3T en pratique, application au calcul de la suite de Fibonnaci, en 5 minutes

par Thierry Leriche-Dessirier (www.icauda.com)

Date de publication : 15 février 2012

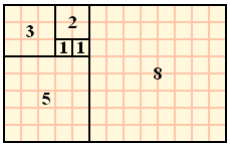
Dernière mise à jour :

Ce petit tutoriel montre comment mettre en œuvre 3T (Tests en Trois Temps), pour développer une fonctionnalité simple (la suite de Fibonnaci dans l'exemple) en s'aidant des tests, le tout en quelques minutes.

I - Introduction

3T (Tests en Trois Temps) est une déclinaison simplifiée et cadrée des TDD (Test Driven Development). Dans ce petit tutoriel, nous allons l'utiliser et voir comment cette méthodologie peut nous aider à développer une fonctionnalité.

Dans cet article, nous allons nous intéresser à la célèbre "suite de Fibonnaci" dont les premiers éléments sont "1, 1, 2, 3, 5, 8, 13..." et dont le calcul est relativement simple. Ce choix nous dispensera d'avoir à développer des algorithmes complexes qui nous détourneraient du sujet principal : les tests.



Carrés de Fibonacci en spirale

La suite de Fibonnaci, bien que simple en apparence, possède de nombreuses applications pratiques. La nature est d'ailleurs intimement liée à cette série dont elle semble vouloir s'inspirer. On connait l'exemple des pommes de pain et des tournesols. Les photographes savent en particulier que le rapport de deux membres consécutifs de la suite de Fibonnaci tend vers le fabuleux nombre d'or.

I-A - À propos

Découvrir une technologie n'est pas chose facile. En aborder plusieurs d'un coup l'est encore moins. Partant de ce constat, cet article a été écrit pour aller à l'essentiel. Les points importants sont présentés dans le corps de l'article et les éléments secondaires sont expliqués en annexe.

I-B - Avant de commencer

Pour écrire ce tutoriel, j'ai utilisé les éléments suivants :

- Java JDK 1.6.0_24-b07 ;
- Eclipse Indigo 3.7 JEE 64b ;
- Maven 3.0.3 ;
- JUnit 4.8.2.

II - Découverte du projet d'exemple

II-A - Télécharger, installer et importer le projet d'exemple

Pour commencer, je vous propose de télécharger **le fichier ZIP "fibo1.zip"**, contenant un projet Java-Maven d'exemple.

Compilez le projet d'exemple et importez-le dans Eclipse (comme expliqué dans le tutoriel **"Importer un projet Maven dans Eclipse en 5 minutes"**) ou dans l'IDE de votre choix.

Pour suivre ce tutoriel, vous pouvez vous contenter de lire les codes proposés ci-dessous (codes complets en annexe) et de vous repérer à l'aide des captures d'écran.

II-B - Ce que fait déjà le projet

Le projet est relativement vide. Il n'y a aucune classe. Seul le fichier "pom.xml" contient la configuration nécessaire pour utiliser des plugins de reporting.

Pour lancer la génération des différents rapports, il suffit d'exécuter la commande Maven suivante.

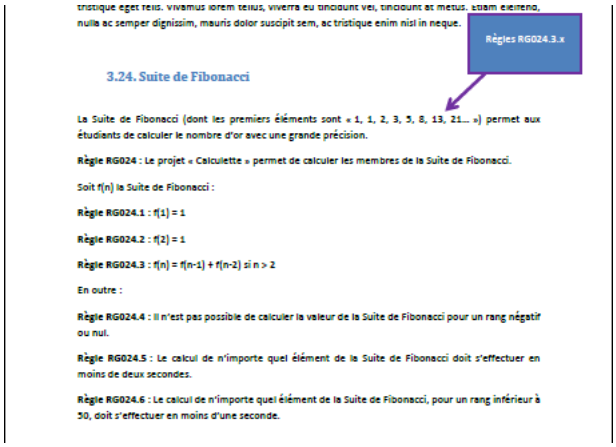
Génération du reporting

```
mvn clean install site
```

Cette commande va créer le dossier "target/site" avec en particulier le fichier "index.html" qui sert de page d'accueil pour consulter les rapports. Sans surprise, puisque le projet est initialement "vide", le reporting est relativement vide également.

III - Action

Nous allons voir comment utiliser "3T" pour développer la fonctionnalité "024", définie (page 45) dans le cahier des charges (fictif), et correspondant au calcul des éléments de la suite de Fibonnaci.


tristique eget felis. Vivamus lorem tellus, viverra eu tincidunt vel, tincidunt at metus. Etiam eleifend, nulla ac semper dignissim, mauris dolor suscipit sem, ac tristique enim nisl in neque.

Règles RG024.3.x

3.24. Suite de Fibonacci

La Suite de Fibonacci (dont les premiers éléments sont « 1, 1, 2, 3, 5, 8, 13, 21... ») permet aux étudiants de calculer le nombre d'or avec une grande précision.

Règle RG024 : Le projet « Calculette » permet de calculer les membres de la Suite de Fibonacci.

Soit f(n) la Suite de Fibonacci :

Règle RG024.1 : f(1) = 1

Règle RG024.2 : f(2) = 1

Règle RG024.3 : f(n) = f(n-1) + f(n-2) si n > 2

En outre :

Règle RG024.4 : Il n'est pas possible de calculer la valeur de la Suite de Fibonacci pour un rang négatif ou nul.

Règle RG024.5 : Le calcul de n'importe quel élément de la Suite de Fibonacci doit s'effectuer en moins de deux secondes.

Règle RG024.6 : Le calcul de n'importe quel élément de la Suite de Fibonacci, pour un rang inférieur à 50, doit s'effectuer en moins d'une seconde.

Page 45 du cahier des charges (cliquer pour voir la page en entier)

Le PDF de la page 45 du cahier des charges (fictif) est disponible dans le ZIP du projet d'exemple.

III-A - Ajout d'un élément dans l'interface

Durée estimée : 1 minute.

Dans ce tutoriel, nous allons considérer que la suite de Fibonnaci est la première qu'on développe, et qu'il n'y a donc aucun code préexistant.

On commence donc par créer l'interface "Calculette.java" dans laquelle on spécifie ce que doit faire la fonctionnalité. C'est la première étape définie par 3T.

Calculette.java

```
public interface Calculette {

    Long fibonacci(Integer n);

}
```

Conformément aux préconisations de 3T, on ajoute un peu de documentation. Pour cela, on se contente de recopier le cahier des charges.

Calculette.java

```
public interface Calculette {

    /**
     * Calcule les elements de la suite de Fibonnaci.
     *
     * REGLE RG024 Le projet permet de calculer les membres de la Suite de
     * Fibonacci.
     *
     * REGLE RG024.1 : f(1) = 1
     * REGLE RG024.2 : f(2) = 1
     *
     * REGLE RG024.3 : f(n) = f(n-1) + f(n-2) si n > 2
     *
     * REGLE RG024.4 : il n'est pas possible de calculer la valeur de la Suite
     * de Fibonacci pour un rang negatif.
     *
     * REGLE RG024.5 : le calcul de n'importe quel element de la Suite de
     * Fibonacci doit s'effectuer en moins de deux secondes.
     *
     * REGLE RG024.6 : le calcul de n'importe quel element de la Suite de
     * Fibonacci, pour un rang inferieur a 50, doit s'effectuer en moins d'une
     * seconde.
     *
     * @param n
     *            le rang pour lequel on calcule le membre.
     * @return Le membre de rang n dans la Suite.
     */
    Long fibonacci(Integer n);

}
```

Les développeurs ont identifié des règles implicites dans le cahier des charges. Ces règles ont été indiquées à l'aide du rectangle bleu, avec la référence "RG024.3.x". Puisqu'on a la chance d'avoir de nombreux exemples, on les ajoute à la documentation.

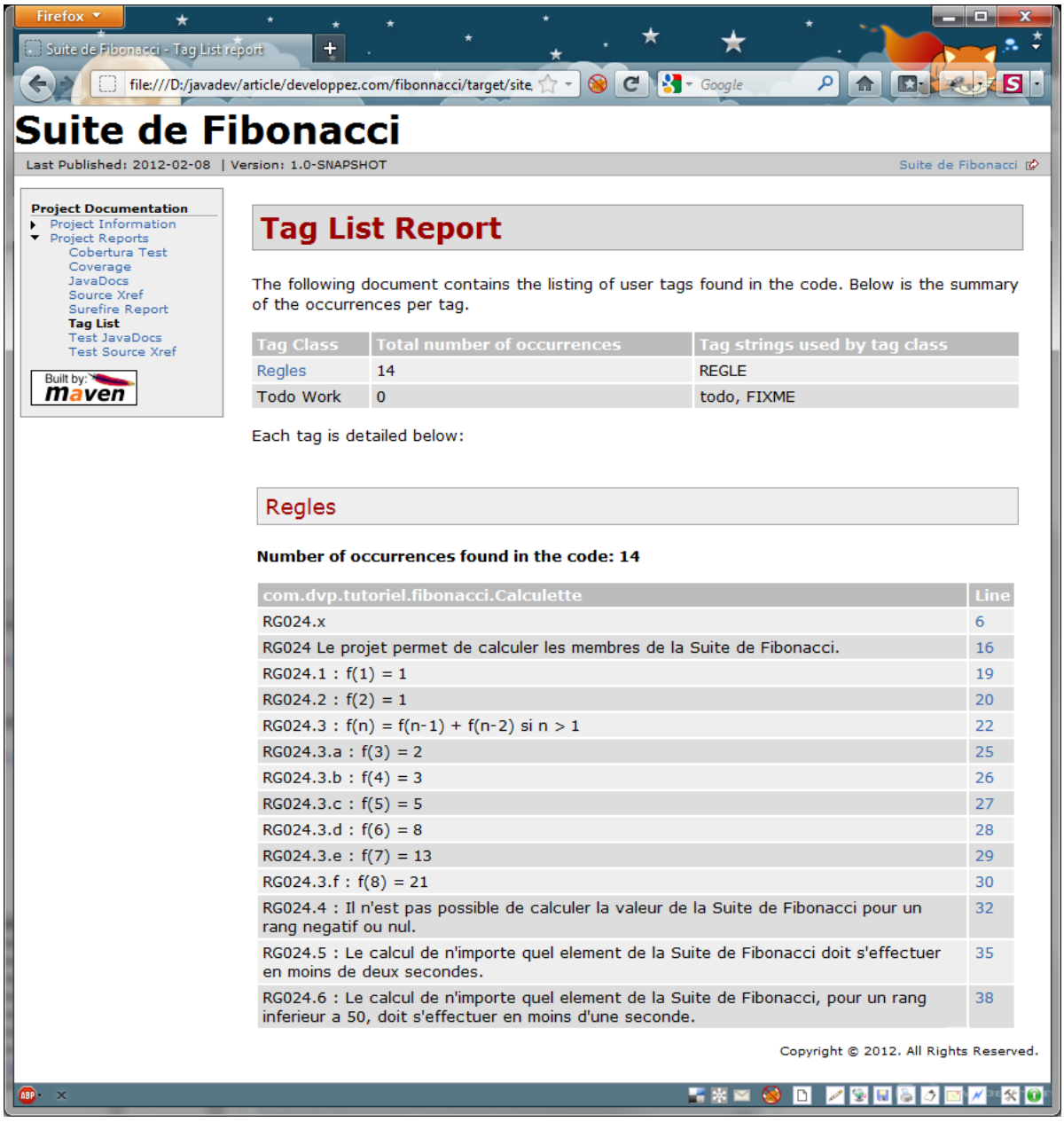
Calculette.java

```
public interface Calculette {

    /**
     * Calcule les elements de la suite de Fibonnaci.
     *
     * REGLE RG024 : le projet permet de calculer les membres de la Suite de
     * Fibonacci.
     *
     * REGLE RG024.1 : f(1) = 1
     * REGLE RG024.2 : f(2) = 1
     *
     * REGLE RG024.3 : f(n) = f(n-1) + f(n-2) si n > 2
     *
     * Exemples :
     * REGLE RG024.3.a : f(3) = 2
     * REGLE RG024.3.b : f(4) = 3
     * REGLE RG024.3.c : f(5) = 5
     * REGLE RG024.3.d : f(6) = 8
     * REGLE RG024.3.e : f(7) = 13
     * REGLE RG024.3.f : f(8) = 21
     *
     * REGLE RG024.4 : il n'est pas possible de calculer la valeur de la Suite
     * de Fibonacci pour un rang negatif.
     *
     * REGLE RG024.5 : le calcul de n'importe quel element de la Suite de
     * Fibonacci doit s'effectuer en moins de deux secondes.
     *
```

Calculette.java

```
     * REGLE RG024.6 : le calcul de n'importe quel element de la Suite de
     * Fibonacci, pour un rang inferieur a 50, doit s'effectuer en moins d'une
     * seconde.
     *
     * @param n
     *            le rang pour lequel on calcule le membre.
     * @return Le membre de rang n dans la Suite.
     */
    Long fibonacci(Integer n);

}
```

On peut dès à présent lancer la génération des rapports. Le plugin "taglist", en particulier, nous indique la liste de toutes les règles que nous allons prendre en compte. Cette information est utile pour tous les membres de l'équipe, en particulier pour le chef de projet.



Règles prises en compte.

Pour finir cette étape, on crée l'implémentation "MaCalculette" destinée à recevoir le code, en la laissant vide.

MaCalculette.java

```
public class MaCalculette implements Calculette {

    @Override
    public Long fibonacci(Integer n) {
        throw new UnsupportedOperationException("Cette fonction n'est pas encore disponible.");
    }

}
```

III-B - Ajout des tests

Durée estimée : 3 minutes.

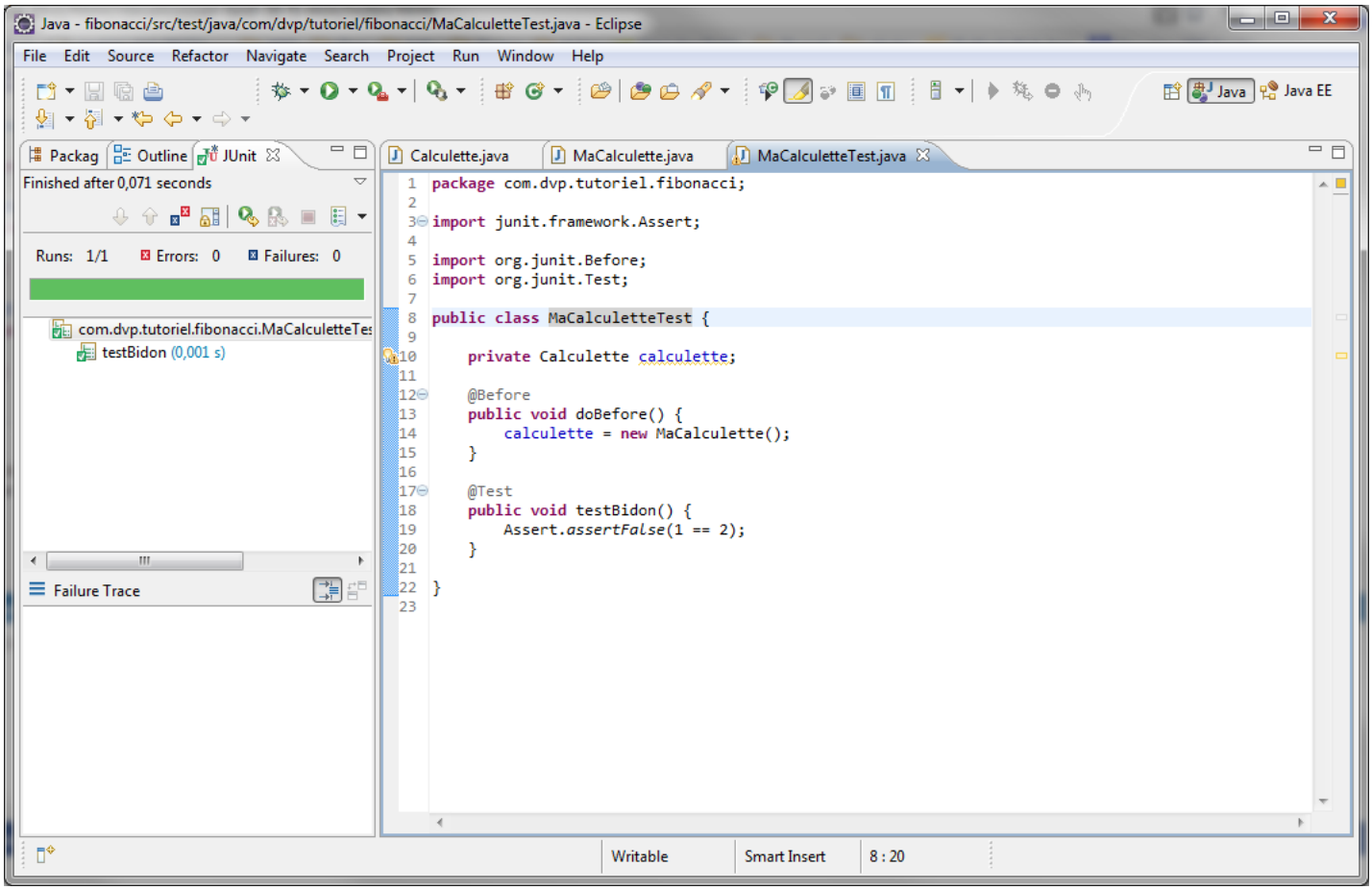
3T conseille que le développeur qui code les tests ne soit pas celui qui a déjà écrit l'interface.

On ne code pas directement la fonctionnalité. On écrit d'abord des tests. Pour cela, on s'inspire de l'interface qu'on vient de créer. C'est la deuxième étape définie par 3T.

MaCalculetteTest.java

```
public class MaCalculetteTest {

    private Calculette calculette;

    @Before
    public void doBefore() {
        calculette = new MaCalculette();
    }

    @Test
    public void testBidon() {
        Assert.assertFalse(1 == 2);
    }

}
```

Par habitude, je commence toujours par créer un test "bidon" pour vérifier que JUnit fonctionne correctement. Je supprime ce test immédiatement après avoir constaté que tout va bien, pour ne pas polluer le code.



JUnit dans Eclipse

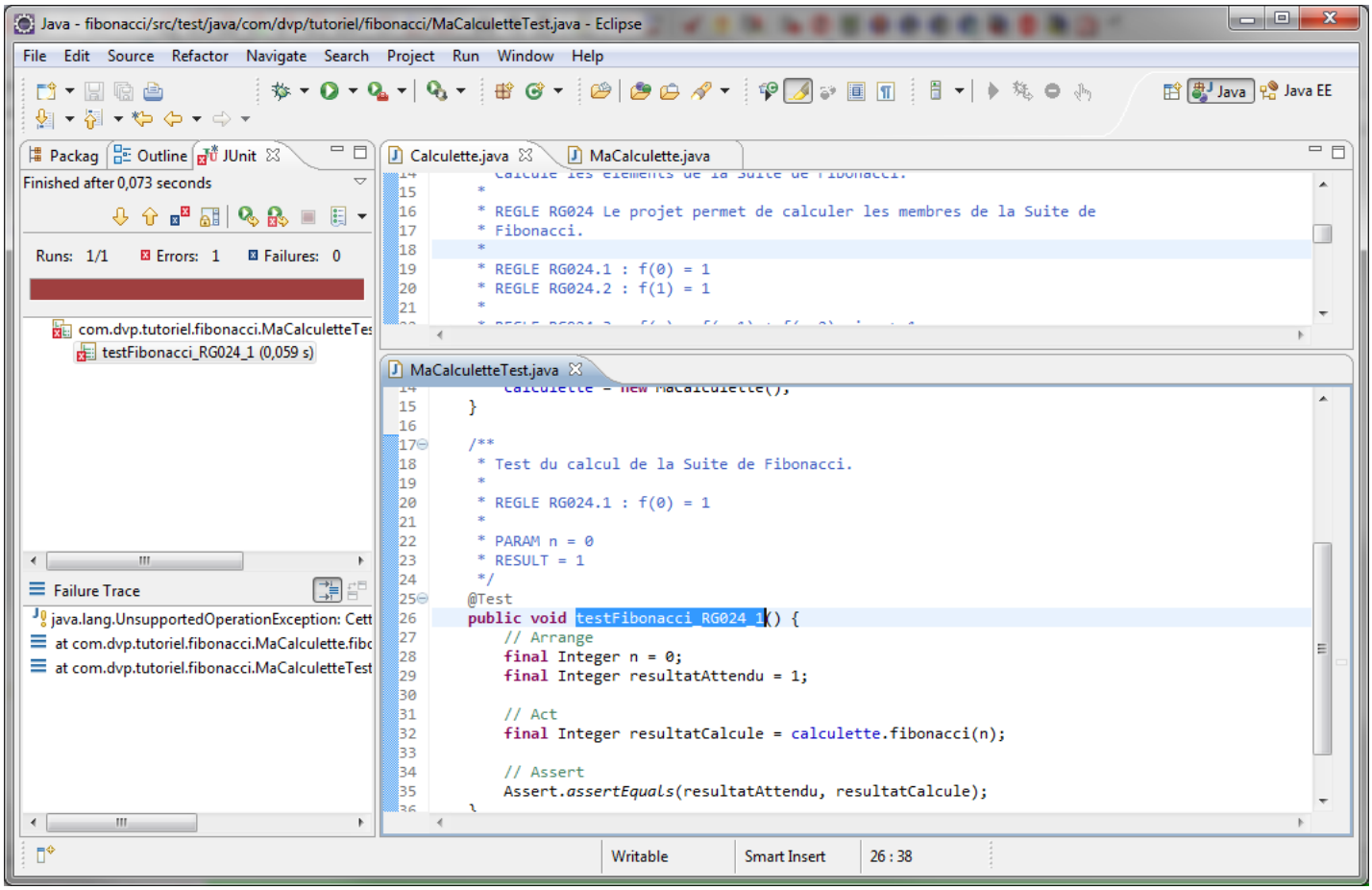
On écrit ensuite les tests. Conformément à 3T, on écrit un test (dédié) pour chaque règle.

MaCalculetteTest.java

```
/**
 * Test du calcul de la suite de Fibonnaci.

 * REGLE RG024.1 : f(1) = 1
 *
 * PARAM n = 1
 * RESULT = 1
 */
@Test
public void testFibonacci_RG024_1() {
    // Arrange
    final Integer n = 1;
    final Integer resultatAttendu = new Long(1);

    // Act
    final Long resultatCalcule = calculette.fibonacci(n);

    // Assert
    Assert.assertEquals(resultatAttendu, resultatCalcule);
}
```

Conformément à 3T, on utilise le formalisme AAA (Arrange, Act, Assert) pour organiser les tests et rendre la lecture plus simple. On écrit aussi un peu de documentation en recopiant le numéro de la règle testée et en précisant les paramètre envoyés ainsi que la réponse attendue.

Quand on lance le test, il doit être rouge dans Eclipse, ce qui est logique puisque la fonctionnalité n'est pas encore implémentée.



Le(s) test(s) est rouge

On fait de même pour l'ensemble des règles, en factorisant ce qui peut l'être.

MaCalculetteTest.java

```
public class MaCalculetteTest {

    private Calculette calculette;

    @Before
    public void doBefore() {
        calculette = new MaCalculette();
    }

    /**
     * Test du calcul de la suite de Fibonnaci.
     *
     * REGLE RG024.1 : f(1) = 1
     *
     * PARAM n = 1
     * RESULT = 1
     */
    @Test
    public void testFibonacci_RG024_1() {
        // Arrange
        final Integer n = 1;
        final long resultatAttendu = 1;

        // Act and assert
        testFibonacci_RG024_x(n, resultatAttendu);
    }

    /**
     * Test du calcul de la suite de Fibonnaci.
```

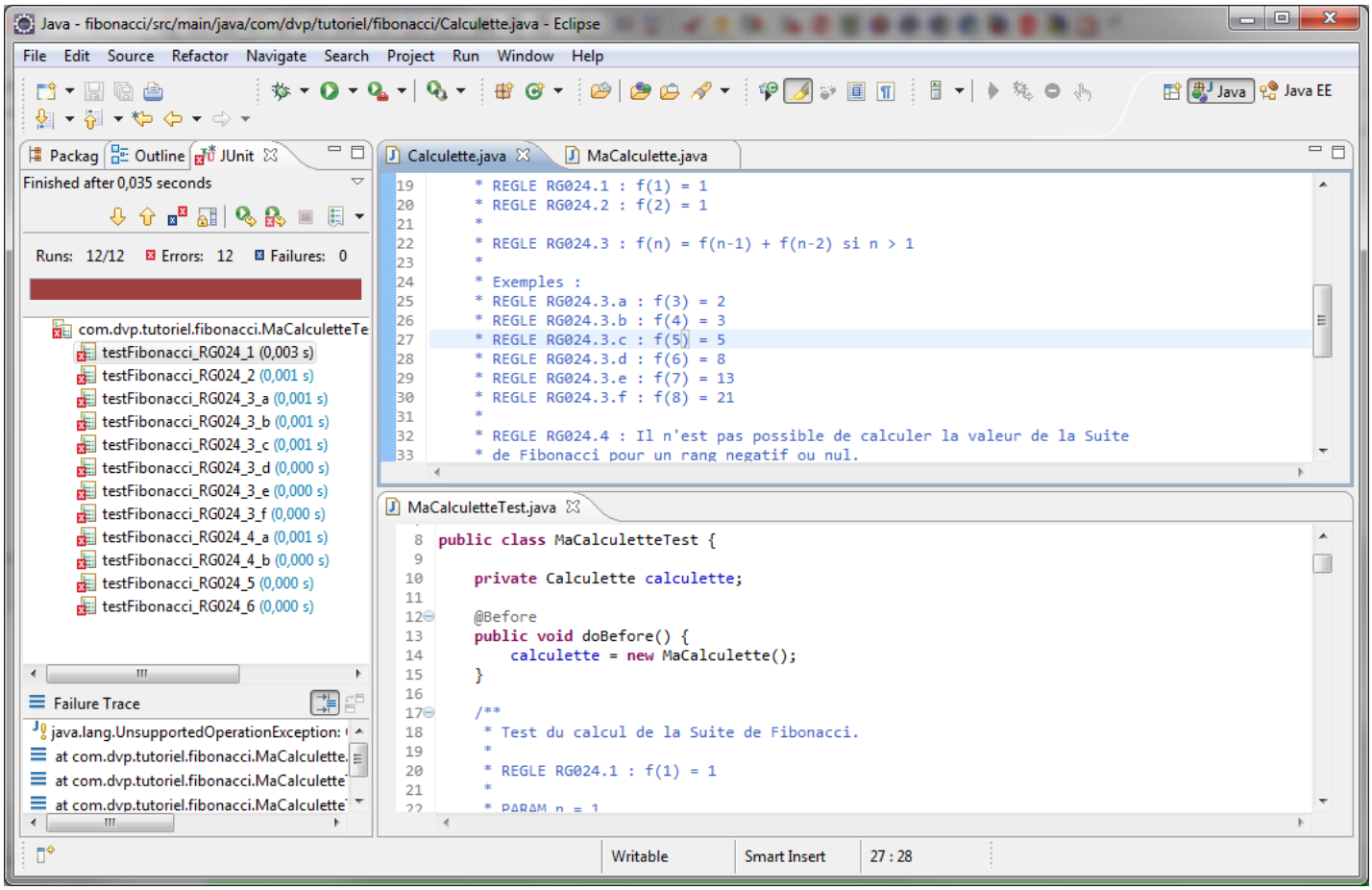
MaCalculetteTest.java

```
     *
     * REGLE RG024.2 : f(2) = 1
     *
     * PARAM n = 2
     * RESULT = 1
     */
    @Test
    public void testFibonacci_RG024_2() {
        // Arrange
        final Integer n = 2;
        final long resultatAttendu = 1;

        // Act and assert
        testFibonacci_RG024_x(n, resultatAttendu);
    }

    /**
     * Test du calcul de la suite de Fibonnaci.
     *
     * REGLE RG024.3.a : f(3) = 2
     *
     * PARAM n = 3
     * RESULT = 2
     */
    @Test
    public void testFibonacci_RG024_3_a() {
        // Arrange
        final Integer n = 3;
        final long resultatAttendu = 2;

        // Act and assert
        testFibonacci_RG024_x(n, resultatAttendu);
    }

    ...

    /**
     * Test du calcul de la suite de Fibonnaci.
     *
     * REGLE RG024.4.a : il n'est pas possible de calculer la valeur de la Suite
     * de Fibonacci pour un rang negatif ou nul.
     *
     * PARAM n = 0
     * RESULT = Exception
     */
    @Test(expected = IllegalArgumentException.class)
    public void testFibonacci_RG024_4_a() {
        // Arrange
        final Integer n = 0;

        // Act and assert
        testFibonacci_RG024_x(n, null);
    }

    ...

    /**
     * Test du calcul de la suite de Fibonnaci.
     *
     * REGLE RG024.5 : f(55) = 139583862445
     *
     * PARAM n = 55
     * RESULT = 139583862445 (en moins de 2 secondes)
     */
    @Test(timeout = 2000)
    public void testFibonacci_RG024_5() {
        // Arrange
        final Integer n = 55;
        final long resultatAttendu = 139583862445L;

        // Act and assert
```

MaCalculetteTest.java

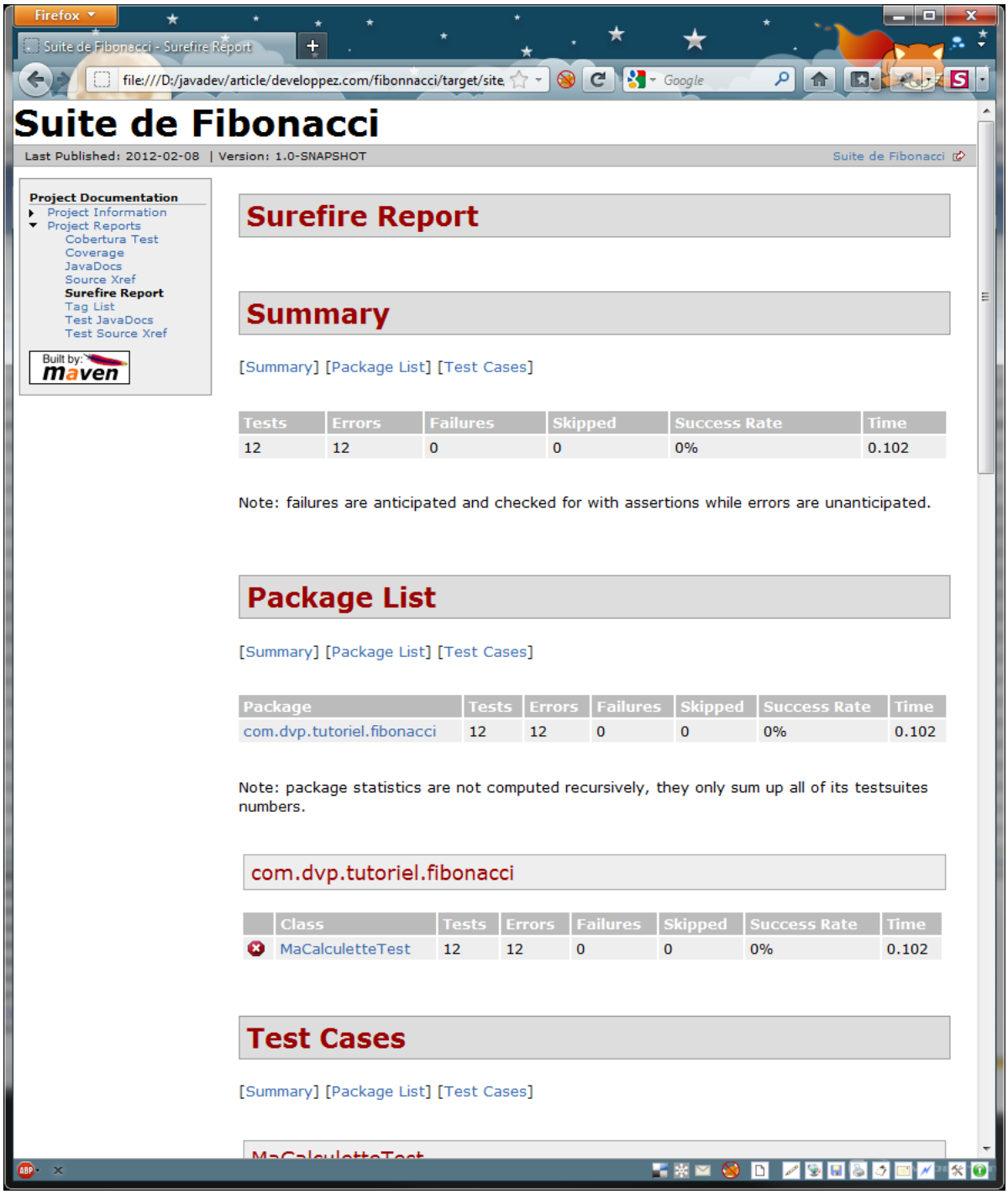
```
        testFibonacci_RG024_x(n, resultatAttendu);
    }

    ...

    private void testFibonacci_RG024_x(Integer n, Long resultatAttendu) {
        // Act
        final Long resultatCalcule = calculette.fibonacci(n);

        // Assert
        assertEquals(resultatAttendu, resultatCalcule);
    }

}
```

Conformément aux TDD, tous les tests doivent être rouges à ce stade.



Tous les tests sont rouges

On peut à nouveau demander à Maven de générer les reporting. Le rapport Surefire, en particulier, montre que tous les tests échouent.



Tous les tests sont en échec

Le chef de projet peut utiliser ce rapport pour suivre l'avancée des développements.

III-C - Développement de la fonctionnalité

Durée estimée : 2 minutes.

Maintenant que l'interface et les tests sont écrits, on peut programmer la fonctionnalité à proprement parler. C'est la troisième et dernière étape définie par 3T. Lorsque tous les tests seront verts, on pourra légitimement penser que le code est terminé.

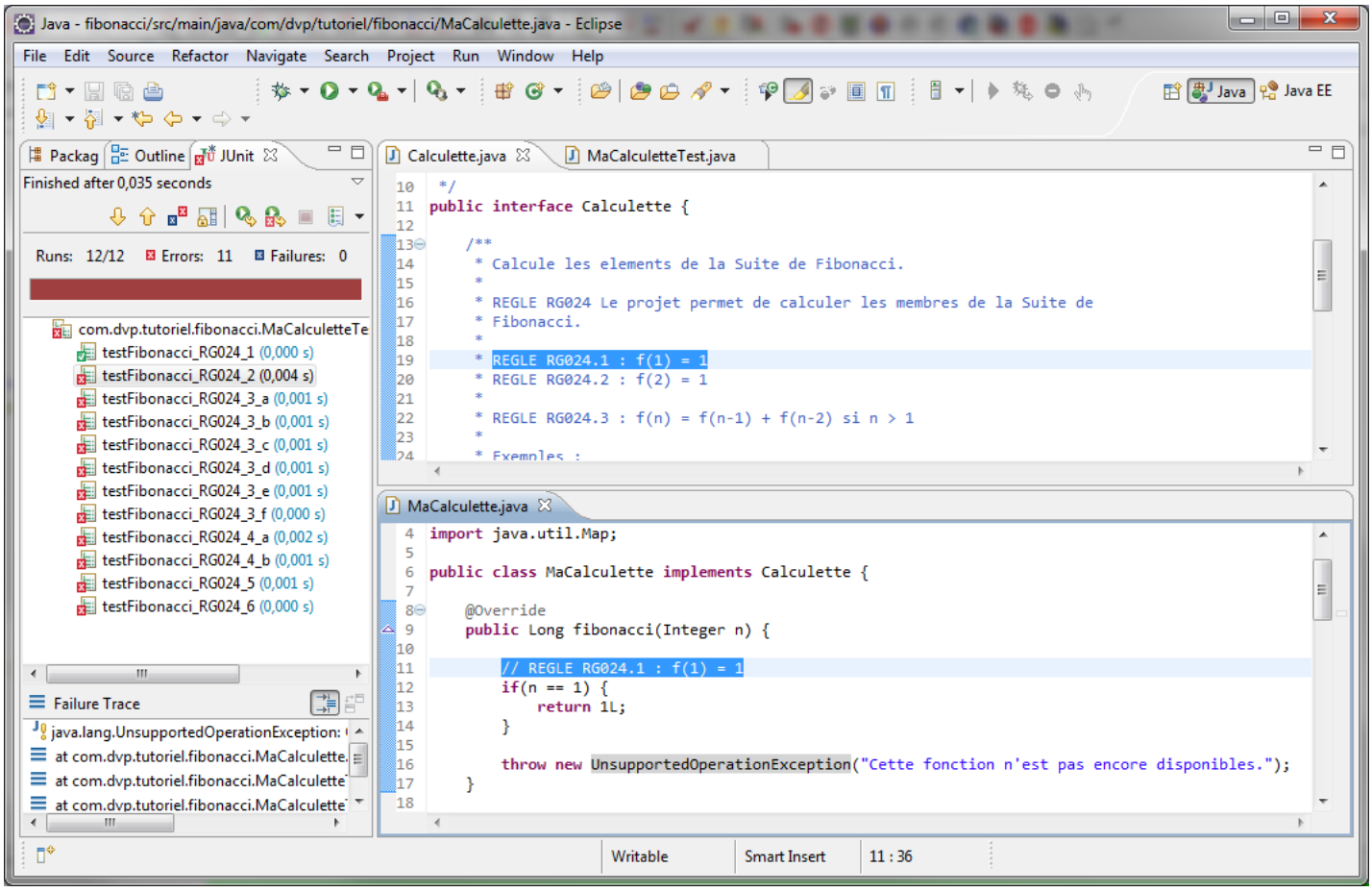
3T conseille que le développeur qui programme la fonctionnalité ne soit pas celui qui a déjà codé les tests.

On peut programmer les règles dans l'ordre qu'on veut. Faute de mieux, on peut en particulier les prendre dans l'ordre. Dans le code, on en profite pour rappeler les références des règles.

MaCalculette.java

```
public class MaCalculette implements Calculette {

    @Override
    public Long fibonacci(Integer n) {

        // REGLE RG024.1 : f(1) = 1
        if(n == 1) {
            return 1L;
        }

        throw new UnsupportedOperationException("Cette fonction n'est pas encore disponible.");
    }
```

On relance les tests et on constate que le test lié à la règle "RG024.1" est bien passé au vert.



Le test de RG024.1 est vert

On continue avec la règle suivante.

MaCalculette.java

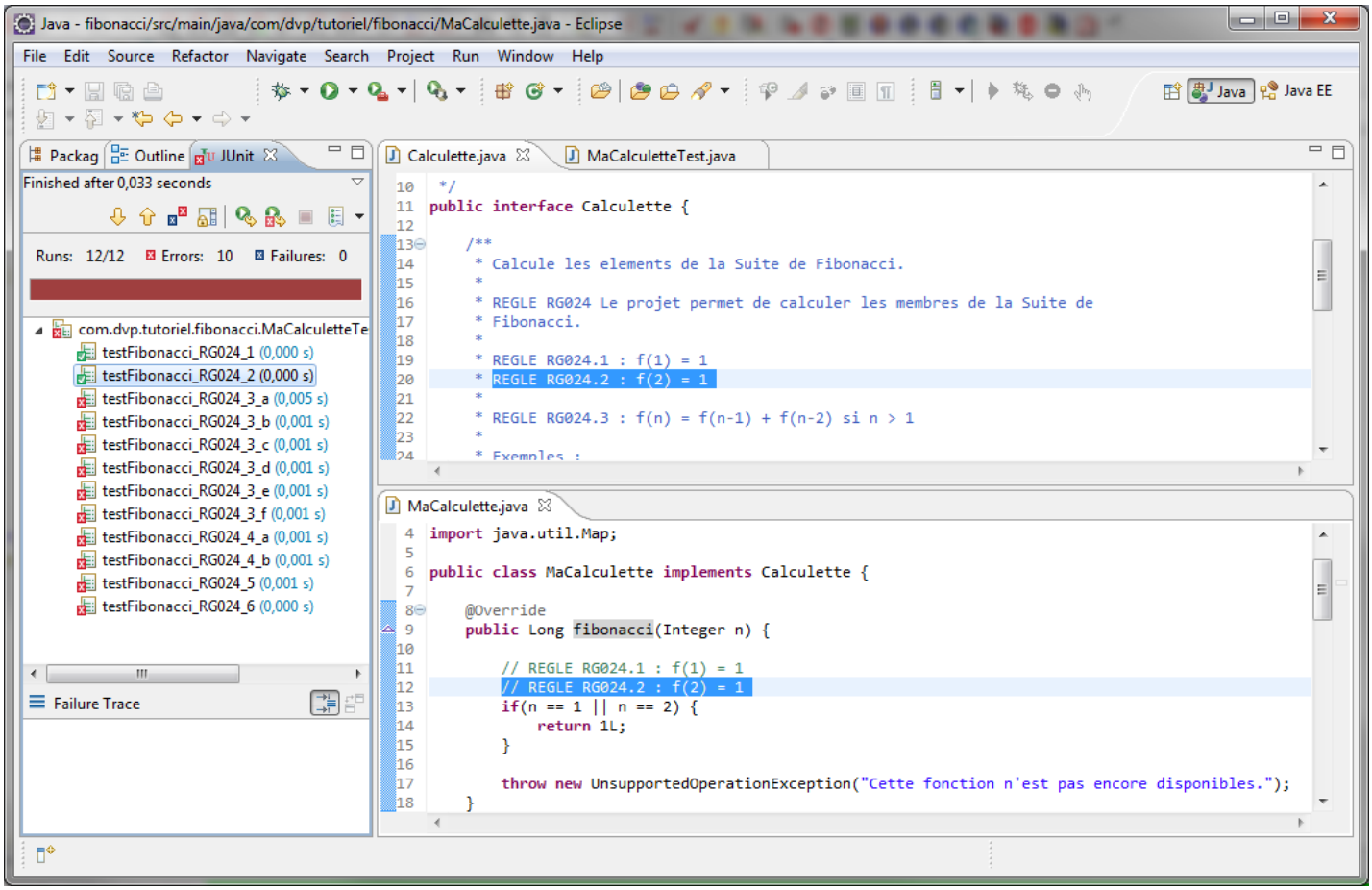
```
public class MaCalculette implements Calculette {
```

MaCalculette.java

```
    @Override
    public Long fibonacci(Integer n) {

        // REGLE RG024.1 : f(1) = 1
        // REGLE RG024.2 : f(2) = 1
        if(n == 1 || n == 2) {
            return 1L;
        }

        throw new UnsupportedOperationException("Cette fonction n'est pas encore disponible.");
    }
```

On relance les tests et on constate que le test lié à la règle "RG024.2" est bien passé au vert.

⚠ *Au passage, on vérifie que les tests déjà verts restent bien verts. Si c'est le cas, on peut continuer en mode automatique. Si, au contraire, un test précédemment vert devient rouge, cela signifie qu'on vient de créer une régression. Il faut alors poser le stylo et analyser ce qui ne va pas.*



Le test de RG024.2 est vert

On continue avec la (les) règle(s) suivante(s).

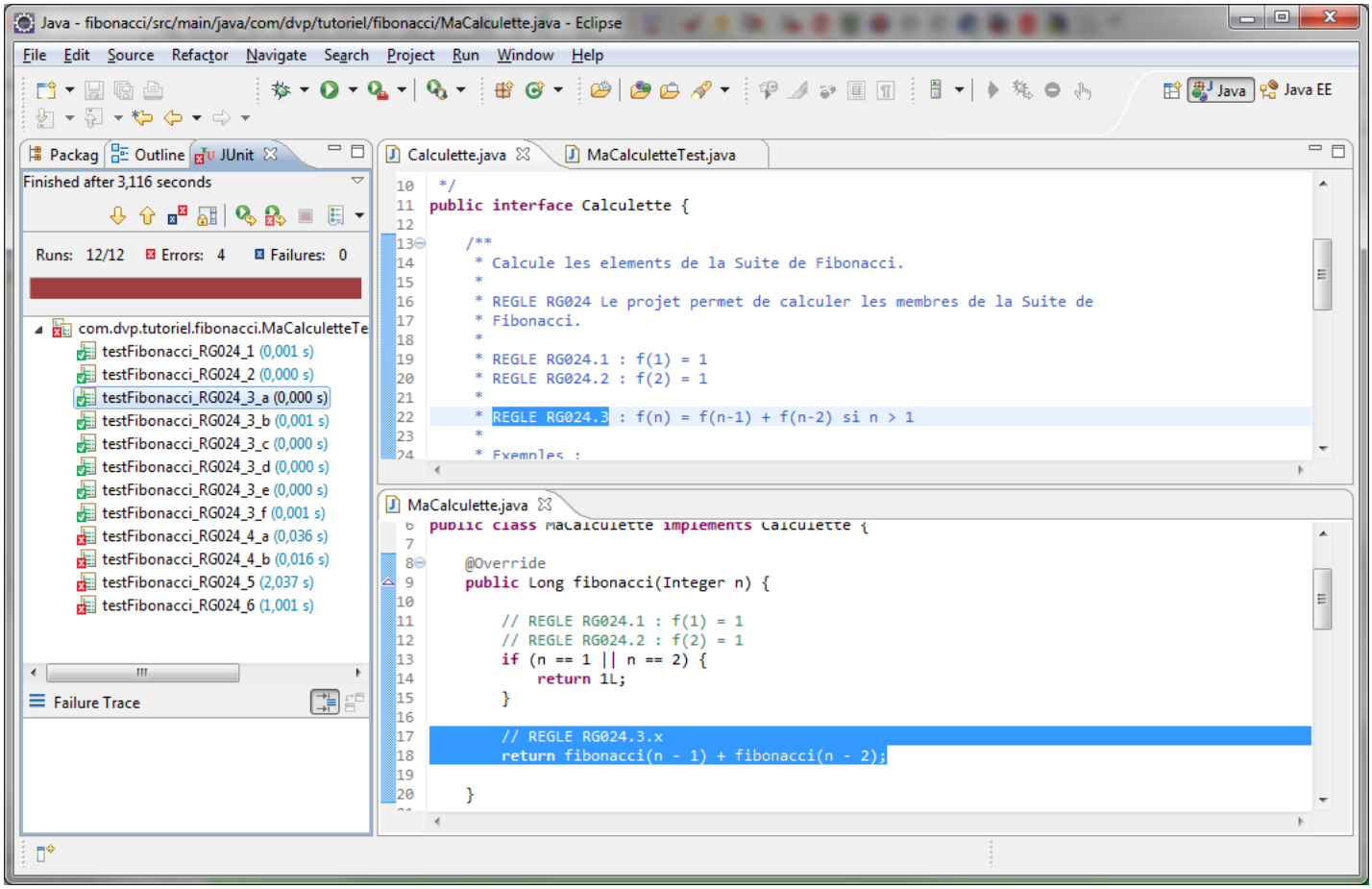
MaCalculette.java

```
public class MaCalculette implements Calculette {

    @Override
    public Long fibonacci(Integer n) {

        // REGLE RG024.1 : f(1) = 1
```

MaCalculette.java

```
        // REGLE RG024.2 : f(2) = 1
        if(n == 1 || n == 2) {
            return 1L;
        }

        // REGLE RG024.3.x
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
```

On relance les tests et on constate que de nouveaux tests sont passés au vert. On est donc sur le bon chemin.



De nouveaux tests sont verts

On continue avec la (les) règle(s) suivante(s).

MaCalculette.java

```
public class MaCalculette implements Calculette {

    @Override
    public Long fibonacci(Integer n) {

        // REGLE RG024.4
        if(n <= 0) {
            throw new IllegalArgumentException("On ne calcule que pour des nombres positifs");
        }

        // REGLE RG024.1 : f(1) = 1
        // REGLE RG024.2 : f(2) = 1
        if(n == 1 || n == 2) {
            return 1L;
        }
```

MaCalculette.java

```
        // REGLE RG024.3.x
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
```

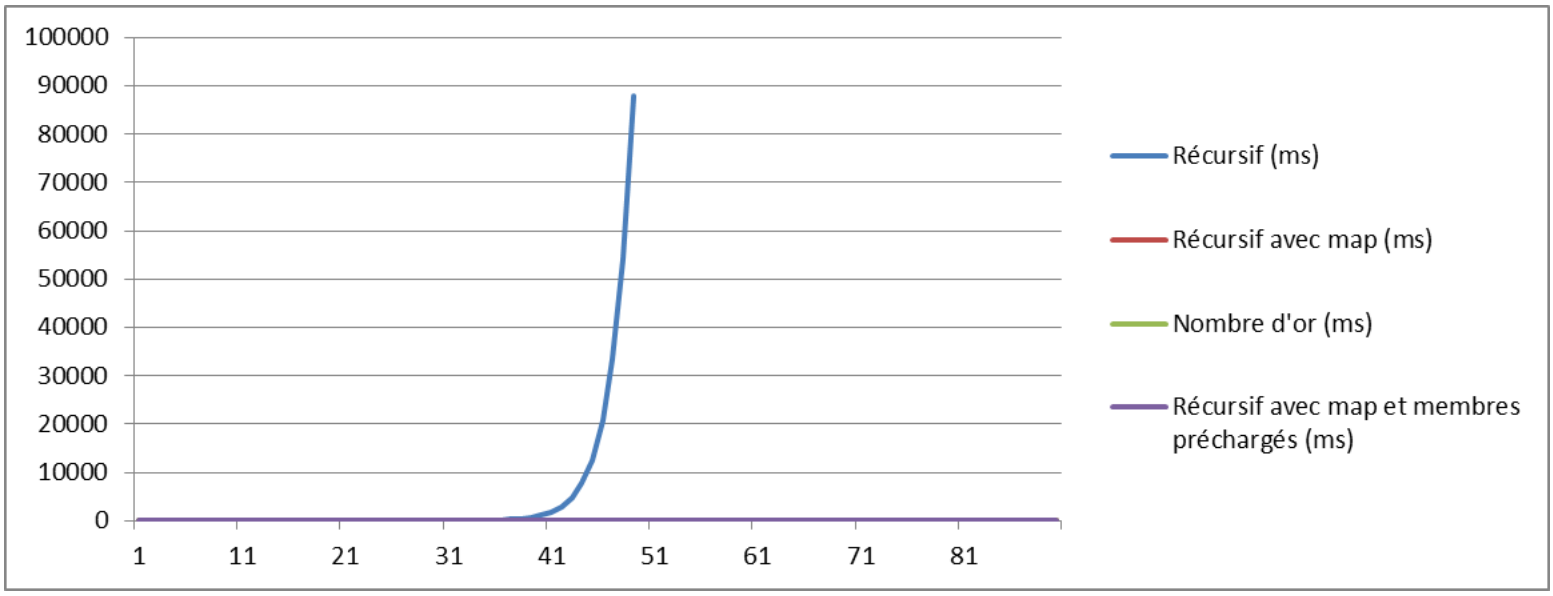
On relance les tests et on constate que de nouveaux tests sont passés au vert. On approche du but.

On continue avec les règles suivantes. Ce sont les premières qui vont demander un peu de réflexion.

MaCalculette.java

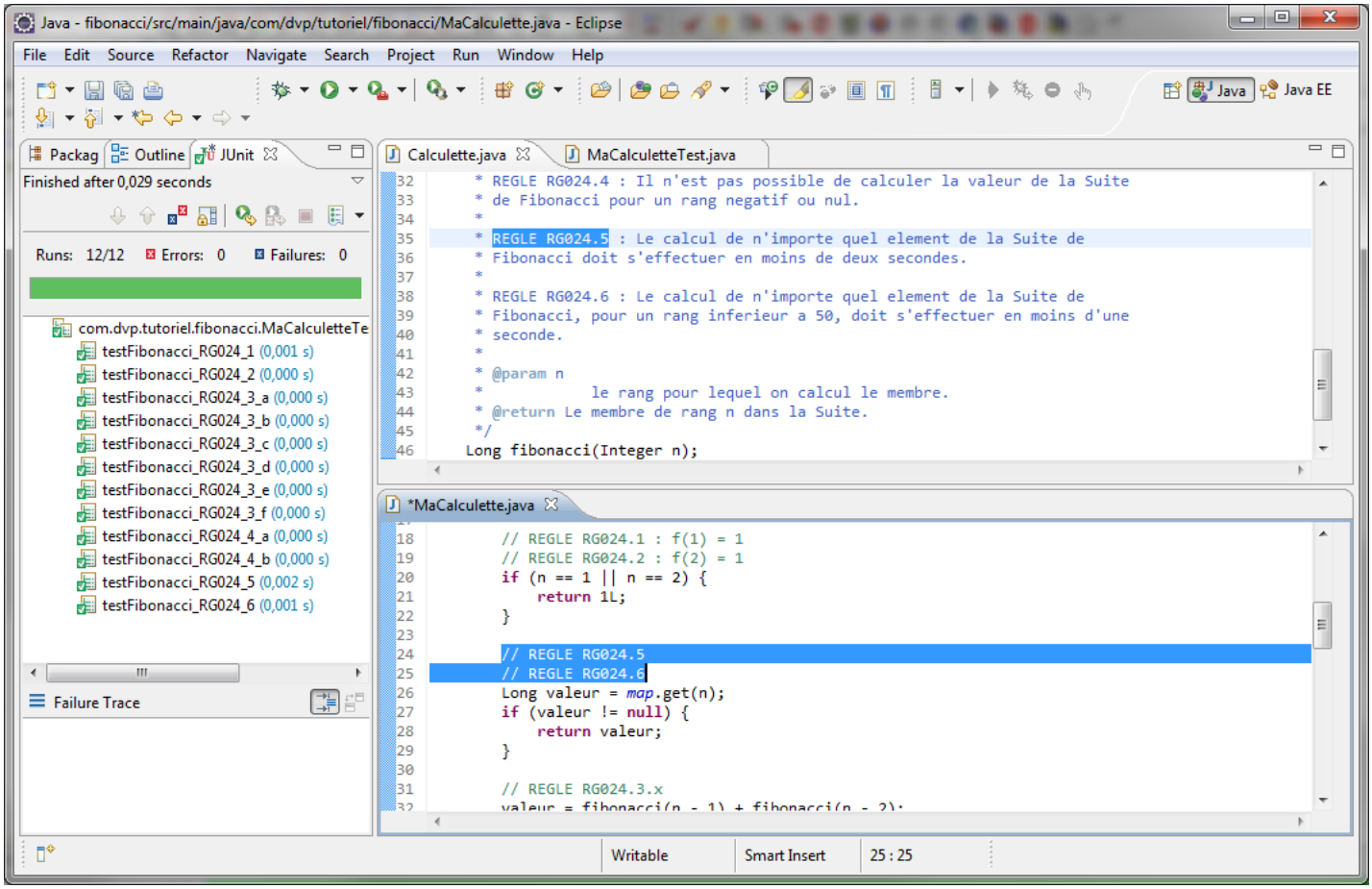
```
public class MaCalculette implements Calculette {

    private static Map<Integer, Long> map = new HashMap<Integer, Long>();

    @Override
    public Long fibonacci(Integer n) {

        // REGLE RG024.4
        if(n <= 0) {
            throw new IllegalArgumentException("On ne calcule que pour des nombres positifs");
        }

        // REGLE RG024.1 : f(1) = 1
        // REGLE RG024.2 : f(2) = 1
        if (n == 1 || n == 2) {
            return 1L;
        }

        // REGLE RG024.5
        // REGLE RG024.6
        Long valeur = map.get(n);
        if (valeur != null) {
            return valeur;
        }

        // REGLE RG024.3.x
        valeur = fibonacci(n - 1) + fibonacci(n - 2);
        map.put(n, valeur);
        return valeur;

    }
}
```

*Le fichier **fibonacci.pdf** donne une indication sur l'optimisation apportée par la map. On voit en effet que les temps de calcul augmentent de manière exponentielle dès le trentième rang environ quand on utilise un algorithme récursif simple. Grâce à l'emploi d'une map, le temps de calcul est quasi instantané (proche de 0 ms) ; il suit une progression linéaire de très faible pente qu'on pourra presque négliger.*



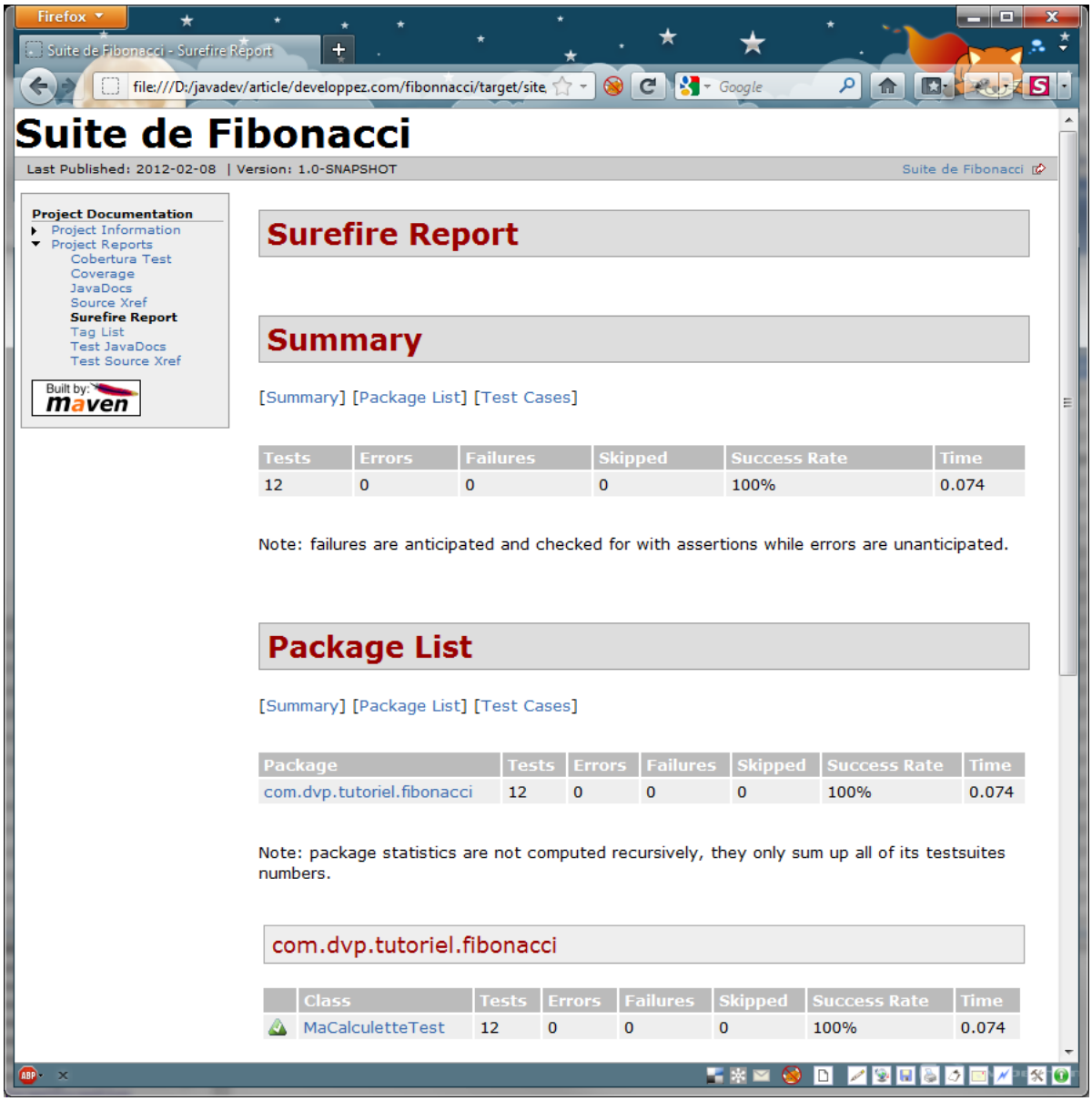
Temps de calcul en ms (voir algorithmes alternatifs en annexe)

On relance les tests et on constate que tous les tests sont désormais verts.



Tous les tests sont verts

Le rapport Surefire (JUnit), quant à lui, confirme que tous les tests passent avec succès, ce qui devrait ravir le chef de projet.



Surefire indique que tout est bon

Et voilà...

IV - Conclusions

Comme on vient de le voir dans ce petit tutoriel, il est très simple d'écrire une fonctionnalité en utilisant 3T (Tests en Trois Temps) car on est guidé tout du long. Le processus se déroule en trois étapes : écriture de l'interface, écriture des tests puis développement de la fonction. Chaque étape s'appuie largement sur la précédente. 3T conseille en outre que chaque étape soit réalisée par une personne différente.

Pour écrire l'interface, il suffit de recopier le cahier des charges. Pour coder les tests, il suffit de reprendre la documentation de l'interface et de la mettre en forme. Enfin, pour développer la fonctionnalité, il suffit de se laisser guider par les tests, comme on le ferait avec les TDD (Test Driven Development). La plupart du temps, cette démarche est quasi systématique, permettant aux développeurs de se concentrer sur l'essentiel.

Le code final de ce tutoriel est disponible dans le fichier ZIP **fibo2.zip**. Les rapports générés sont également disponibles dans le ZIP, dans le dossier "site". Le fichier **fibo3.zip**, quant à lui, contient également le code des solutions proposées en annexe, et est réorganisé pour prendre en compte plusieurs solutions.

Retrouvez les tutoriels de la série "en 5 minutes" sur Developpez.com à l'adresse **http://thierry-leriche-dessirier.developpez.com/#page_articles** .

V - Remerciements

Je tiens à remercier, en tant qu'auteur de tutoriel rapide, toutes les personnes qui m'ont aidé et soutenu. Je pense tout d'abord à mes collègues qui subissent mes questions au quotidien mais aussi à mes contacts et amis du Web, dans le domaine de l'informatique ou non, qui m'ont fait part de leurs remarques et critiques. Bien entendu, je n'oublie pas l'équipe de Developpez.com qui m'a guidé dans la rédaction de cet article et m'a aidé à le corriger et le faire évoluer, principalement sur le forum.

Plus particulièrement j'adresse mes remerciements à **Mickael BARON (**, Maxime Gault (*_Max_*) et Jacques THERY (*jacques_jean*).

VI - Annexes

VI-A - Liens

Article, en français, consacré à la suite de Fibonnaci sur Wikipédia :
http://fr.wikipedia.org/wiki/Suite_de_Fibonacci

Article "3T : les Tests en Trois Temps" sur Developpez.com :
http://thierry-leriche-dessirier.developpez.com/tutoriels/java/methode-3t/

Article "Les Tests en Trois Temps (3T) en pratique" sur Developpez.com :
http://thierry-leriche-dessirier.developpez.com/tutoriels/java/3t-en-pratique/

VI-B - Les fichiers importants en entier

Par ordre de développement...

Interface Calculette.java

```
public interface Calculette {

    /**
     * Calcule les elements de la suite de Fibonnaci.
     *
     * REGLE RG024 : le projet permet de calculer les membres de la Suite de
     * Fibonacci.
     *
     * REGLE RG024.1 : f(1) = 1
     * REGLE RG024.2 : f(2) = 1
     *
     * REGLE RG024.3 : f(n) = f(n-1) + f(n-2) si n > 1
     *
     * Exemples :
     * REGLE RG024.3.a : f(3) = 2
```

Interface Calculette.java

```
     * REGLE RG024.3.b : f(4) = 3
     * REGLE RG024.3.c : f(5) = 5
     * REGLE RG024.3.d : f(6) = 8
     * REGLE RG024.3.e : f(7) = 13
     * REGLE RG024.3.f : f(8) = 21
     *
     * REGLE RG024.4 : il n'est pas possible de calculer la valeur de la Suite
     * de Fibonacci pour un rang negatif ou nul.
     *
     * REGLE RG024.5 : le calcul de n'importe quel element de la Suite de
     * Fibonacci doit s'effectuer en moins de deux secondes.
     *
     * REGLE RG024.6 : le calcul de n'importe quel element de la Suite de
     * Fibonacci, pour un rang inferieur a 50, doit s'effectuer en moins d'une
     * seconde.
     *
     * @param n
     *            le rang pour lequel on calcule le membre.
     * @return Le membre de rang n dans la suite.
     */
    Long fibonacci(Integer n);

}
```

Tests MaCalculetteTest.java

```
import static junit.framework.Assert.assertEquals;

import org.junit.Before;
import org.junit.Test;

/**
 * Tests de la calculette "MaCalculette.java"
 *
 * @author Thierry Leriche-Dessirier
 *
 */
public class MaCalculetteTest {

    private Calculette calculette;

    @Before
    public void doBefore() {
        calculette = new MaCalculette();
    }

    /**
     * Test du calcul de la suite de Fibonnaci.
     *
     * REGLE RG024.1 : f(1) = 1
     *
     * PARAM n = 1
     * RESULT = 1
     */
    @Test
    public void testFibonacci_RG024_1() {
        // Arrange
        final Integer n = 1;
        final long resultatAttendu = 1;

        // Act and assert
        testFibonacci_RG024_x(n, resultatAttendu);
    }

    /**
     * Test du calcul de la suite de Fibonnaci.
     *
     * REGLE RG024.2 : f(2) = 1
     *
     * PARAM n = 2
```

Tests MaCalculetteTest.java

```
     * RESULT = 1
     */
    @Test
    public void testFibonacci_RG024_2() {
        // Arrange
        final Integer n = 2;
        final long resultatAttendu = 1;

        // Act and assert
        testFibonacci_RG024_x(n, resultatAttendu);
    }

    /**
     * Test du calcul de la suite de Fibonnaci.
     *
     * REGLE RG024.3.a : f(3) = 2
     *
     * PARAM n = 3
     * RESULT = 2
     */
    @Test
    public void testFibonacci_RG024_3_a() {
        // Arrange
        final Integer n = 3;
        final long resultatAttendu = 2;

        // Act and assert
        testFibonacci_RG024_x(n, resultatAttendu);
    }

    /**
     * Test du calcul de la suite de Fibonnaci.
     *
     * REGLE RG024.3.b : f(4) = 3
     *
     * PARAM n = 4
     * RESULT = 3
     */
    @Test
    public void testFibonacci_RG024_3_b() {
        // Arrange
        final Integer n = 4;
        final long resultatAttendu = 3;

        // Act and assert
        testFibonacci_RG024_x(n, resultatAttendu);
    }

    /**
     * Test du calcul de la suite de Fibonnaci.
     *
     * REGLE RG024.3.c : f(5) = 5
     *
     * PARAM n = 5
     * RESULT = 5
     */
    @Test
    public void testFibonacci_RG024_3_c() {
        // Arrange
        final Integer n = 5;
        final long resultatAttendu = 5;

        // Act and assert
        testFibonacci_RG024_x(n, resultatAttendu);
    }

    /**
     * Test du calcul de la suite de Fibonnaci.
     *
     * REGLE RG024.3.d : f(6) = 8
     *
```

Tests MaCalculetteTest.java

```
     * PARAM n = 6
     * RESULT = 8
     */
    @Test
    public void testFibonacci_RG024_3_d() {
        // Arrange
        final Integer n = 6;
        final long resultatAttendu = 8;

        // Act and assert
        testFibonacci_RG024_x(n, resultatAttendu);
    }

    /**
     * Test du calcul de la suite de Fibonnaci.
     *
     * REGLE RG024.3.e : f(7) = 13
     *
     * PARAM n = 7
     * RESULT = 13
     */
    @Test
    public void testFibonacci_RG024_3_e() {
        // Arrange
        final Integer n = 7;
        final long resultatAttendu = 13;

        // Act and assert
        testFibonacci_RG024_x(n, resultatAttendu);
    }

    /**
     * Test du calcul de la suite de Fibonnaci.
     *
     * REGLE RG024.3.f : f(8) = 21
     *
     * PARAM n = 8
     * RESULT = 21
     */
    @Test
    public void testFibonacci_RG024_3_f() {
        // Arrange
        final Integer n = 8;
        final long resultatAttendu = 21;

        // Act and assert
        testFibonacci_RG024_x(n, resultatAttendu);
    }

    /**
     * Test du calcul de la suite de Fibonnaci.
     *
     * REGLE RG024.4.a : il n'est pas possible de calculer la valeur de la Suite
     * de Fibonacci pour un rang negatif ou nul.
     *
     * PARAM n = 0
     * RESULT = Exception
     */
    @Test(expected = IllegalArgumentException.class)
    public void testFibonacci_RG024_4_a() {
        // Arrange
        final Integer n = 0;

        // Act and assert
        testFibonacci_RG024_x(n, null);
    }

    /**
     * Test du calcul de la suite de Fibonnaci.
     *
     * REGLE RG024.4.b : il n'est pas possible de calculer la valeur de la Suite
```

Tests MaCalculetteTest.java

```
     * de Fibonacci pour un rang negatif ou nul.
     *
     * PARAM n = -1
     * RESULT = Exception
     */
    @Test(expected = IllegalArgumentException.class)
    public void testFibonacci_RG024_4_b() {
        // Arrange
        final Integer n = -1;

        // Act and assert
        testFibonacci_RG024_x(n, null);
    }

    /**
     * Test du calcul de la suite de Fibonnaci.
     *
     * REGLE RG024.5 : f(55) = 139583862445
     *
     * PARAM n = 55
     * RESULT = 225851433717 (en moins de 2 secondes)
     */
    @Test(timeout = 2000)
    public void testFibonacci_RG024_5() {
        // Arrange
        final Integer n = 55;
        final long resultatAttendu = 139583862445L;

        // Act and assert
        testFibonacci_RG024_x(n, resultatAttendu);
    }

    /**
     * Test du calcul de la suite de Fibonnaci.
     *
     * REGLE RG024.6 : f(49) = 7778742049
     *
     * PARAM n = 49
     * RESULT = 7778742049 (en moins de 1 seconde)
     */
    @Test(timeout = 1000)
    public void testFibonacci_RG024_6() {
        // Arrange
        final Integer n = 49;
        final long resultatAttendu = 7778742049L;

        // Act and assert
        testFibonacci_RG024_x(n, resultatAttendu);
    }

    private void testFibonacci_RG024_x(Integer n, Long resultatAttendu) {
        // Act
        final Long resultatCalcule = calculette.fibonacci(n);

        // Assert
        assertEquals(resultatAttendu, resultatCalcule);
    }
}
```

Implémentation MaCalculette.java

```
import java.util.HashMap;
import java.util.Map;

public class MaCalculette implements Calculette {

    private static Map<Integer, Long> map = new HashMap<Integer, Long>();

    @Override
    public Long fibonacci(Integer n) {
```

Implémentation MaCalculette.java

```
        // REGLE RG024.4
        if (n <= 0) {
            throw new IllegalArgumentException("On ne calcule que pour des nombres positifs");
        }

        // REGLE RG024.1 : f(1) = 1
        // REGLE RG024.2 : f(2) = 1
        if (n == 1 || n == 2) {
            return 1L;
        }

        // REGLE RG024.5
        // REGLE RG024.6
        Long valeur = map.get(n);
        if (valeur != null) {
            return valeur;
        }

        // REGLE RG024.3.x
        valeur = fibonacci(n - 1) + fibonacci(n - 2);
        map.put(n, valeur);
        return valeur;

    }

}
```

Sans oublier le fichier "pom.xml"...

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>

 <groupId>com.dvp.tutoriel.fibonacci</groupId>
 <artifactId>fibonacci</artifactId>
 <version>1.0-SNAPSHOT</version>
 <packaging>jar</packaging>

 <name>Suite de Fibonacci</name>
 <description>Calculette tres simple pour servir d'exemple.</description>
 <url>http://www.thierryler.com</url>

 <licenses>
  <license>
   <name>Copyright ©1995-2012 thierryler.com et Copyright ©2012 Developpez.com</name>
   <comments>Les sources présentés sur cette page sont libres de droits, ...</comments>
  </license>
 </licenses>

 <developers>
  <!-- Thierry -->
  <developer>
   <name>Thierry Leriche-Dessirier</name>
   <roles>
    <role>Developper</role>
   </roles>
   <organization>ICAUDA</organization>
  </developer>

 </developers>

 <properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <java.version>1.6</java.version>
```

pom.xml

```
  <maven-compiler-plugin.version>2.3.1</maven-compiler-plugin.version>
  <junit.version>4.8.2</junit.version>
  <maven-site-plugin.version>3.0-beta-3</maven-site-plugin.version>
  <maven-javadoc-plugin.version>2.8</maven-javadoc-plugin.version>
  <maven-jxr-plugin.version>2.3</maven-jxr-plugin.version>
  <taglist-maven-plugin.version>2.4</taglist-maven-plugin.version>
  <cobertura-maven-plugin.version>2.5.1</cobertura-maven-plugin.version>
  <maven-surefire-plugin.version>2.9</maven-surefire-plugin.version>
  <maven-surefire-report-plugin>2.9</maven-surefire-report-plugin>
  <log4j.version>1.2.13</log4j.version>
 </properties>

 <dependencyManagement>
  <dependencies>
   <!-- JUnit -->
   <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>${junit.version}</version>
    <scope>test</scope>
   </dependency>

   <!-- JAX WS RT -->
   <dependency>
    <groupId>com.sun.xml.ws</groupId>
    <artifactId>jaxws-rt</artifactId>
    <version>${jaxws-rt.version}</version>
   </dependency>

   <!-- log4j -->
   <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>${log4j.version}</version>
   </dependency>
  </dependencies>
 </dependencyManagement>

 <dependencies>
  <dependency>
   <groupId>junit</groupId>
   <artifactId>junit</artifactId>
   <scope>test</scope>
  </dependency>
 </dependencies>

 <build>
  <finalName>Suite de Fibonacci</finalName>

  <plugins>
   <plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>${maven-compiler-plugin.version}</version>
    <configuration>
     <source>${java.version}</source>
     <target>${java.version}</target>
     <encoding>${project.build.sourceEncoding}</encoding>
    </configuration>
   </plugin>

   <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-site-plugin</artifactId>
    <version>${maven-site-plugin.version}</version>
   </plugin>

   <!-- Surefire -->
   <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>${maven-surefire-plugin.version}</version>
```

pom.xml

```
    </plugin>

   </plugins>
  </build>


  <reporting>
   <plugins>
    <!-- javadoc -->
    <plugin>
     <groupId>org.apache.maven.plugins</groupId>
     <artifactId>maven-javadoc-plugin</artifactId>
     <version>${maven-javadoc-plugin.version}</version>
    </plugin>

    <!-- JXR : pour lier les sources -->
    <plugin>
     <groupId>org.apache.maven.plugins</groupId>
     <artifactId>maven-jxr-plugin</artifactId>
     <version>${maven-jxr-plugin.version}</version>
    </plugin>

    <!-- taglist -->
    <plugin>
     <groupId>org.codehaus.mojo</groupId>
     <artifactId>taglist-maven-plugin</artifactId>
     <version>${taglist-maven-plugin.version}</version>
     <configuration>
      <tagListOptions>
       <tagClasses>
        <tagClass>
         <displayName>Todo Work</displayName>
         <tags>
          <tag>
           <matchString>todo</matchString>
           <matchType>ignoreCase</matchType>
          </tag>
          <tag>
           <matchString>FIXME</matchString>
           <matchType>exact</matchType>
          </tag>
         </tags>
        </tagClass>
        <tagClass>
         <displayName>Regles</displayName>
         <tags>
          <tag>
           <matchString>REGLE</matchString>
           <matchType>ignoreCase</matchType>
          </tag>
         </tags>
        </tagClass>
       </tagClasses>
      </tagListOptions>
     </configuration>
    </plugin>

    <!-- Cobertura : couverture de code -->
    <plugin>
     <groupId>org.codehaus.mojo</groupId>
     <artifactId>cobertura-maven-plugin</artifactId>
     <version>${cobertura-maven-plugin.version}</version>
    </plugin>

    <!-- Surefire (junit) -->
    <plugin>
     <groupId>org.apache.maven.plugins</groupId>
     <artifactId>maven-surefire-report-plugin</artifactId>
     <version>${maven-surefire-report-plugin}</version>
    </plugin>
```

pom.xml
```
   </plugins>
  </reporting>

</project>
}
```

VI-C - Calculs alternatifs

VI-C-1 - Calcul direct

Le calcul récursif est rarement le plus efficace. À la place, je propose la méthode de calcul suivante, faisant largement appel à mes vieux souvenirs de math et, accessoirement, à Wikipédia.

Implémentation MaCalculette2.java
```
public class MaCalculette2 implements Calculette {

    private final static double NOMBRE_OR = 1.61803398874989;
    private final static  double RACINE_5 = 2.236067977499;

    @Override
    public Long fibonacci(Integer n) {
        // REGLE RG024.4
        if(n <= 0) {
            throw new IllegalArgumentException("On ne calcule que pour des nombres positifs");
        }

        // REGLE RG024.1 : f(1) = 1
        // REGLE RG024.2 : f(2) = 1
        if (n == 1 || n == 2) {
            return 1L;
        }

        final double nominateur = Math.pow(NOMBRE_OR, n);

        final double result = nominateur / RACINE_5;
        return Math.round(result);
    }
```

Cette méthode, bien que très rapide, ne fonctionne plus au-delà d'un certain rang. Cela provient des approximations numériques utilisées. Toutefois, selon le besoin (avoir rapidement une valeur ou avoir un valeur exacte), elle peut s'avérer suffisante...

Bien entendu, il faut ajouter quelques tests pour valider cette version. On en profite pour réorganiser un peu le code.

Test (abstract) AbstractMaCalculetteTest.java
```
public abstract class AbstractMaCalculetteTest {

    protected static Calculette calculette;


    /**
     * Test du calcul de la suite de Fibonnaci.
     *
     * REGLE RG024.1 : f(1) = 1
     *
     * PARAM n = 1
     * RESULT = 1
     */
    @Test
    public void testFibonacci_RG024_1() {
        // Arrange
```

Test (abstract) AbstractMaCalculetteTest.java

```
        final Integer n = 1;
        final long resultatAttendu = 1;

        // Act and assert
        testFibonacci_RG024_x(n, resultatAttendu);
    }

    ...
    // le reste est juste une copie de MaCalculetteTest.java, telle qu'on l'avait developpee plus haut.
```

On adapte "MaCalculetteTest.java" en fonction et on crée "MaCalculette2Test.java" dans la foulée.

Test MaCalculetteTest.java remanié

```
public class MaCalculetteTest extends AbstractMaCalculetteTest {

    @Before
    public void doBefore() {
        calculette = new MaCalculette();
    }

}
```

Test MaCalculette2Test.java

```
public class MaCalculette2Test extends AbstractMaCalculetteTest {

    @Before
    public void doBefore() {
        calculette = new MaCalculette2();
    }

    /**
     * Test du calcul de la suite de Fibonnaci.
     *
     * REGLE RG024.3.divergence : f(70) = 190392490709135
     *
     * PARAM n = 70
     * RESULT = 190392490709135
     */
    @Test
    public void testFibonacci_RG024_3_divergeance() {
        // Arrange
        final Integer n = 70;
        final long resultatAttendu = 190392490709135L;

        // Act and assert
        testFibonacci_RG024_x(n, resultatAttendu);
    }

}
```

Le test "testFibonacci_RG024_3_divergence" échoue, montrant que la précision des valeurs utilisées n'est plus suffisante dès le rang "70".

*Le fichier **fibonacci.pdf** résume les erreurs de calcul, dues aux approximations. Toutefois, l'erreur commise reste toujours proche de 0 %...*

VI-C-2 - La solution de l'étudiant

Quelques-uns de mes étudiants prennent un raccourci qui n'est pas dénué de sens, même s'il est en général emprunté pour de mauvaises raisons.

Implémentation MaCalculette3.java

```
public class MaCalculette3 implements Calculette {

    @Override
    public Long fibonacci(Integer n) {

        // REGLE RG024.4
        if (n <= 0) {
            throw new IllegalArgumentException("On ne calcule que pour des nombres positifs");
        }

        // REGLE RG024.1
        // REGLE RG024.2
        // REGLE RG024.3
        switch (n) {
            case 1:
            case 2:
                return 1L;

            case 3:    return 2L;
            case 4:    return 3L;
            case 5:    return 5L;
            case 6:    return 8L;
            case 7:    return 13L;
            case 8:    return 21L;

            default:
                return fibonacci(n - 1) + fibonacci(n - 2);
        }
    }
}
```

Ce n'est pas génial mais c'est hyper rapide pour les éléments pris en compte. Comme les professeurs ne vérifient généralement que les premières valeurs, les élèves s'en tirent à bon compte, et avec de très bonnes performances. Ce code échoue néanmoins lors du test des règles "RG024.5" et "RG024.6" avec des valeurs de "n" plus importantes.

On peut toutefois s'en inspirer en combinant des valeurs précalculées et stockées dans un fichier CSV, par exemple, avec une map comme proposé plus haut.

Implémentation MaCalculette4.java

```
public class MaCalculette4 implements Calculette {

    private static Map<Integer, Long> map;

    // Singleton (initialisation directe)
    private static MaCalculette4 instance = new MaCalculette4();

    private MaCalculette4 () {
        FibonacciDao dao = new FibonacciDao();
        map = dao.charger();
    }

    public static synchronized MaCalculette4 getInstance() {
        return instance;
    }

    @Override
    public Long fibonacci(Integer n) {

        // REGLE RG024.4
        if (n <= 0) {
            throw new IllegalArgumentException("On ne calcule que pour des nombres positifs");
        }

        // REGLE RG024.1 : f(1) = 1
        // REGLE RG024.2 : f(2) = 1
```

Implémentation MaCalculette4.java

```
        if (n == 1 || n == 2) {
            return 1L;
        }

        // REGLE RG024.5
        // REGLE RG024.6
        Long valeur = map.get(n);
        if (valeur != null) {
            return valeur;
        }

        // REGLE RG024.3.x
        valeur = fibonacci(n - 1) + fibonacci(n - 2);
        map.put(n, valeur);
        return valeur;
    }
}
```

Le fichier "fibonacci.csv" ressemblerait alors au code suivant.

fibonacci.csv

```
# suite de Fibonnaci

Rang,membre

1,1
2,1
3,2
4,3
5,5
6,8
7,13
8,21
9,34
10,55
11,89
12,144
13,233
14,377
15,610
16,987
17,1597
18,2584
19,4181
20,6765
21,10946
22,17711
23,28657
24,46368
25,75025
26,121393
27,196418
28,317811
29,514229
30,832040
...
```

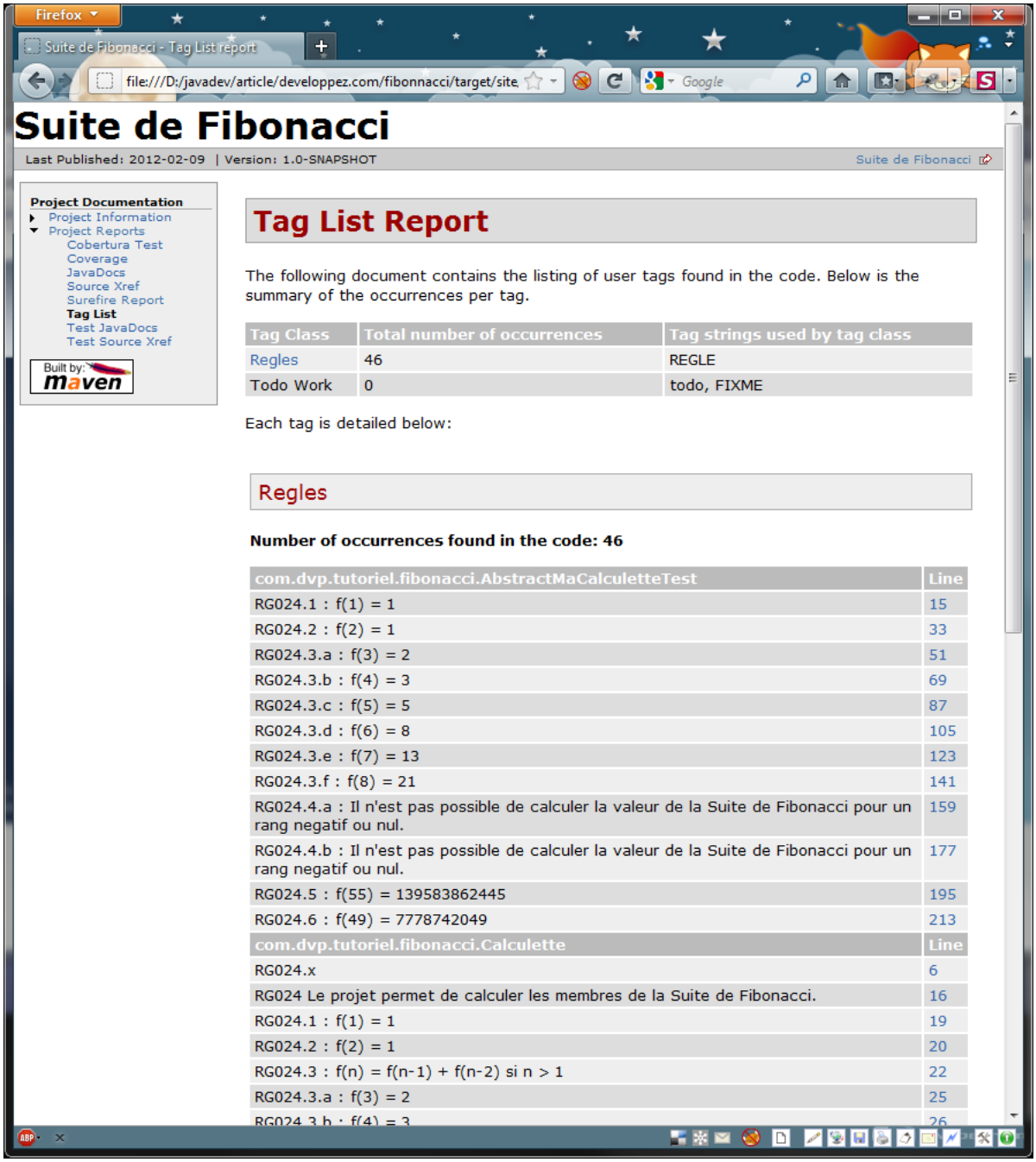
*Je vous invite à lire le tutoriel "Charger des données depuis un fichier CSV simple en 5 minutes", disponible à l'adresse **http://thierry-leriche-dessirier.developpez.com/tutoriels/java/charger-donnees-fichier-csv-5-min** pour savoir comment lire un fichier CSV à moindre coût.*

Je crée, bien entendu, une classe de test pour vérifier que tout est bon, en suivant les recommandations de 3T.

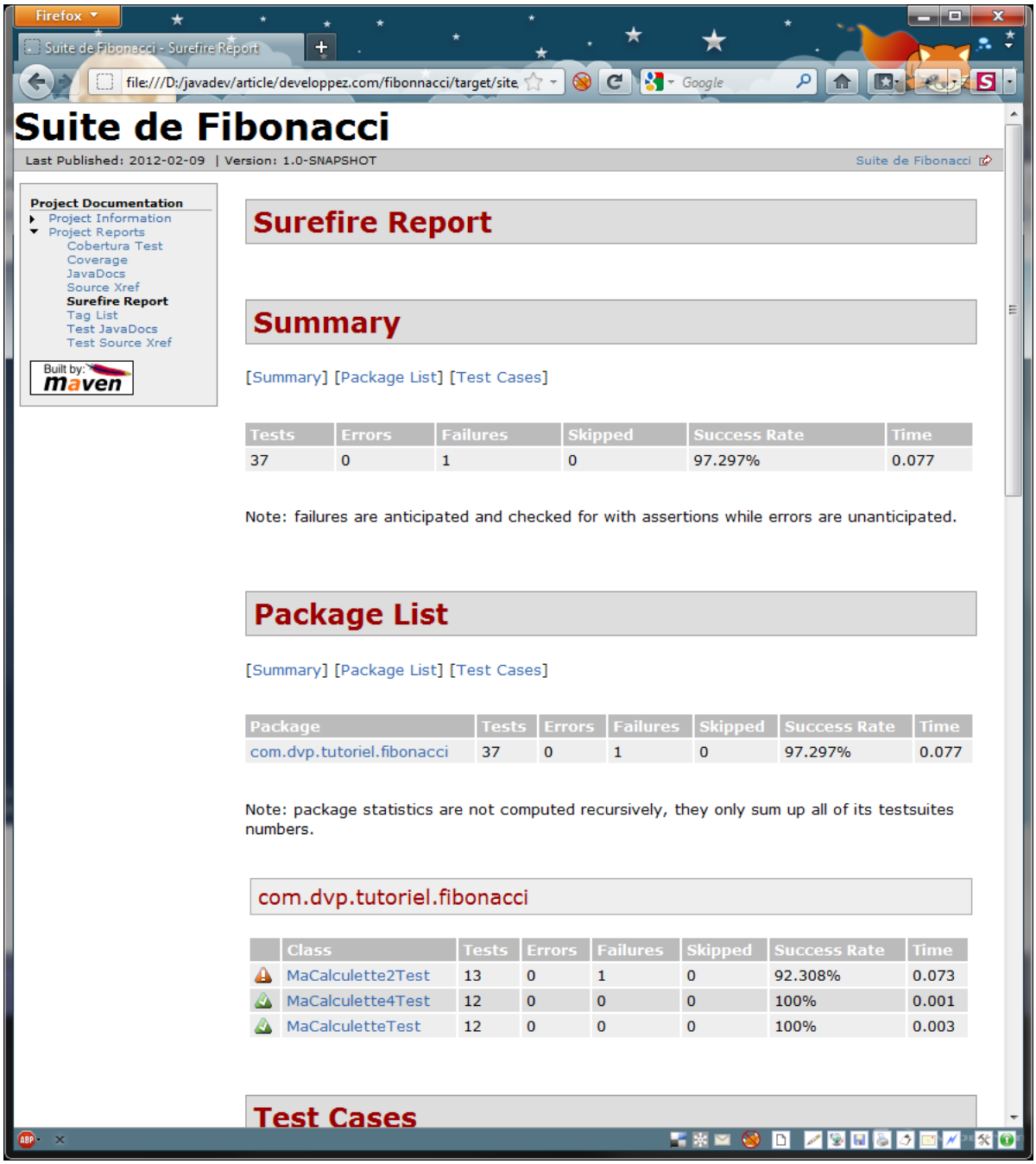
Test MaCalculette4Test.java

```
public class MaCalculetteTest4 extends AbstractMaCalculetteTest {

    @BeforeClass
    public static void doBeforeClass() {
        calculette = MaCalculette4.getInstance();
    }

}
```

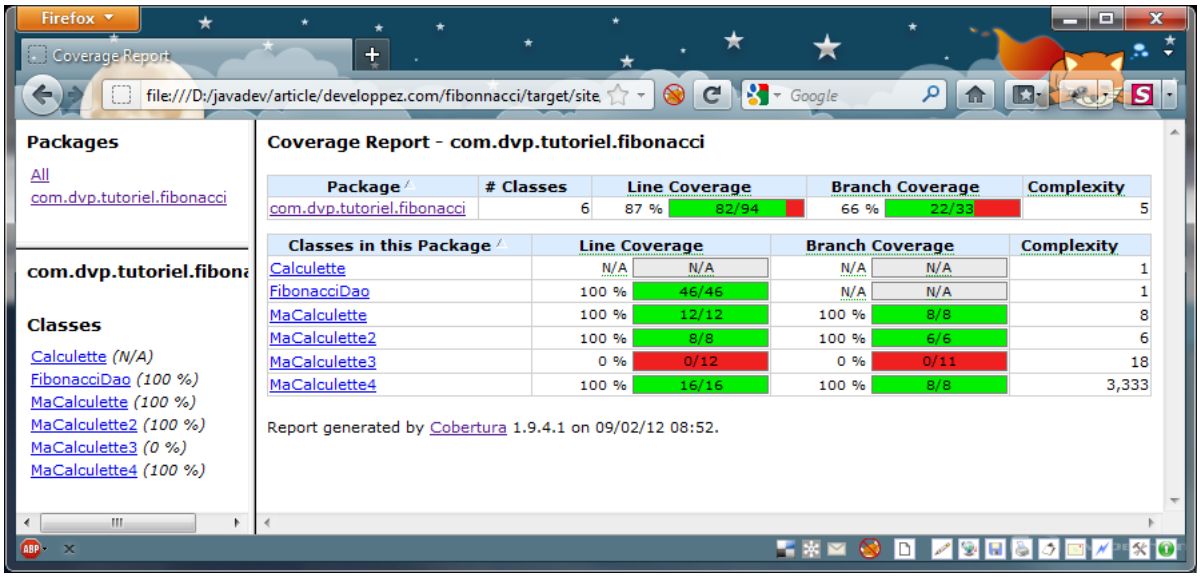
VI-D - Rapports finaux



Liste des règles (taglist) dans le code



Rapport Surefire



Couverture de test avec Cobertura

VI-E - Pour s'amuser

Pour aller plus loin et/ou s'entrainer, on peut aussi développer, sur le même modèle (3T), les suites de Lucas et les suites de k-bonacci.