ok.. c'est juste un nom)*

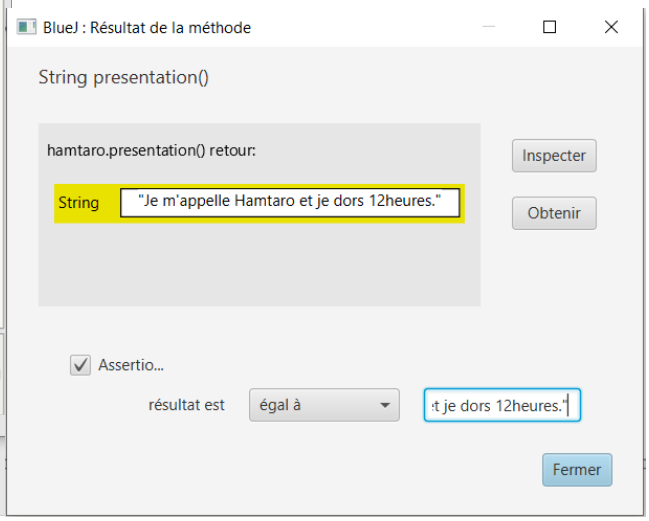
Vous constaterez dès la validation qu'un bouton rouge "enregistrement" apparaît sur la gauche de votre écran et matérialise le début des tests. Pour tester notre fonction présentation, il suffit de créer un nouvel objet puis de cliquer (Hé oui toujours le droit) sur cet objet créé en rouge et enfin de sélectionner la méthode présentation.

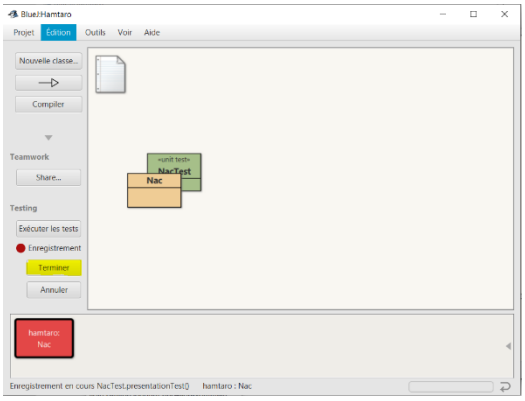
Un petit effort de réflexion vous est demandé (Ohhhh non…). En effet, c'est le moment d'être aussi intelligent que votre programme et d'imaginer le résultat attendu. Celui-ci sera à entrer au niveau du champ "résultat est" et permettra à l'outil de comparer le résultat attendu et celui renvoyé par la méthode. Une fois la fenêtre fermée, vous pouvez arrêter l'enregistrement en cliquant sur le bouton "terminer" à gauche de votre écran.




Si vous êtes curieux, cliquez double sur la nouvelle classe NacTest et vous devriez voir le code suivant apparaître ! (Waaaaaaaaaaa)

```java
@Test
public void presentationTest()
{
    Nac hamtaro = new Nac("Hamtaro", 12);
    assertEquals("Je m'appelle Hamtaro et je dors 12heures.", hamtaro.presentation());
}
```

Pour afficher le résultat de la comparaison (promis c'est la dernière étape), cliquez sur le bouton "exécuter les tests". Tadaaaa, si le programme fonctionne (et si non tant pis je ne recommencerai pas toutes ces étapes), un checkpoint vert apparaît à côté de la méthode vous signifiant le bon fonctionnement. Vous rentrez maintenant dans la cour des grands.

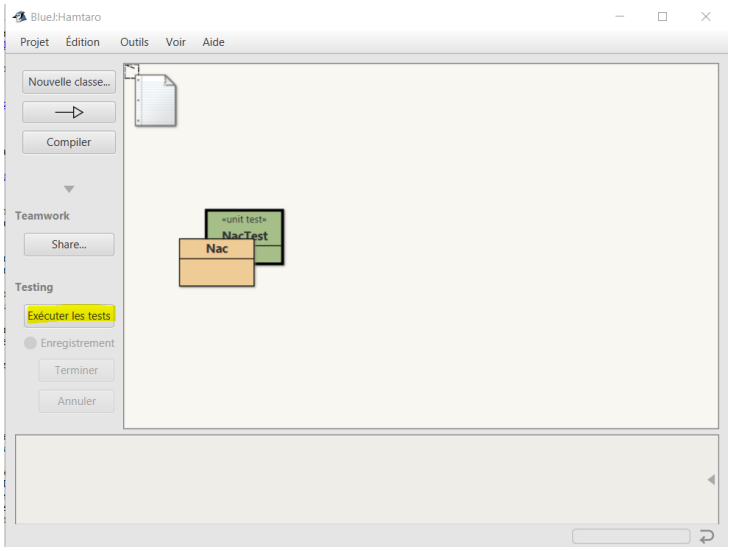



*Figure 14: Exécution des tests*

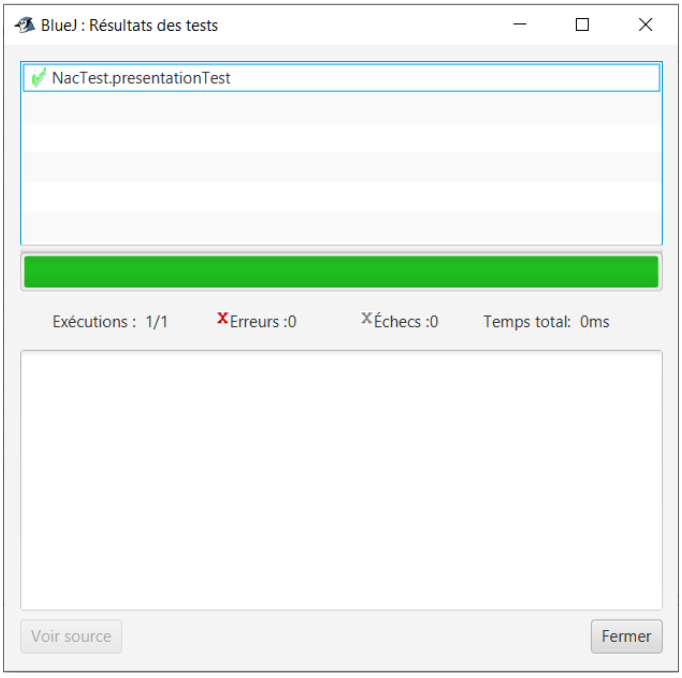



*Figure 15: La barre est verte pas de rouge à l'horizon tout est bon*

Q10. Ajouter une seconde classe et associer la à la classe fétiche avec une multiplicité 0..1 à 0...1
A présent, permettons à Hamtaro de connaître son maître (eh oui uniquement pour pimenter les choses). Pour ce faire, nous créons une classe "Maître" caractérisée par un nom, un âge ainsi qu'un animal. Pour pallier à la solitude du maître au travail ou à l'école, nous lui proposons de pouvoir promener son hamster. (Triviale pour quelqu'un de votre envergure)

Q11. Ajouter une méthode qui collabore avec la classe fétiche

Voici l'implémentation de cette classe. L'idéal serait de ne pas regarder ! On connait tous la technique de la feuille qui cache un peu mais qui glisse de ligne en ligne… 😊

```
public class Maitre
{
    // variables d'instance - remplacez l'exemple qui suit par le vôtre
    private int age;
    private String nom;
    private Nac animal;

    /**
     * Constructeur d'objets de classe Maitre
     */
    public Maitre()
    {
        // initialisation des variables d'instance
        age = 0;
        nom = "Laura";
    }

    public int getAge(){
        return(this.age);
    }
    public void setAge(int number){
        this.age = number;
    }
```

```
public String getNom(){
    return(this.nom);
}
public void setAge(String name){
    this.nom = name;
}
public Nac getAnimal(){
    return(this.animal);
}
public void setAnimal(Nac nc){
    this.animal = nc;
}
public String promener()
{
    // Insérez votre code ici
    return "*Arrive dans son jardin* Hey, je suis " + this.nom + "... Hoooo reviens ici "+ animal.getNom();
}
```

*Figure 16: Implémentation de la classe Maitre*

Nous devons par la suite compiler le tout, mais je ne devrais même plus le dire … puis exécuter notre méthode promener().

Nous allons donc créer Laura et Hamtaro. Les images ci-dessous décrivent la procédure à faire pour lier les deux petits cœurs solitaires. Leur création n'est plus un mystère pour vous alors je vous épargne les répétitions.

Q12. Instancier les classes et relier les objets, les sauvegarder dans la ficture d'une classe test

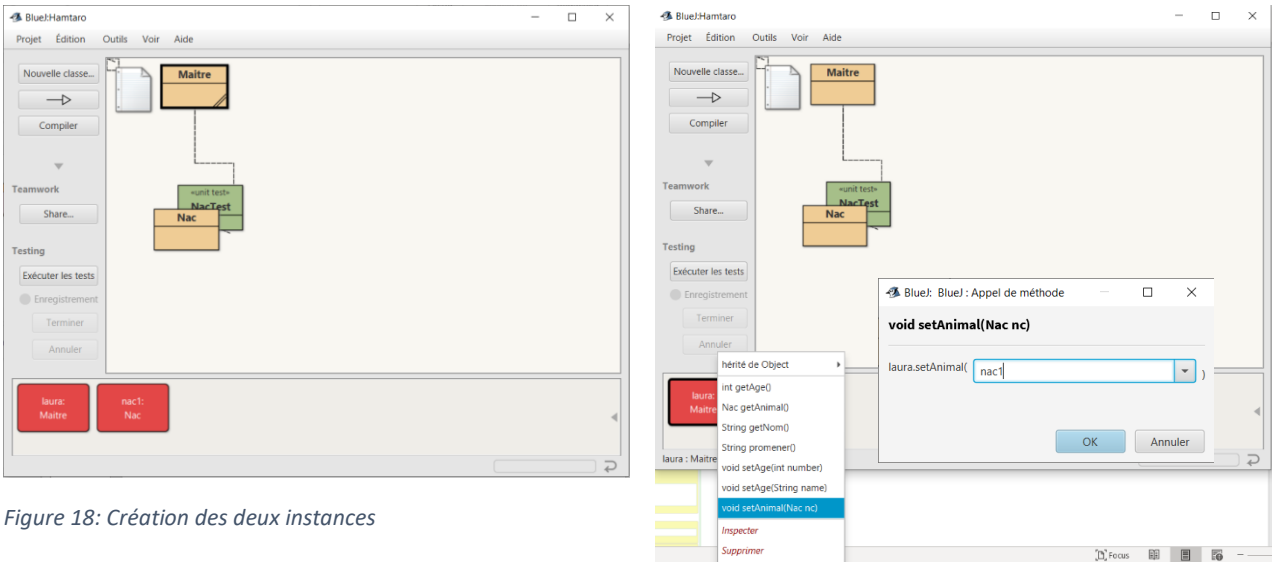



*Figure 18: Création des deux instances*

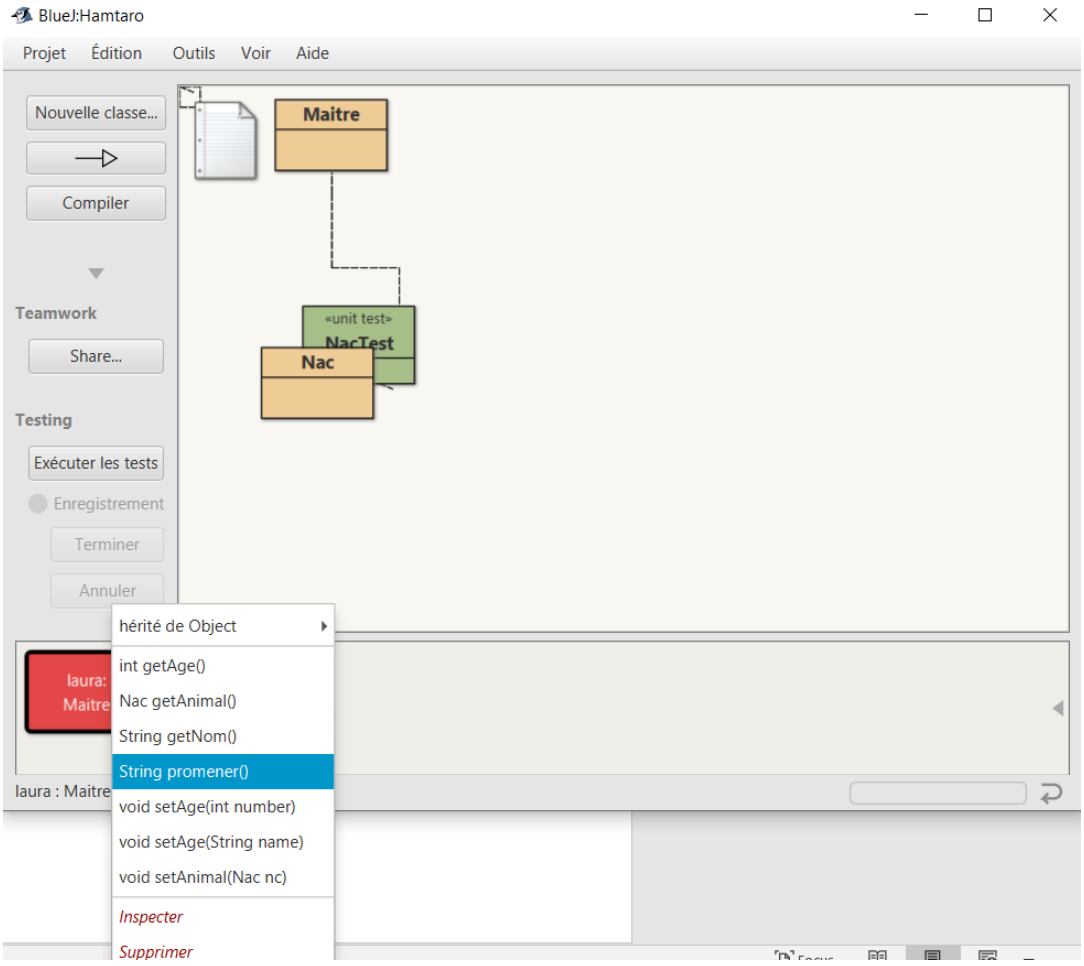
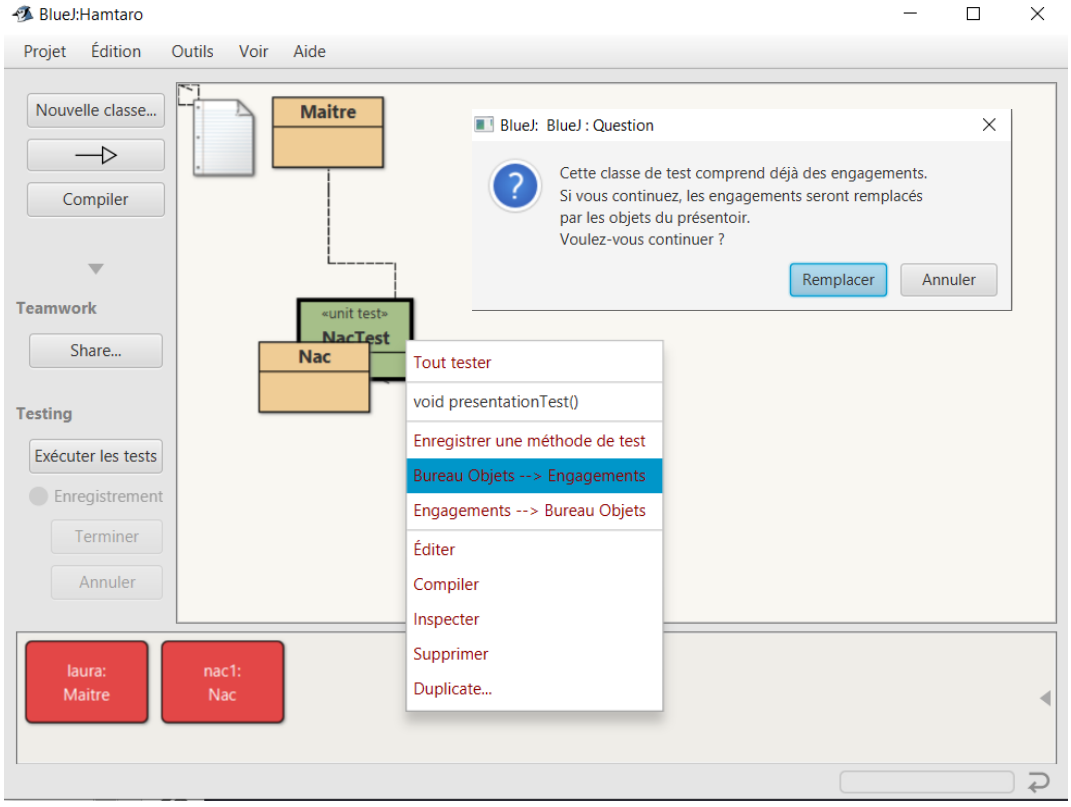



*Figure 19: Débuter la promenade (espérons que tout se passe bien mais ....)*


*Figure 20: Résultat .. CATASTROPHE*

## Encore des tests … toujours des tests

Le test de cette méthode (que vous maîtrisez si bien à présent) requiert une subtilité préalable: l'enregistrement des instances dans la classe test. ("C'est bon il m'a perdu je m'en vais…")




Je m'explique: lorsque vous allez tester la méthode, il faudra préalablement avoir créé un objet (correspondant à l'instance d'une classe) de chaque classe. Ensuite, en un clic (en vérité deux mais le premier ne compte pas) vous pourrez stocker l'animal et son maître dans la classe de test en faisant clic droit sur la classe de test > "Bureau Objets --> Engagements"

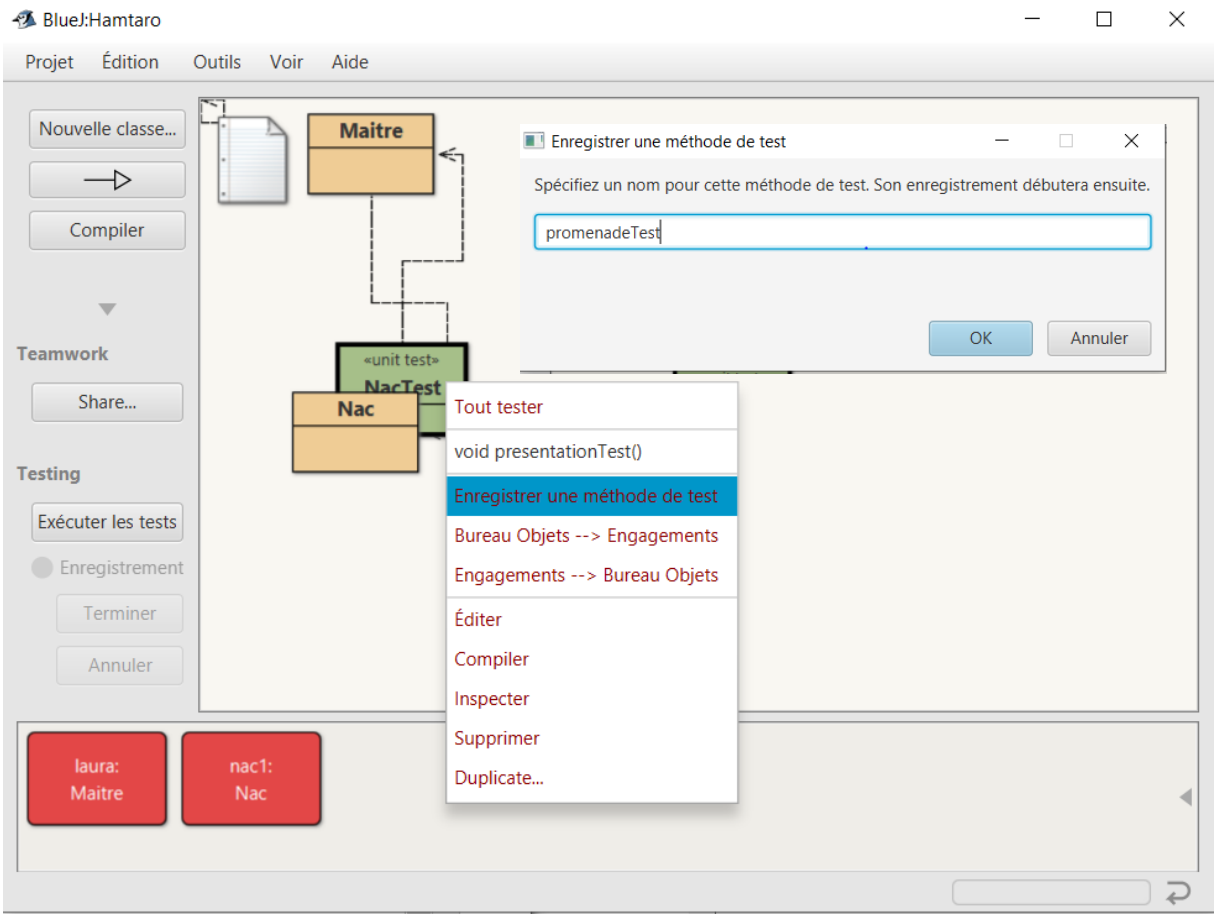
En remplaçant les données présentes par celles utiles au test, il ne vous reste plus qu'à reprendre l'étape de test comme mentionné ci-dessus et confirmer leur bonne exécution.

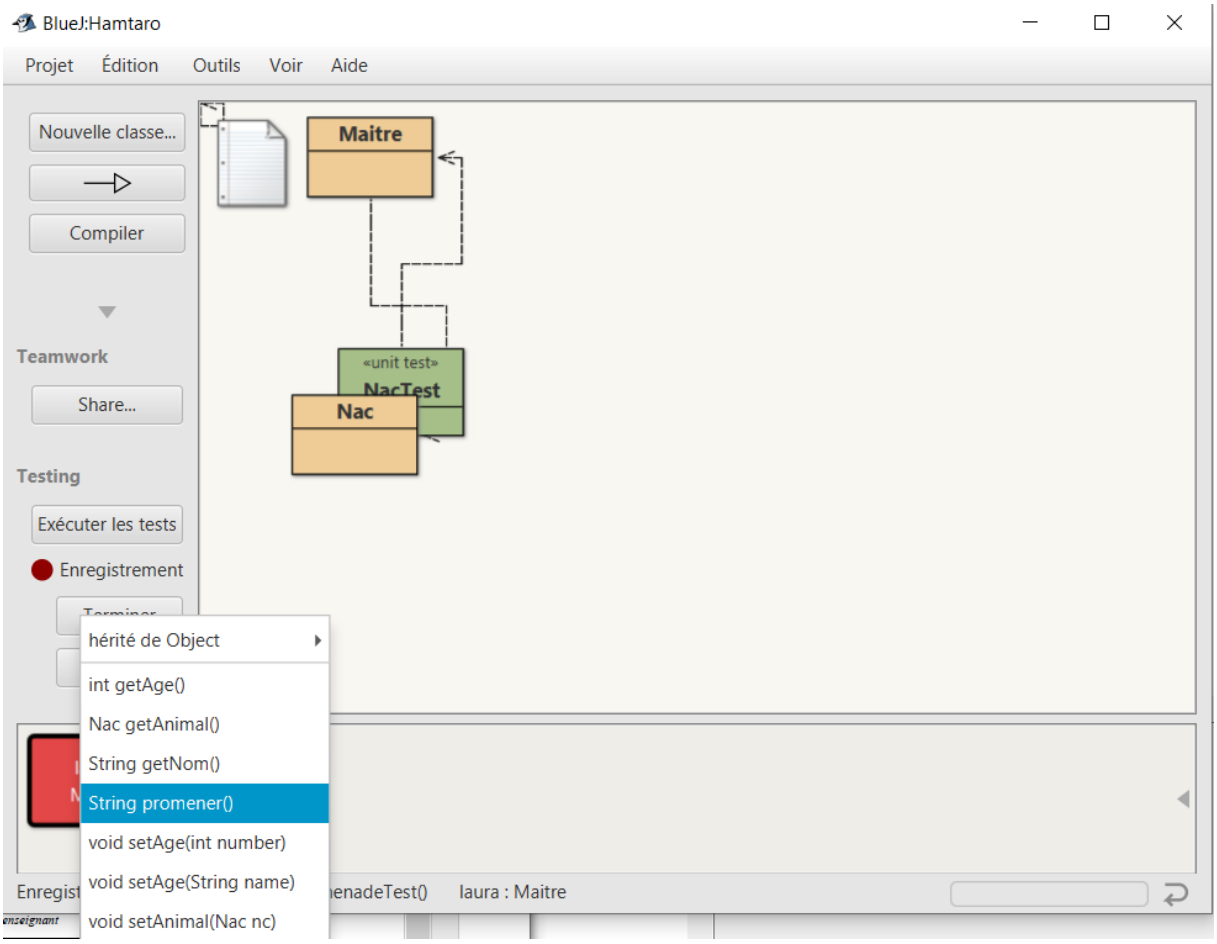
```
@Before
public void setUp() // throws java.lang.Exception
{
    laura = new Maitre();
    nac1 = new Nac("Hamtaro", 12);
    laura.setAnimal(nac1);
}
```

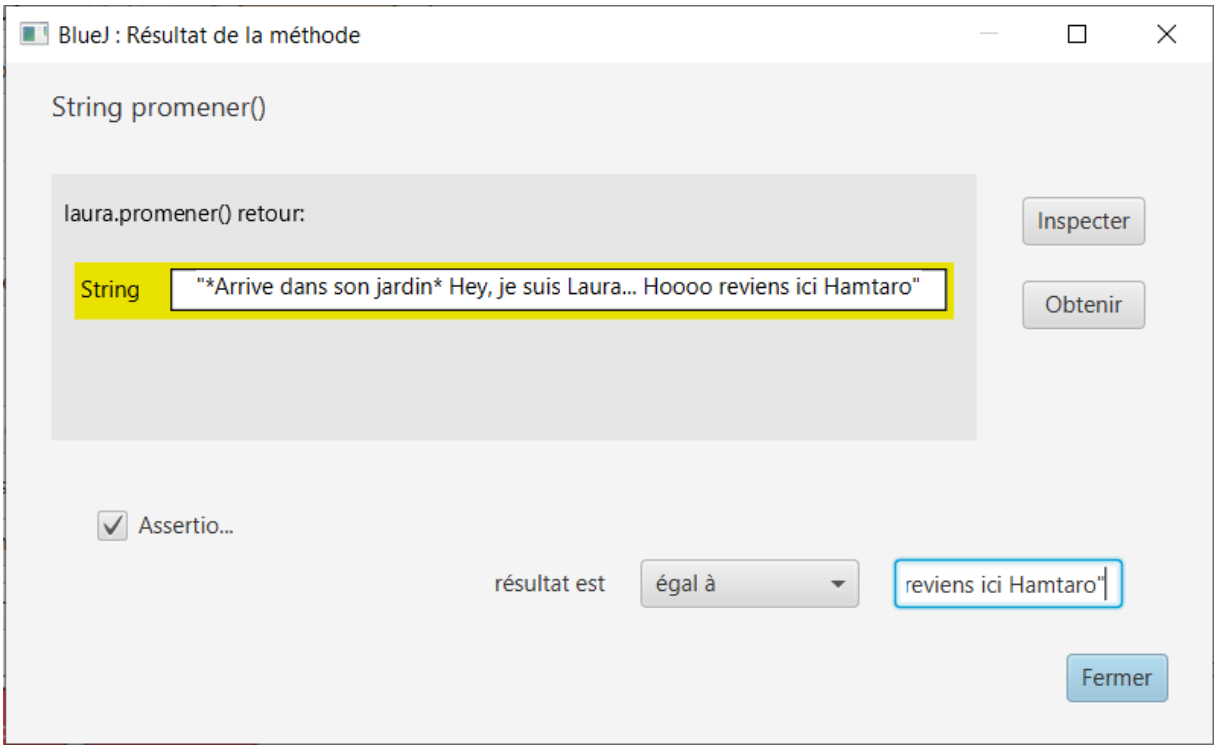
En reprenant le même principe pour la création du test de la promenade vous devriez à nouveau apercevoir non plus une mais deux barres vertes !

Q13. Créer interactivement une méthode de test qui utilise la fixture et montrer le résultat de la barre

Suivez les captures suivantes :



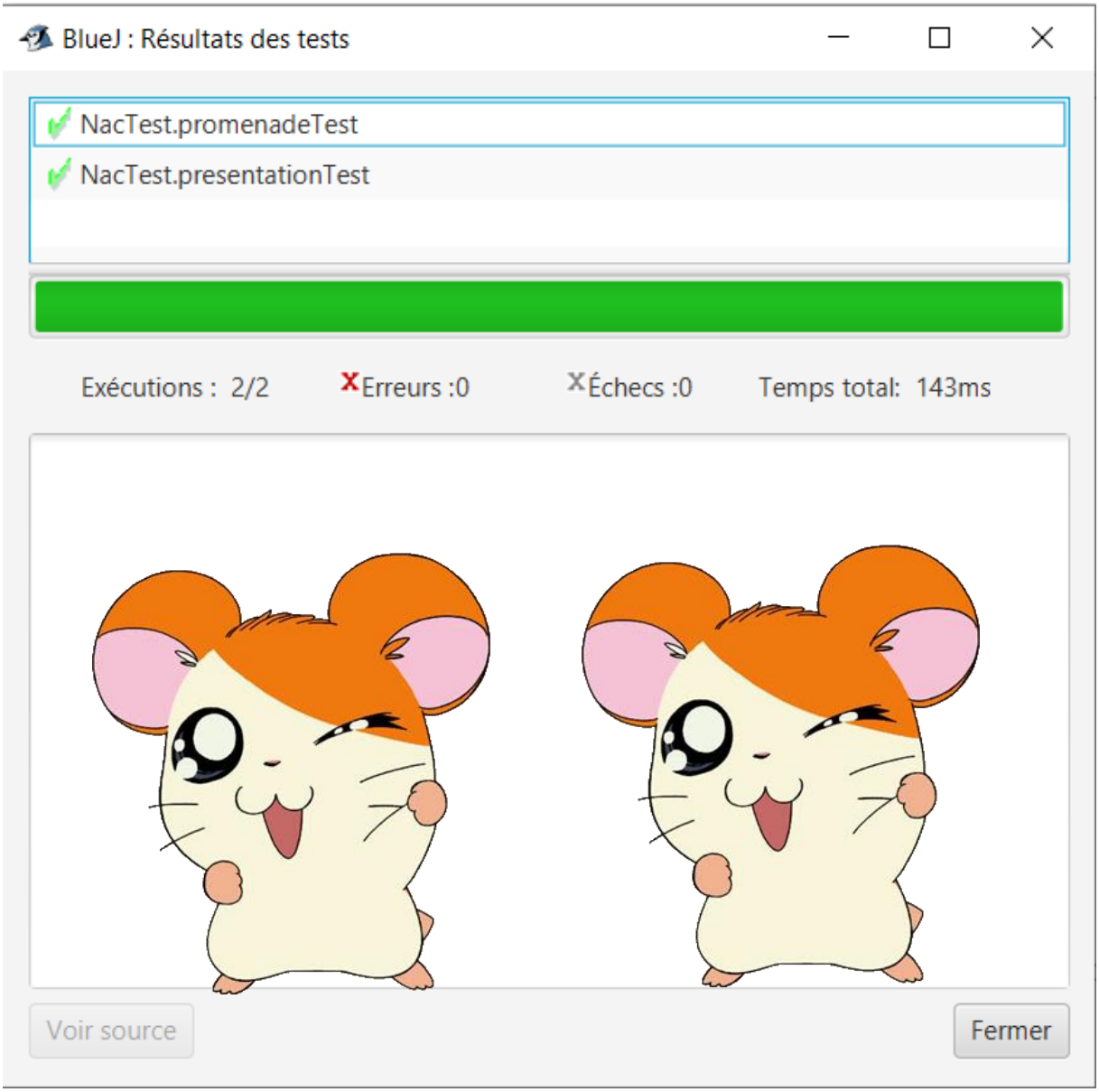

BlueJ:Hamtaro
Projet
Édition
Outils
Voir
Aide
Nouvelle classe...
Compiler
Teamwork
Share...
Testing
Exécuter les tests
Enregistrement
Maitre
«unit test»
NacTest
Nac
hérité de Object
int getAge()
Nac getAnimal()
String getNom()
String promener()
void setAge(int number)
void setAge(String name)
void setAnimal(Nac nc)
laura : Maitre


BlueJ : Résultat de la méthode
String promener()
laura.promener() retour:
String
"*Arrive dans son jardin* Hey, je suis Laura... Hoooo reviens ici Hamtaro"
Inspecter
Obtenir
Assertio...
résultat est
égal à
reviens ici Hamtaro"
Fermer

Voici le code généré par les manipulations précédentes :

```
@Test
public void presentationTest()
{
    Nac hamtaro = new Nac("Hamtaro", 12);
    assertEquals("Je m'appelle Hamtaro et je dors 12heures.", hamtaro.presentation());
}
```

```
@Test
public void promenadeTest()
{
    assertEquals("*Arrive dans son jardin* Hey, je suis Laura... Hoooo reviens ici Hamtaro", laura.promener());
}
```




A bientôt pour de nouvelles aventures ….

# Seconde partie : Eclipse et jUnit

## Création d'un projet avec IntelliJ

Q14. Créer un projet avec un IDE

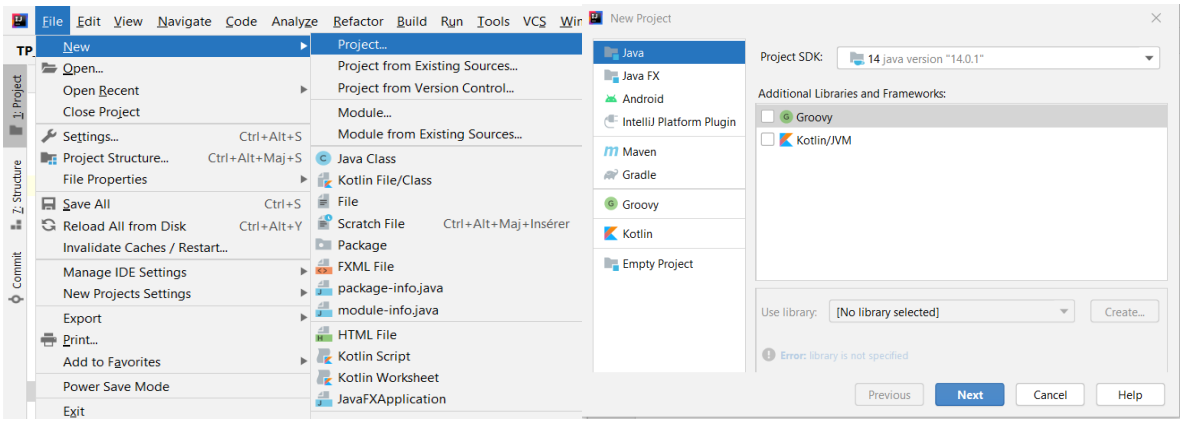
Les petites manipulations avec BlueJ c'est bien beau mais je suppose que vous avez découvert une âme de développeur en vous. C'est donc bel et bien le moment d'aller plus loin.

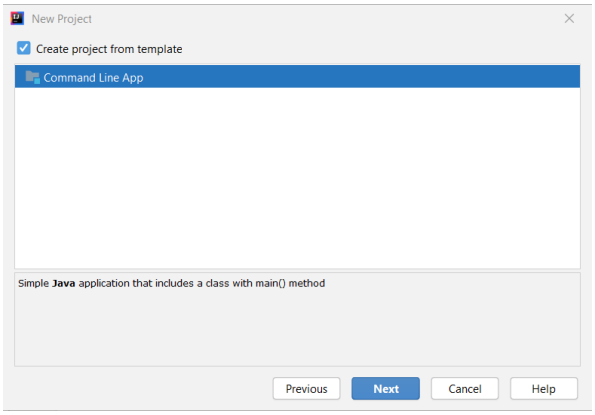
Le défi ici sera de créer la famille d'Hamtaro. En effet … il s'ennuie un peu tout seul à tourner indéfiniment dans sa roue. C'est pourquoi nous allons implémenter une association bidirectionnelle entre le maître et les nouveaux animaux de compagnie.Dans notre modélisation, un animal peut avoir au maximum un maître tandis qu'un maître peut posséder un ou plusieurs mini Hamtaros (trop mignon !).

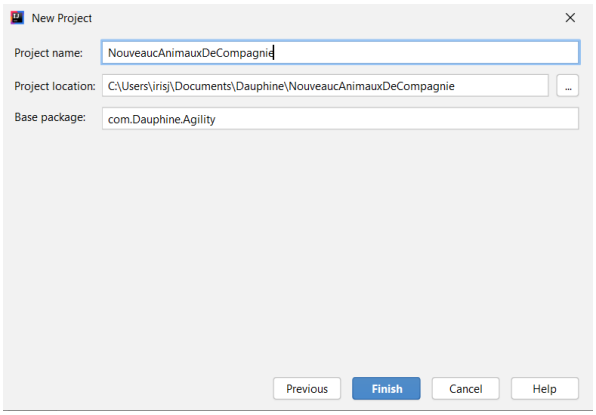
Pour créer un nouveau projet, nous installons Intelli J en cliquant sur le lien suivant : https://www.jetbrains.com/fr-fr/idea/download/#section=windows

Vous pouvez donc cliquer sur File -> New -> Project. Par la suite on sélectionne Java et la version que vous souhaitez utiliser (nous conseillons la version 14 que vous trouverez facilement en téléchargement sur google). Vous pouvez suivre les manipulations indiquées par les images suivante.

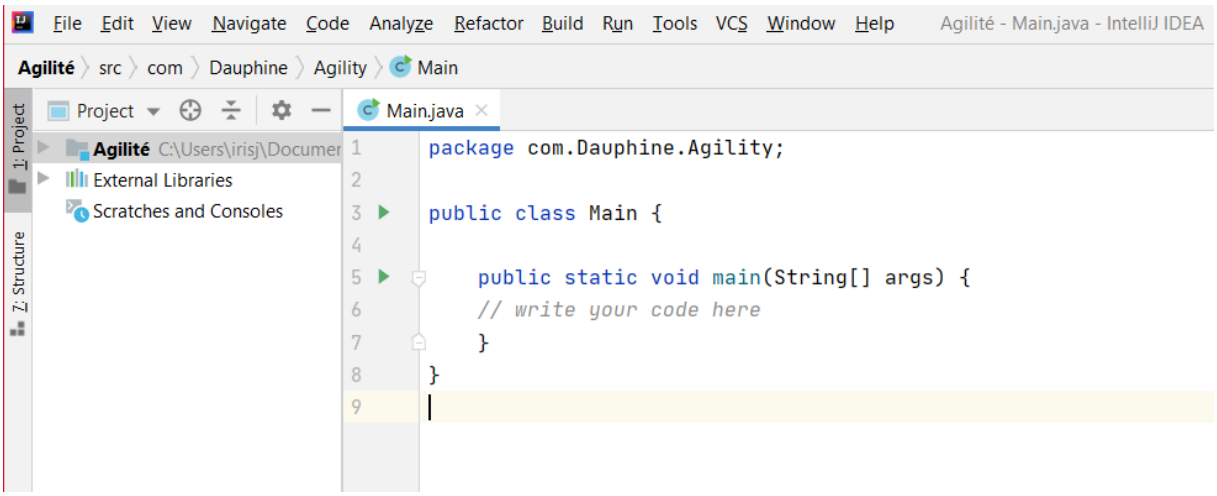
Finalement, vous pouvez choisir le nom souhaité pour manipuler vos petits Hamsters.



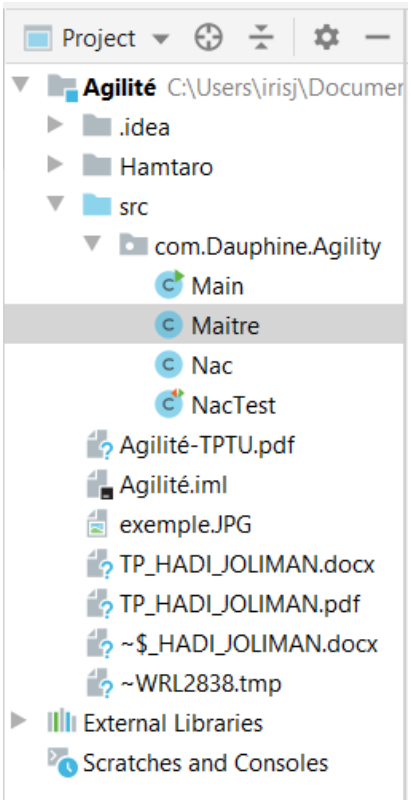



Si jusqu'ici tout s'est bien passé, vous devriez normalement apercevoir quelque chose d'extraordinaire ! .. Non en fait c'est juste ça (pour le moment).

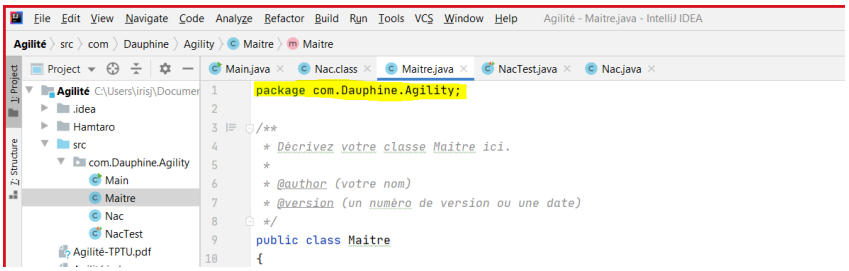



Q15. Importer les classes créées avec BlueJ et créer un package

C'est l'heure de faire déménager notre petit Hamtaro et l'ouvrir au monde extraordinaire de la programmation avec Intelli J. Nous allons donc importer les classes Maître, Nac et NacTest.




Pour cela, nous récupérons dans le fichier Hamtaro créé précédemment puis nous insérons les classes de ce fichier dans le fichier **src/package**. La classe main étant inutile pour la suite, vous pouvez la supprimer.




Finalement, et comme tout déménagement mérite du travail, insérer la ligne surlignée pour toutes vos nouvelles classes.

Nous récupérons donc les classes. Pour permettre à Laura d'adopter tout pleins d'animaux, quelques modifications sont nécessaires, notamment de passer une liste de Nacs en attribut de la classe Maître.

## Oulala une avalanche de code !

Q15. Implémenter une association bidirectionnelle 0..1 à * en encapsulant bien et testez unitairement sa robustesse

D'autres modifications et ajout de fonction sont réalisés afin de permettre une bonne encapsulation (il ne faudrait pas que n'importe qui pénètre dans la maison de Laura pour kidnapper Hamtaro)

## CLASSE MAÎTRE

```java
package com.Dauphine.Agility;
import java.util.ArrayList;
public class Maitre {
    // ****** ATTRIBUTS ****** //
    private int age;
    private String nom;
    private ArrayList<Nac> animaux = new ArrayList<Nac>(); // Un maître possède 0 ou plusieurs NAC
    // ****** CONSTRUCTEURS ****** //
    // Constructeur vide
    public Maitre() {
        this.age = 0;
        this.nom = "Laura";
    }
    // Ce constructeur permet de créer un maître ne possédant pas de nac
    public Maitre(int age, String nom) {
        this.age = age;
        this.nom = nom;
    }
```

---

```java
    // Ce constructeur permet de créer un maître possédant des Nacs
    public Maitre(int age, String nom, ArrayList<Nac> animal) {
        this.age = age;
        this.nom = nom;
        this.animaux = animal;
        for (Nac puppy : this.animaux) {
            Maitre puppyMaitre = puppy.getMaitre();
            if(puppyMaitre != null) {
                if (!puppy.getMaitre().equals(this)) {
                    puppy.setMaitre(this);
                }
            }
            else{
                puppy.setMaitre(this); } }
    }
    // ****** ACCESSEURS ****** //
    public int getAge(){ return(this.age); }
    public void setAge(int number) { this.age = number; }
    public String getNom() { return(this.nom); }
    public void setNom(String name) { this.nom = name; }
    public ArrayList<Nac> getAnimal(){
        return(this.animaux); }
    public void setAnimal(ArrayList<Nac> nc){
        if(nc != null) {
            this.animaux = nc;
            for (Nac puppy : this.animaux) {
                if (!puppy.getMaitre().equals(this)) {
                    puppy.setMaitre(this); } } }
    }
```

```
// ****** METHODES ****** //

@Override
public String toString() { return "Je suis : " + this.nom + " et j'ai " + this.age + " ans."; }
@Override
public boolean equals(Object obj){
    boolean retour = false;
    if (obj!= null && (obj.getClass().equals(this.getClass()))){
        if (obj instanceof Maitre){
            Maitre maitre = (Maitre)obj;
            if(checkAnimalExistance(maitre) && checkAnimalExistance( maitre: this)){
                retour = this.nom.equals(maitre.getNom()) && this.age == maitre.getAge(); }
            else{
                if(checkAnimalExistance(maitre) || checkAnimalExistance( maitre: this)){
                    retour = false; }
                else{
                    retour = this.nom.equals(maitre.getNom()) &&
                            this.age == maitre.getAge() && this.animaux.equals(maitre.getAnimal());
                } } } }
    return(retour);
}
private boolean checkAnimalExistance(Maitre maitre) {
    return maitre.getAnimal().size() == 0;
}
// Cette fonction ajoute un animal à la liste déjà existante
public void addNac(Nac puppy) {
    if(! this.isNacInAnimals(puppy)) {
        this.animaux.add(puppy);
    }
}


 public boolean isNacInAnimals(Nac puppy){
     for (Nac mypuppy:this.animaux) {
         if(puppy.equals(mypuppy)){
             return(true);
         }
     }
     return(false);
 }

 // Préciser qu'on doit modifier la méthode promener dans le rapport
 public String promener() {
     String promenade = "Pour cette promenade, " + this.nom + " vous présente :";
     for (Nac puppy:this.animaux) {
         promenade = promenade + '\n' + '-' + puppy.presentation();
     }
     return(promenade);
 }
```

## CLASSE NAC

```java
package com.Dauphine.Agility;
// La classe Nac représente un Nouvel Animal de Compagnie
public class Nac{

    // ****** ATTRIBUTS ****** //
    private String nom;
    private int nbrHeureSommeil;
    private Maitre maitre; // Un animal possède 0 ou 1 maître

    // ****** CONSTRUCTEURS ****** //
    // Constructeur vide
    public Nac(){
        this.nom = "animal sans nom";
        this.nbrHeureSommeil = 0;
        this.maitre = null; }
    // Ce constructeur permet de créer un animal ne possédant pas de maître à sa création
    public Nac(String nom, int nbrHeureSommeil){
        this.nom = nom;
        this.nbrHeureSommeil = nbrHeureSommeil;
        this.maitre = null; }
    // Ce constructeur permet de créer un animal possédant un maître à sa création
    public Nac(String nom, int nbrHeureSommeil, Maitre maitre){
        this.nom = nom;
        this.nbrHeureSommeil = nbrHeureSommeil;
        this.maitre = maitre;
        maitre.addNac( puppy: this); }
```

---

```java
// ****** ACCESSEURS ****** //
public String getNom() { return(this.nom); }
public void setNom(String nom) { this.nom = nom; }
public int getNbrDodo() { return(this.nbrHeureSommeil); }
public void setNbrDodo(int nbrHeureSommeil) { this.nbrHeureSommeil = nbrHeureSommeil; }
public Maitre getMaitre() { return(this.maitre); }
public void setMaitre(Maitre maitre){
    this.maitre = maitre;
    maitre.addNac( puppy: this); }

// ****** METHODES ****** //
@Override
public String toString(){
    String retour =  "Je suis : " + this.nom + " et j'ai besoin de " + this.nbrHeureSommeil + " heures de sommeil.";
    if(this.maitre != null){
        retour = retour + "\n Mon maître s'appelle : " + this.maitre.getNom(); }
    return(retour); }
```

---

```java
@Override
public boolean equals(Object obj){
    boolean retour = false;
     if (obj!= null && (obj.getClass().equals(this.getClass()))){
        if (obj instanceof Nac){
            Nac puppy = (Nac)obj;
            if(puppy.getMaitre() == null && this.maitre == null){
                retour = this.nom.equals(puppy.getNom()) && this.nbrHeureSommeil == puppy.getNbrDodo();
            }
            else {
                if(puppy.getMaitre() == null || this.maitre == null){
                    retour = false;
                }
                else {
                    retour = this.nom.equals(puppy.getNom()) &&
                            this.nbrHeureSommeil == puppy.getNbrDodo() && this.maitre.equals(puppy.getMaitre());
                } } } }
    return(retour);
}

public String presentation(){
    return "Je m'appelle " + this.nom + " et je dors " + this.nbrHeureSommeil + " heures.";
}
```

## CLASSE MAÎTRE TEST

```java
package com.Dauphine.Agility;
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import java.util.ArrayList;
public class MaitreTest {

    public MaitreTest() { }
    @Before
    public void setUp() // throws java.lang.Exception
    { }
    @After
    public void tearDown() // throws java.lang.Exception
    { }

    @Test
    public void constructeurVideTest() {
        Maitre m1 = new Maitre();
        assertEquals(m1.getNom(), actual: "Laura");
        assertEquals(m1.getAge(), actual: 0);
        assertEquals(m1.getAnimal(), new ArrayList<Nac>()); }

    @Test
    public void constructeur2argumentsTest() {
        Maitre m1 = new Maitre( age: 10, nom: "Laye");
        assertEquals(m1.getNom(), actual: "Laye");
        assertEquals(m1.getAge(), actual: 10);
        assertEquals(m1.getAnimal(), new ArrayList<Nac>()); }
```

```java
@Test
public void constructeur3argumentsTest() {
    Nac n1 = new Nac( nom: "Hamtaro",  nbrHeureSommeil: 12);
    Nac n2 = new Nac( nom: "Hamtara",  nbrHeureSommeil: 11);
    Nac n3 = new Nac( nom: "Hamtaru",  nbrHeureSommeil: 10);
    ArrayList<Nac> myNacs = new ArrayList<Nac>();
    myNacs.add(n1);
    myNacs.add(n2);
    myNacs.add(n3);
    Maitre m1 = new Maitre( age: 10,  nom: "Laye", myNacs);
    assertEquals(m1.getNom(),  actual: "Laye");
    assertEquals(m1.getAge(),  actual: 10);
    assertEquals(m1.getAnimal(), myNacs);
    for (Nac puppy:m1.getAnimal()) {
        assertEquals(m1, puppy.getMaitre());
    }
    Maitre m2 = new Maitre( age: 10,  nom: "Simona");
    Nac n4 = new Nac( nom: "Hamtara",  nbrHeureSommeil: 11, m1);
    ArrayList<Nac> myNacsZam = new ArrayList<Nac>();
    myNacsZam.add(n4);
    Maitre m3 = new Maitre( age: 60, nom: "Zam", myNacsZam);
    assertEquals(m3.getNom(),  actual: "Zam");
    assertEquals(m3.getAge(),  actual: 60);
    assertEquals(myNacsZam, m3.getAnimal());
    for (Nac puppy:m3.getAnimal()) {
        assertEquals(m3, puppy.getMaitre()); }
}
```

```java
@Test
public void setNomTest() {
    Maitre m1 = new Maitre( age: 10,  nom: "Laye");
    m1.setNom("Bijou");
    assertEquals( expected: "Bijou", m1.getNom()); }
@Test
public void setAge() {
    Maitre m1 = new Maitre( age: 10,  nom: "Laye");
    m1.setAge(20);
    assertEquals( expected: 20, m1.getAge()); }
@Test
public void setAnimalTest() {
    Maitre m2 = new Maitre( age: 10,  nom: "Laye");
    Nac n1 = new Nac( nom: "Hamtaro",  nbrHeureSommeil: 12,m2);
    Nac n2 = new Nac( nom: "Hamtara",  nbrHeureSommeil: 11,m2);
    Nac n3 = new Nac( nom: "Hamtaru",  nbrHeureSommeil: 10,m2);
    Nac n5 = new Nac( nom: "amtaro",  nbrHeureSommeil: 2,m2);
    Nac n6 = new Nac( nom: "amtara",  nbrHeureSommeil: 1,m2);
    Nac n7 = new Nac( nom: "amtaru",  nbrHeureSommeil: 0,m2);
    ArrayList<Nac> myNacs = new ArrayList<Nac>();
    ArrayList<Nac> myNacsNew = new ArrayList<Nac>();
    myNacs.add(n1);
    myNacs.add(n2);
    myNacs.add(n3);
    myNacsNew.add(n5);
    myNacsNew.add(n6);
    myNacsNew.add(n7);
    Maitre m1 = new Maitre( age: 10,  nom: "Laye", myNacs);
    m1.setAnimal(myNacsNew);
    assertEquals(myNacsNew, m1.getAnimal()); }
```

```java
@Test
public void getNomTest() {
    Maitre m1 = new Maitre( age: 10,  nom: "Laye");
    assertEquals( expected: "Laye", m1.getNom()); }

@Test
public void getAgeTest() {
    Nac n1 = new Nac( nom: "Hamtaro",  nbrHeureSommeil: 12);
    Nac n2 = new Nac( nom: "Hamtara",  nbrHeureSommeil: 11);
    Nac n3 = new Nac( nom: "Hamtaru",  nbrHeureSommeil: 10);
    ArrayList<Nac> myNacs = new ArrayList<Nac>();
    myNacs.add(n1);
    myNacs.add(n2);
    myNacs.add(n3);
    Maitre m1 = new Maitre( age: 10,  nom: "Laye",myNacs);
    assertEquals( expected: 10, m1.getAge()); }

@Test
public void getAnimals() {
    Nac n1 = new Nac( nom: "Hamtaro",  nbrHeureSommeil: 12);
    Nac n2 = new Nac( nom: "Hamtara",  nbrHeureSommeil: 11);
    Nac n3 = new Nac( nom: "Hamtaru",  nbrHeureSommeil: 10);
    ArrayList<Nac> myNacs = new ArrayList<Nac>();
    myNacs.add(n1);
    myNacs.add(n2);
    myNacs.add(n3);
    Maitre m1 = new Maitre( age: 10,  nom: "Laye", myNacs);
    assertEquals(myNacs, m1.getAnimal()); }
```

```java
@Test
public void toStringTest(){
    Maitre maitre = new Maitre( age: 8,  nom: "Laura");
    assertEquals(maitre.toString(),  actual: "Je suis : Laura et j'ai 8 ans."); }

@Test
public void equalsTest(){
    Nac n1 = new Nac( nom: "Hamtaro",  nbrHeureSommeil: 12);
    Nac n2 = new Nac( nom: "Hamtara",  nbrHeureSommeil: 11);
    Nac n3 = new Nac( nom: "Hamtaru",  nbrHeureSommeil: 10);
    Nac n5 = new Nac( nom: "amtaro",  nbrHeureSommeil: 2);
    Nac n6 = new Nac( nom: "amtara",  nbrHeureSommeil: 1);
    Nac n7 = new Nac( nom: "amtaru",  nbrHeureSommeil: 0);
    ArrayList<Nac> myNacs1 = new ArrayList<Nac>();
    ArrayList<Nac> myNacs2 = new ArrayList<Nac>();
    myNacs1.add(n1);
    myNacs1.add(n2);
    myNacs1.add(n3);
    myNacs2.add(n5);
    myNacs2.add(n6);
    myNacs2.add(n7);
    Maitre maitre0 = new Maitre( age: 8,  nom: "Laura");
    Maitre maitre1 = new Maitre( age: 10,  nom: "Laurent",myNacs2);
    Maitre maitre2 = new Maitre( age: 10,  nom: "Laurent",myNacs1);
    Maitre maitre3 = new Maitre( age: 10,  nom: "Laurent",myNacs1);
    Maitre maitre4 = new Maitre( age: 8,  nom: "Phil");
    assertEquals( expected: true, maitre3.equals(maitre2));
    assertEquals( expected: false, maitre0.equals(maitre1));
    assertEquals( expected: false, maitre1.equals(maitre2));
    assertEquals( expected: false, maitre0.equals(maitre4)); }
```

```java
@Test
public void promenadeTest() {
    Nac n1 = new Nac( nom: "Hamtaro",  nbrHeureSommeil: 12);
    Nac n2 = new Nac( nom: "Hamtara",  nbrHeureSommeil: 11);
    Nac n3 = new Nac( nom: "Hamtaru",  nbrHeureSommeil: 10);
    ArrayList<Nac> myNacs1 = new ArrayList<Nac>();
    myNacs1.add(n1);
    myNacs1.add(n2);
    myNacs1.add(n3);
    Maitre maitre2 = new Maitre( age: 10,  nom: "Laurent",myNacs1);
    assertEquals( expected: "Pour cette promenade, Laurent vous présente :\n" +
            "-Je m'appelle Hamtaro et je dors 12 heures.\n" +
            "-Je m'appelle Hamtara et je dors 11 heures.\n" +
            "-Je m'appelle Hamtaru et je dors 10 heures.", maitre2.promener()); }
```

```java
@Test
public void addNacTest(){
    Nac n1 = new Nac( nom: "Hamtaro",  nbrHeureSommeil: 12);
    Nac n2 = new Nac( nom: "Hamtara",  nbrHeureSommeil: 11);
    Nac n3 = new Nac( nom: "Hamtaru",  nbrHeureSommeil: 10);
    Nac n4 = new Nac( nom: "NewAnimal",  nbrHeureSommeil: 4);
    ArrayList<Nac> myNacs = new ArrayList<Nac>();
    ArrayList<Nac> myNacsNew = new ArrayList<Nac>();
    ArrayList<Nac> myNacsOne = new ArrayList<Nac>();
    myNacs.add(n1);
    myNacs.add(n2);
    myNacs.add(n3);
    myNacsNew = myNacs;
    myNacsNew.add(n4);
    myNacsOne.add(n4);
    Maitre maitreWithoutAnimal = new Maitre( age: 10,  nom: "Laurent");
    maitreWithoutAnimal.addNac(n4);
    Maitre maitreWithAnimal = new Maitre( age: 8,  nom: "Laura", myNacs);
    maitreWithAnimal.addNac(n4);
    assertEquals(myNacsNew, maitreWithAnimal.getAnimal());
    assertEquals(myNacsOne, maitreWithoutAnimal.getAnimal());
    for (Nac puppy:maitreWithAnimal.getAnimal()) {
        assertEquals(maitreWithAnimal, puppy.getMaitre());
    }
}
```

```java
@Test
public void isNacInAnimalsTest(){
    Nac n1 = new Nac( nom: "Hamtaro",  nbrHeureSommeil: 12);
    Nac n2 = new Nac( nom: "Hamtara",  nbrHeureSommeil: 11);
    Nac n3 = new Nac( nom: "Hamtaru",  nbrHeureSommeil: 10);

    Nac n4 = new Nac( nom: "NewAnimal",  nbrHeureSommeil: 4);

    ArrayList<Nac> myNacs = new ArrayList<Nac>();
    myNacs.add(n1);
    myNacs.add(n2);
    myNacs.add(n3);
    Maitre maitreWithAnimal = new Maitre( age: 8,  nom: "Laura", myNacs);
    assertEquals( expected: true, maitreWithAnimal.isNacInAnimals(n3));
    assertEquals( expected: false, maitreWithAnimal.isNacInAnimals(n4));
}
```

## NAC TEST

```java
package com.Dauphine.Agility;
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class NacTest
{

    public NacTest() { }
    @Before
    public void setUp() // throws java.lang.Exception
    { }
    @After
    public void tearDown() // throws java.lang.Exception
    {}
    @Test
    public void constructeurVideTest(){
        Nac nac1 = new Nac();
        assertEquals(nac1.getNom(), actual: "animal sans nom");
        assertEquals(nac1.getNbrDodo(), actual: 0);
        assertEquals(nac1.getMaitre(), actual: null); }
    @Test
    public void constructeur2argumentsTest(){
        Nac nac2 = new Nac( nom: "Hamtaro", nbrHeureSommeil: 12);
        assertEquals(nac2.getNom(), actual: "Hamtaro");
        assertEquals(nac2.getNbrDodo(), actual: 12);
        assertEquals(nac2.getMaitre(), actual: null);
    }
  @Test
  public void setNbrHeureSommeilTest(){
      Nac n = new Nac( nom: "Hamtaro", nbrHeureSommeil: 12);
      n.setNbrDodo(20);
      assertEquals( expected: 20, n.getNbrDodo()); }
  @Test
  public void setMaitreTest(){
      Maitre maitre = new Maitre( age: 8, nom: "Laura");
      Maitre maitre2 = new Maitre( age: 8, nom: "Zam");
      Nac n = new Nac( nom: "Hamtaro", nbrHeureSommeil: 12, maitre);
      n.setMaitre(maitre2);
      assertEquals(maitre2, n.getMaitre()); }
  @Test
  public void equalsTest(){
      Maitre maitre = new Maitre( age: 8, nom: "Laura");
      Maitre maitre1 = new Maitre( age: 10, nom: "Laurent");
      Nac n1 = new Nac( nom: "Hamtaro", nbrHeureSommeil: 12, maitre);
      Nac n2 = new Nac( nom: "Hamtaro", nbrHeureSommeil: 12, maitre);
      Nac n3 = new Nac( nom: "Hamtaro", nbrHeureSommeil: 12, maitre1);
      Nac n4 = new Nac( nom: "Hamtaro", nbrHeureSommeil: 12);
      Nac n5 = new Nac( nom: "Hamtaro", nbrHeureSommeil: 12);
      assertEquals( expected: true, n4.equals(n5));
      assertEquals( expected: true, n1.equals(n2));
      assertEquals( expected: false, n3.equals(n2));
      assertEquals( expected: false, n2.equals(n4));
      assertEquals( expected: false, n4.equals(n2)); }
```

```java
  @Test
  public void constructeur3argumentsTest(){
      Maitre maitre3 = new Maitre( age: 8, nom: "Laura");
      Nac nac3 = new Nac( nom: "Hamtaro", nbrHeureSommeil: 12, maitre3);
      assertEquals(nac3.getNom(), actual: "Hamtaro");
      assertEquals(nac3.getNbrDodo(), actual: 12);
      assertEquals(nac3.getMaitre(), maitre3); }
  @Test
  public void getNomTest(){
      Nac n = new Nac( nom: "Hamtaro", nbrHeureSommeil: 12);
      assertEquals( expected: "Hamtaro", n.getNom()); }
  @Test
  public void getNbrHeureSommeilTest(){
      Nac n = new Nac( nom: "Hamtaro", nbrHeureSommeil: 12);
      assertEquals( expected: 12, n.getNbrDodo()); }
  @Test
  public void getMaitreTest(){
      Maitre maitre = new Maitre( age: 8, nom: "Laura");
      Nac n = new Nac( nom: "Hamtaro", nbrHeureSommeil: 12, maitre);
      assertEquals(maitre, n.getMaitre()); }
  @Test
  public void setNomTest(){
      Nac n = new Nac( nom: "Hamtaro", nbrHeureSommeil: 12);
      n.setNom("Bijou");
      assertEquals( expected: "Bijou", n.getNom()); }
```

```java
'est
ıblic void toStringTest(){
    Maitre maitre = new Maitre( age: 8, nom: "Laura");
    Nac n1 = new Nac( nom: "Hamtaro", nbrHeureSommeil: 12);
    Nac n2 = new Nac( nom: "Hamtara", nbrHeureSommeil: 10, maitre);
    assertEquals(n1.toString(), actual: "Je suis : Hamtaro et j'ai besoin de 12 heures de sommeil.");
    assertEquals(n2.toString(), actual: "Je suis : Hamtara et j'ai besoin de 10 heures de sommeil.\n" +
            " Mon maitre s'appelle : Laura");

'est
ıblic void presentationTest()

    Nac hamtaro = new Nac( nom: "Hamtaro", nbrHeureSommeil: 12);
    Maitre maitre = new Maitre( age: 8, nom: "Laura");
    assertEquals( expected: "Je m'appelle Hamtaro et je dors 12 heures.", hamtaro.presentation());
```

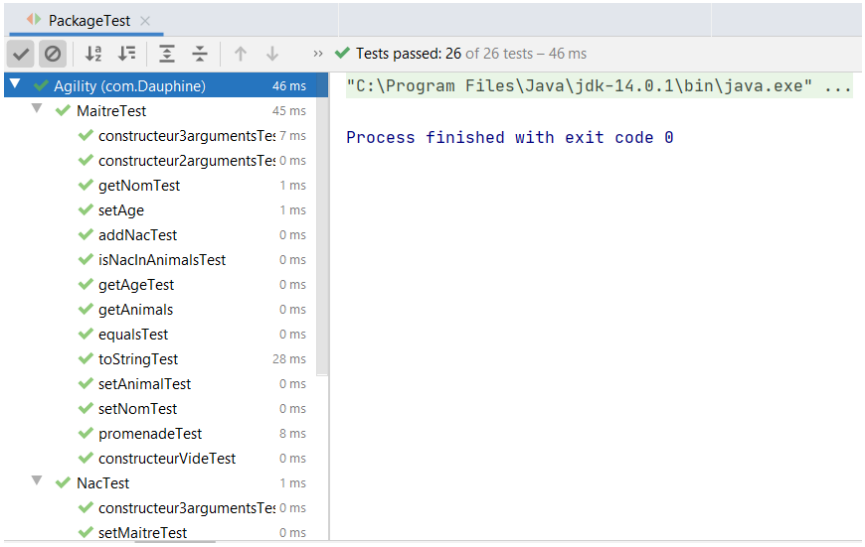
## Escalade de Tests

| 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" |

*Figure 21: Escalade de tests*

Pour vérifier que tout se passe bien dans la conception de notre Hamtaro et de sa maîtresse Laura, il est important de vérifier l'exhaustivité de nos tests. C'est donc le moment checklist / fierté ;) !

- **Convention de nommage:** Pour ne pas se perdre dans le code et mettre toutes les chances de notre côté pour ce grand déménagement, nous avons porté attention aux nommages de nos variables (hamtaro par exemple), à ceux de nos classes (Hamtaro par exemple) et à nos fonctions de tests (HamtaroTest par exemple).
- **Test JUnit:** Nous en avons vécu des hauts et des bas, ou plutôt des rouges et des verts. Malgré toutes ces montagnes russes, nous ne sommes pas restés dans la roue sans fin d'Hamtaro et avons pu donc verdir notre barre de test ! (magnifique non ?)




- **Suppression du Sys.Out et du main:** Là encore on avait tout prévu d'avance (trop forts) … Et oui on veut vraiment que vous puissiez manipuler les instances d'Hamtaro au plus vite ! Donc à bas les sys.out et les main de notre code !

- **Tester les getters:** Comme vous pouvez le constater tous les getters ont été testés avec tous les amis de notre petit Hamtaro.
- **Tester les constructeurs:** Les constructeurs ont effectivement été testés (Et oui les 3!)
- **Tester les setters:** Avec de la douleur certes, mais nous avons également pu tester tous les setters et éviter les boucles indéfinies
- **Tester les associations:** Et oui si Hamtaro change de maître, son maître l'adoptera effectivement et inversement ! (Pas de petit Hamster qui se perd durant sa nouvelle adoption on est pas comme ça nous)
- **Tester les méthodes simples**: Différentes valeurs, différents Hamsters et tout plein de maîtres. Tout ce petit monde nous a permis de tester avec une certaine exhaustivité nos méthodes simples
- **Tester les méthodes complexes:** Nous avons simplifié les choses au maximum. Notre seule méthode complexe à tester était la méthode equals.
- **Tester les exceptions:** Nous avons géré les exceptions à l'intérieur de notre code grâce aux "If"
- **Couverture du code:** Les résultats sont tout simplement stupéfiants, vous pouvez être fiers de vous ! Je vous laisse profiter de la satisfaction de voir que d'une part tous les indicateurs sont au vert et d'autre part que vous avez couvert toutes les lignes de votre code (attention cela ne veut pas dire plus jamais de beugs… mais Hamtaro pourra démarrer sa vie sous les meilleurs hospices)

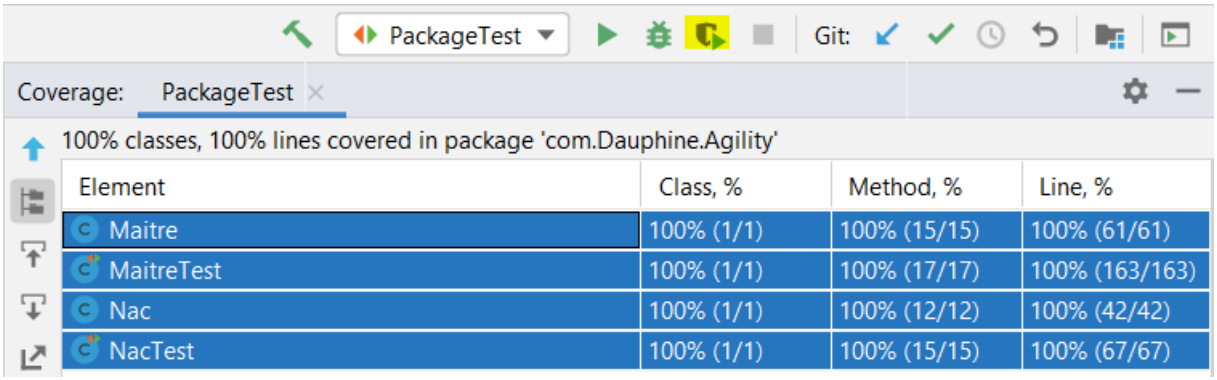



*Figure 22: Pourcentage de couverture du code*

On peut donc s'endormir sur nos lauriers …

## Refactoring

Q17. Illustration de deux techniques de refactoring: rename et extractMethod

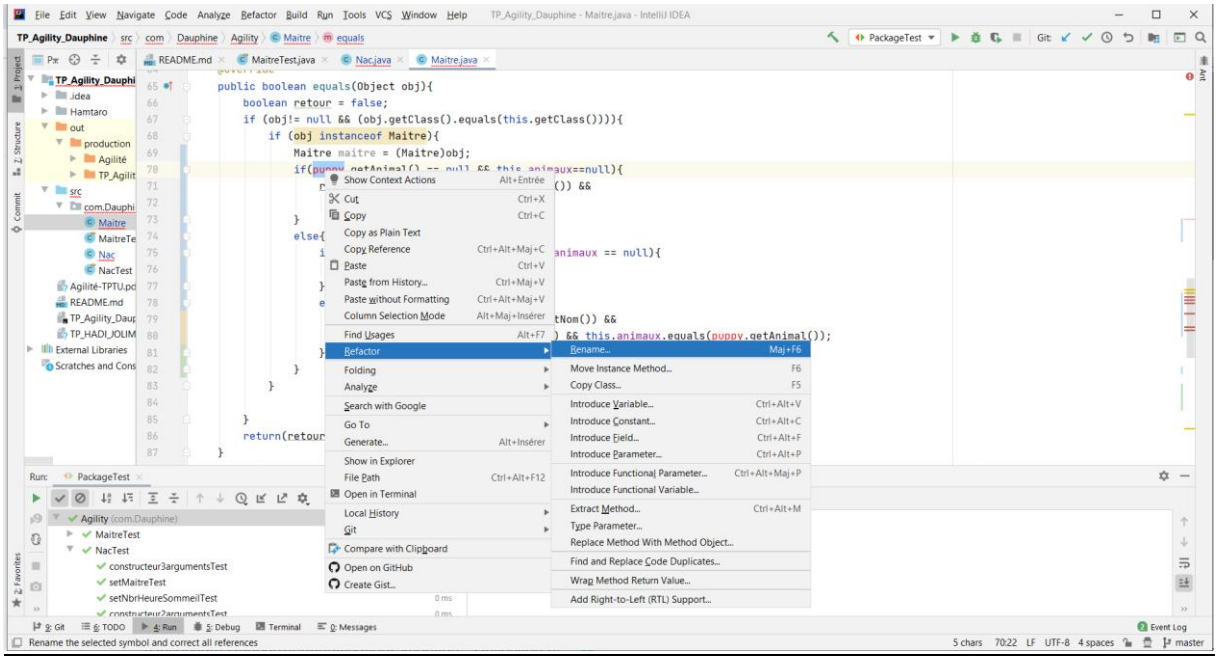



*Figure 23: Refactoring (renaming)*

```
public boolean equals(Object obj){
    boolean retour = false;
    if (obj!= null && (obj.getClass().equals(this.getClass()))){
        if (obj instanceof Maitre){
            Maitre maitre = (Maitre)obj;
            if(puppy.getAnimal() == null && this.animaux==null){
                retour = this.nom.equals(puppy.getNom()) &&
                        this.age == puppy.getAge();
            }
            else{
                if(puppy.getAnimal() == null || this.animaux == null){
                    retour = false;
                }
                else{
                    retour = this.nom.equals(puppy.getNom()) &&
                            this.age == puppy.getAge() && this.animaux.equals(puppy.getAnimal());
                }
            }
        }
```

*Figure 24: Avant refactoring*

```
public boolean equals(Object obj){
    boolean retour = false;
    if (obj!= null && (obj.getClass().equals(this.getClass()))){
        if (obj instanceof Maitre){
            Maitre maitre = (Maitre)obj;
            if(maitre.getAnimal() == null && this.animaux==null){
                retour = this.nom.equals(maitre.getNom()) &&
                        this.age == maitre.getAge();
            }
            else{
                if(maitre.getAnimal() == null || this.animaux == null){
                    retour = false;
                }
                else{
                    retour = this.nom.equals(maitre.getNom()) &&
                            this.age == maitre.getAge() && this.animaux.equals(maitre.getAnimal());
                }
            }
        }
```

*Figure 25: Après refactoring*

Afin de modifier les variables nommées “puppy” par “maître” de manière exhaustive, nous avons utilisés la fonction rename. Fini les heures de travail devant son ordinateur à rechercher le nom de la variable ou encore le fameux CTRL+F pour effectuer le remplacement: RENAME est la pour vous !

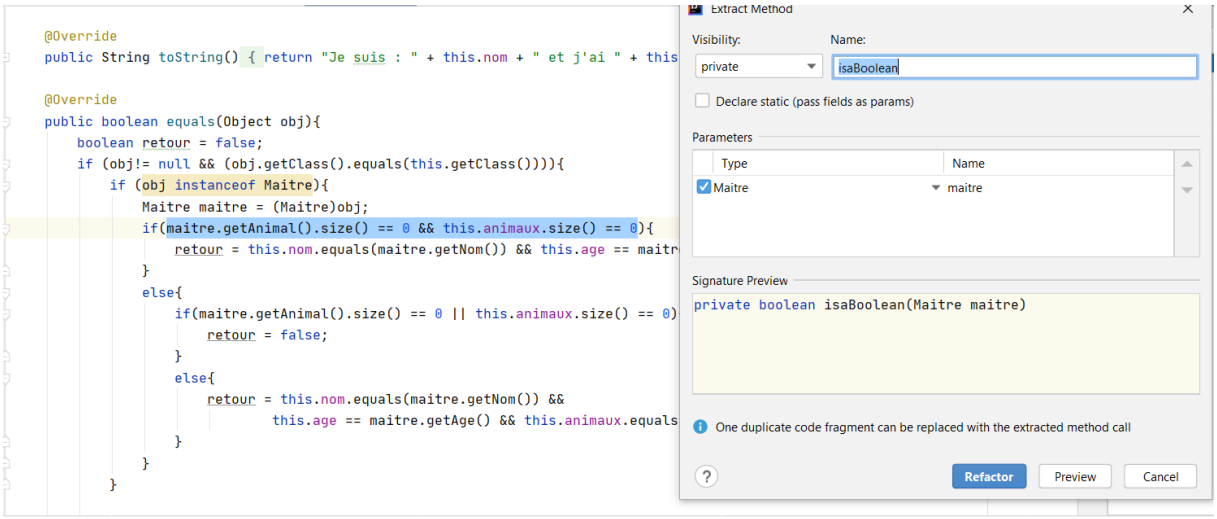
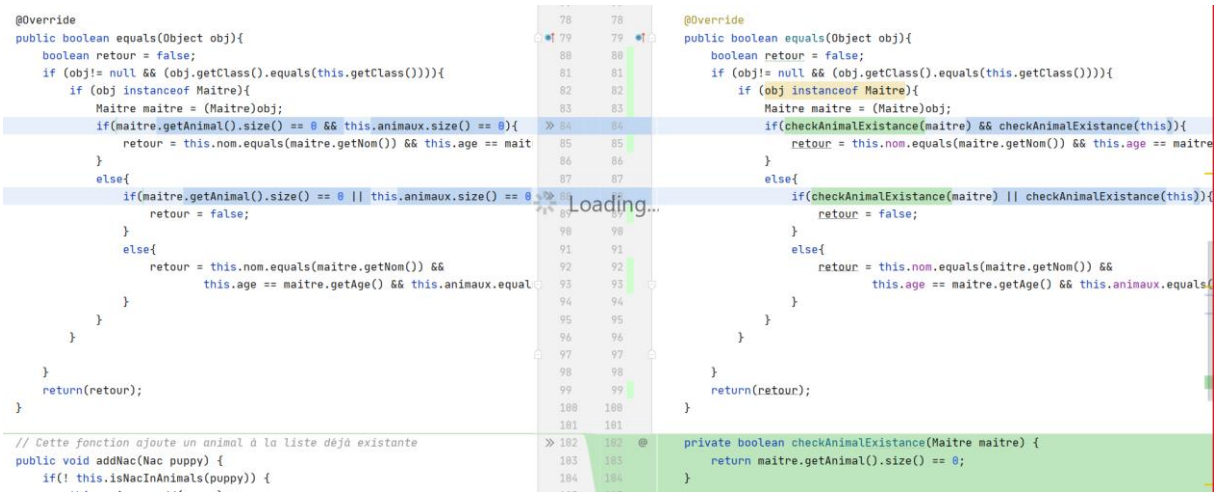



*Figure 26: Extract method*



Vous pensiez connaître toutes les facettes du refactoring ? Détrompez-vous ! Cette technique nous réserve une surprise supplémentaire: l'extractMethod.

En effet, en développant les méthodes equals, nous nous sommes rendus compte qu'il était possible d'utiliser moins de code en insérant des fonctions qui matérialisent un comportement similaire à différents endroit du code. A cet effet, la fonction "checkAnimalExistance" a permis de simplifier notre code en vérifiant la présence d'animaux appartenant à un maître.

Nous pouvons maintenant nous vanter d'optimiser notre code grâce à notre technique secrète (dont vous garderez bien sûr le secret)

Q18. Trouvez et parcourez le site officiel de JUnit. Lire l'article "TestInfected" et proposez une amélioration équivalente adaptée à votre code

```
@Override
public boolean equals(Object obj){
    boolean retour = false;
    if (obj!= null && (obj.getClass().equals(this.getClass()))){
        if (obj instanceof Maitre){
            Maitre puppy = (Maitre)obj;
            retour = this.nom.equals(puppy.getNom()) &&
                    this.age == puppy.getAge() && this.animaux.equals(puppy.getAnimal());
        }

    }
    return(retour);
}
```

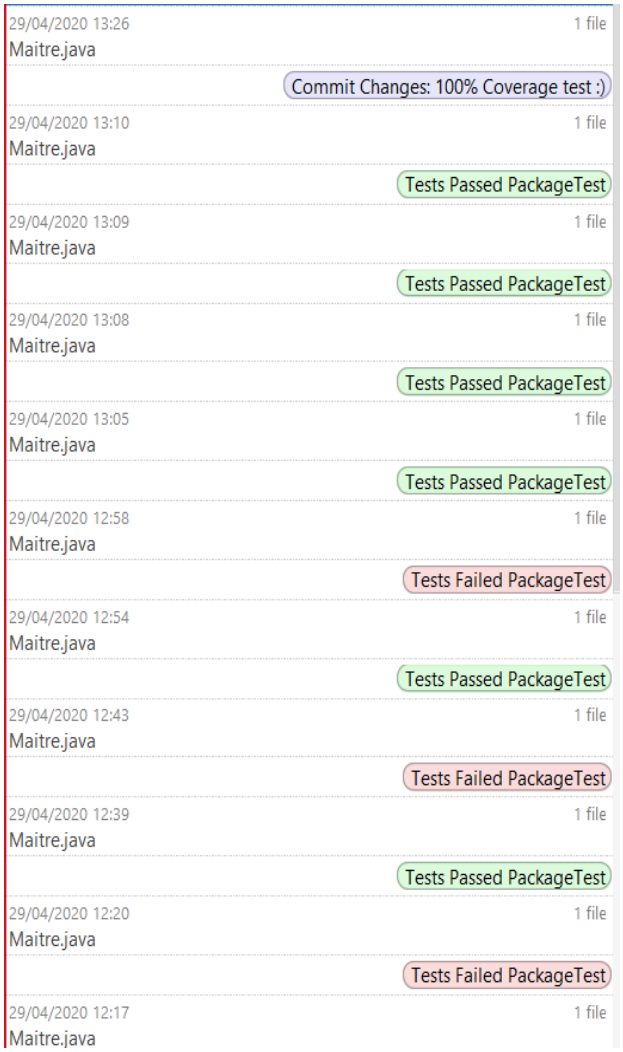
```java
@Override
public boolean equals(Object obj){
    boolean retour = false;
    if (obj!= null && (obj.getClass().equals(this.getClass()))){
        if (obj instanceof Maitre){
            Maitre maitre = (Maitre)obj;
            if(maitre.getAnimal().size() == 0 && this.animaux.size() == 0){
                retour = this.nom.equals(maitre.getNom()) && this.age == maitre.getAge();
            }
            else{
                if(maitre.getAnimal().size() == 0 || this.animaux.size() == 0){
                    retour = false;
                }
                else{
                    retour = this.nom.equals(maitre.getNom()) &&
                            this.age == maitre.getAge() && this.animaux.equals(maitre.getAnimal());
                }
            }
        }

    }
    return(retour);
}
```

Nous allons maintenant vous apprendre à voir dans le futur. Vous ne deviendrez pas pas voyant mais presque. L'idée consiste à tester votre programme jusqu'à en découvrir les failles. Vous pourrez ainsi les résoudre en enrichissant votre code afin de rendre les tests concluant. Ainsi, vous aurez toujours un pas d'avance sur votre programme grâce aux tests. Pour ce faire, voici notre histoire:

L'implémentation de nos classes a donné lieu à la réécriture de la méthode equals associée à chaque classe.

Dans la première capture, nous remarquons que la comparaison des maîtres se fait à partir de l'égalité des différents attributs de la classe. Or, nous proposons à l'utilisateur de créer un maître par le biais de trois constructeurs.

En effectuant des tests sur l'égalité entre des maîtres dont l'attribut "animaux" n'est pas défini, nous nous sommes rendus compte qu'il faudrait enrichir la fonction equals en l'adaptant aux différents constructeurs créés.

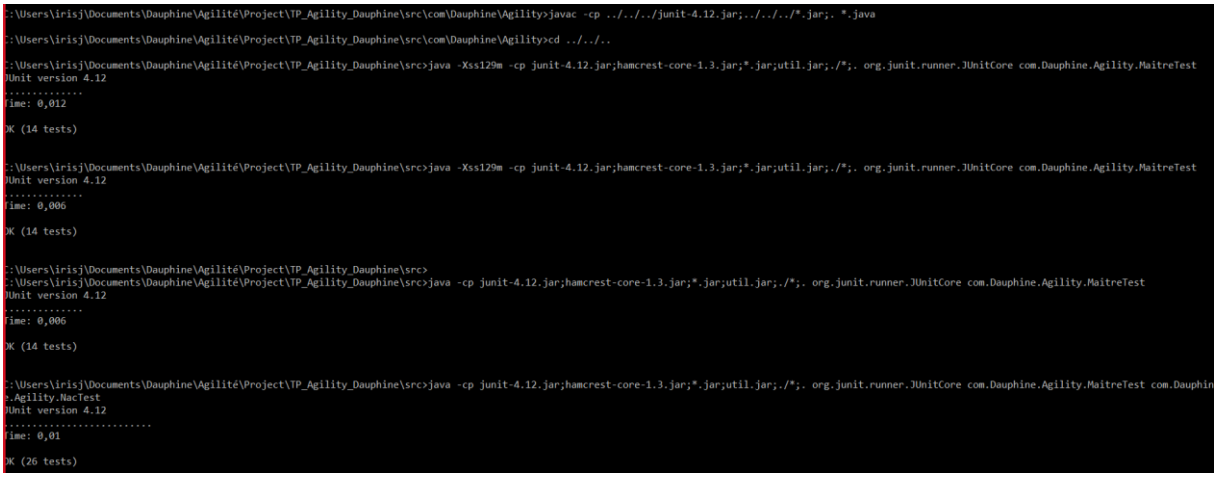



*¡ Youhou c'est noël !*

Nous notons ici l'échec puis la réussite des tests relatifs à la classe MaîtreTest expliqués par les ajouts effectués au sein de la classe equals.

Vous tomberez à nouveau  dans la réalité lorsque votre programme réussira à effectuer tous vos tests.

Q19. Exécuter les tests en ligne de commande




Après avoir testé les tests sur l'IDE, nous les avons exécutés sur l'invité de commande. ("Mais pourquoi les exécuter sur un environnement différent alors qu'ils étaient concluant ?" - Uniquement pour frimer devant la machine à café au travail). Voici (une énième fois de plus), les résultats des tests:

Nous notons la réussite des 26 tests implémentés. (Ne cherchez pas la barre verte, vous ne la trouverez pas)

Q20. Citez une loi de Murphy et associez la à une situation rencontrée lors de ce périple

La loi de Murphy : « Si cela peut mal se passer, cela arrivera » présente parfaitement le ressenti subvenu à la suite du périple menant à l'exécution des tests en ligne de commande. (bien qu'en lisant la loi, nous avons été tentés de nous arrêter après la première étape en prétextant l'échec inévitable de notre projet)

En effet, nous avons utilisé la fonction "unModifiableList" de la classe "Collections" lors du parcours de listes dans les accesseurs. En compilant puis exécutant notre code au sein de l'IDE, il s'est avéré que nos tests furent concluant. Cependant, en exécutant ce même code en ligne de commande, nous avons rencontré une exception malgré la bonne compilation du programme. ("Oh mais c'est l'invité de commande qui ne marche pas…")

Bien tenté, mais après des heures de recherches (des heures, des heures et des heures), nous avons remarqué que l'exception était dû à la méthode "unModifiableList" dont l'intitulé indiquait une source inconnue. En modifiant le code et en retournant une ArrayList, nous avons pu exécuter le code et constater la cohérence des résultats de tests entre ceux de l'IDE et ceux de l'invite de commande. (VICTOIREEE !)