



Illustration TDD

V. Deslandres, IUT de LYON
Module CVDA s2 – Mars 2017

FizzBuzz : énoncé

- Ecrire une classe qui affiche l'entier donné en paramètre, sauf :
 - **Fizz** qd c'est un multiple de 3,
 - **Buzz** quand c'est un multiple de 5
 - et **FizzBuzz** qd c'est un multiple de 3 et de 5.

Merci à Nadia Humbert et sa chaîne Crafties

- <https://www.youtube.com/watch?v=RWYvBNX9wcU>

Premier Test (JUnit 4)



On va écrire un test simple qui échoue, puis écrire ensuite le code :

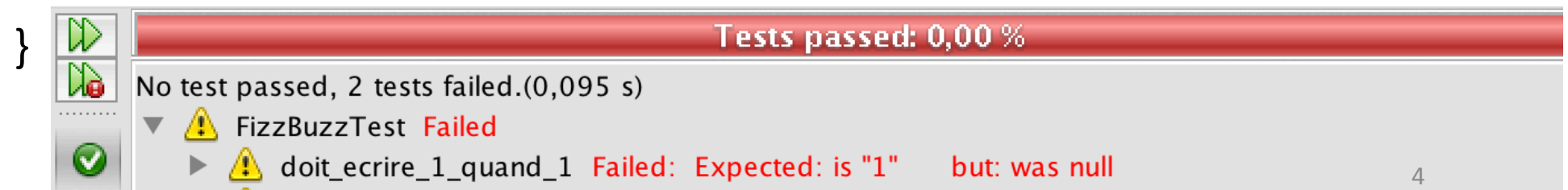
```
public class FizzBuzzTest {  
    public FizzBuzzTest() {  
    }  
    @Test                                // annotation JUnit4, utile au compilateur  
    public void doit_ecrire_1_quand_1() { // nom du test : explicite  
        // given  
        FizzBuzz fb = new FizzBuzz(1);  
        // when  
        String result = fb.afficher();  
        // then  
        assertEquals(ns, result); // JUnit: résultat attendu, résultat de l'appel  
        //assertThat(result, is("1")); // matching avec package harmCrest  
    }  
}
```

NOTA : ordre des paramètres avec HarmCrest = pour faciliter la lecture
(« result is ... », c'est l'inverse d'assertEquals() de JUnit)

Première version du Code

```
class FizzBuzz {  
  
    private final int valeur;  
  
    String afficher() {  
        return null;  
    }  
  
    public FizzBuzz(int nb) {  
        valeur = nb;  
    }  
}
```

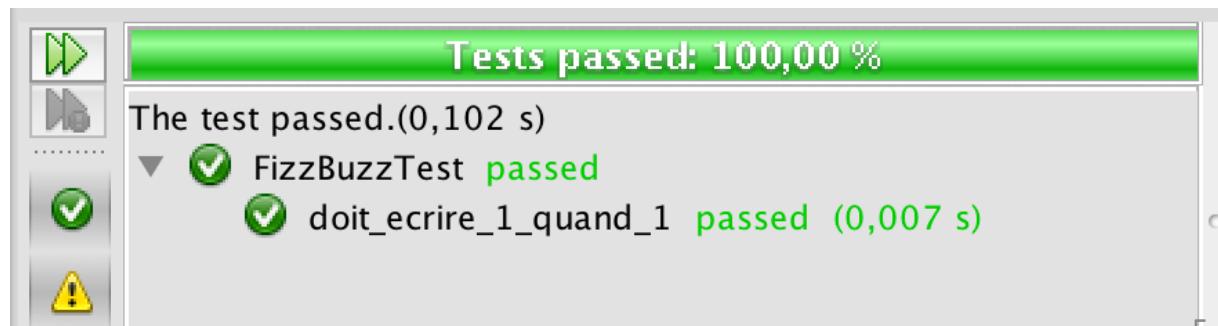
*On fait tourner le test :
clic droit, Run File*



1- Code pour avoir 1 quand 1

```
class FizzBuzz {  
  
    private final int valeur;  
  
    String afficher() {  
        return "1";  
    }  
  
    public FizzBuzz(int nb) {  
        valeur = nb;  
    }  
  
}
```

*OK, on sait afficher 1
quand on appelle la
fonction avec 1*



1- Refactoring du code de test

@Test

```
public void doit_ecrire_1_quand_1() {
```

```
    // given 1
```

```
    int n = 1;
```

```
    // when
```

```
    FizzBuzz fb = new FizzBuzz(n);
```

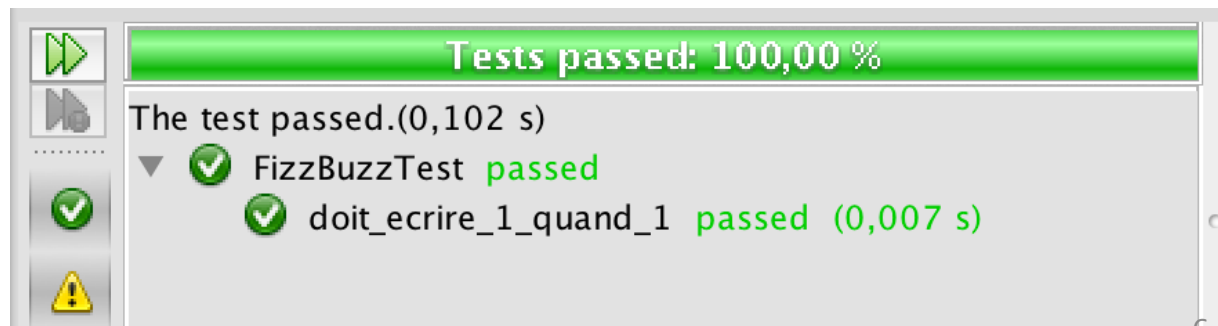
```
    String result = fb.afficher();
```

```
    // then
```

```
    assertThat(result,is("1"));
```

```
}
```

C'est plus correct !



2^{ème} test : affiche n quand n

@Test

```
public void doit_ecrire_n_quand_n() {
```

```
    // given
```

```
    int n = 278;
```

```
    // when
```

```
    FizzBuzz fb = new FizzBuzz(n);
```

```
    String result = fb.afficher();
```

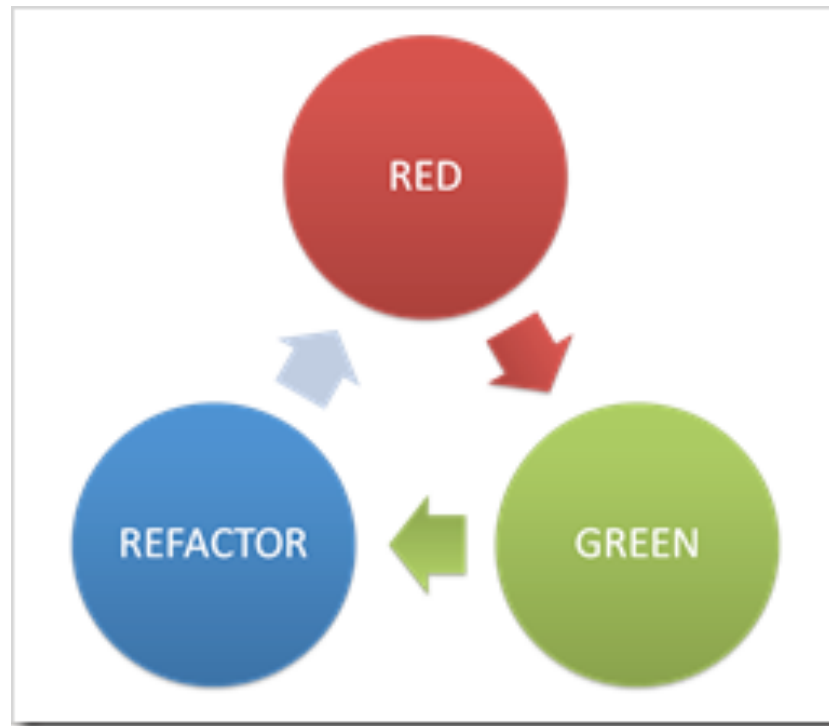
```
    // then
```

```
    assertThat(result,is("278"));
```

Le cycle du TDD

On en est où dans l'exercice ?

On **écrit un test** concernant une nouvelle spécification, qui échoue

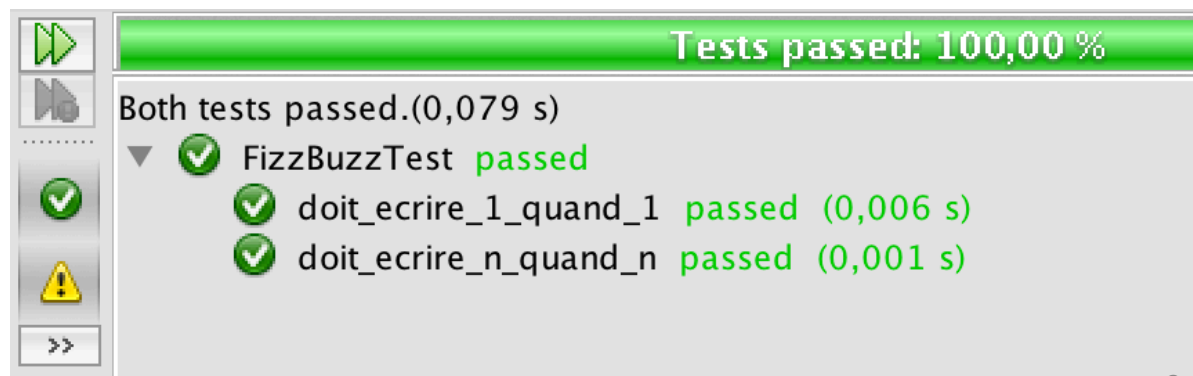


On **optimise** le code, avec la sécurité du test de non régression

On **écrit** le code pour que le test passe

2- Code pour avoir n quand n

```
class FizzBuzz {  
  
    private final int valeur;  
  
    String afficher() {  
        return Integer.toString(valeur);  
    }  
  
    public FizzBuzz(int nb) {  
        valeur = nb;  
    }  
}
```



3^{ème} test :

affiche Fizz quand multiple de 3

@Test

```
public void doit_ecrire_Fizz_quand_multiple_de_3() {
```

```
    // given
```

```
    int n = 3;
```

```
    // when
```

```
    FizzBuzz fb = new FizzBuzz(n);
```

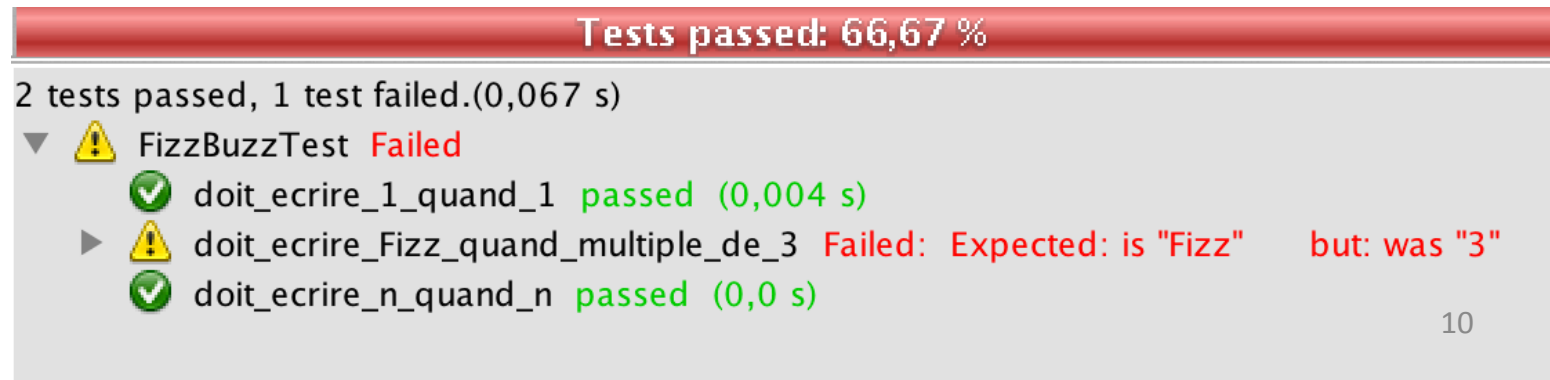
```
    String result = fb.afficher();
```

```
    // then
```



```
    assertThat(result, is("Fizz"));
```

*Normal, on n'a pas encore
écrit le code !*

```
}
```

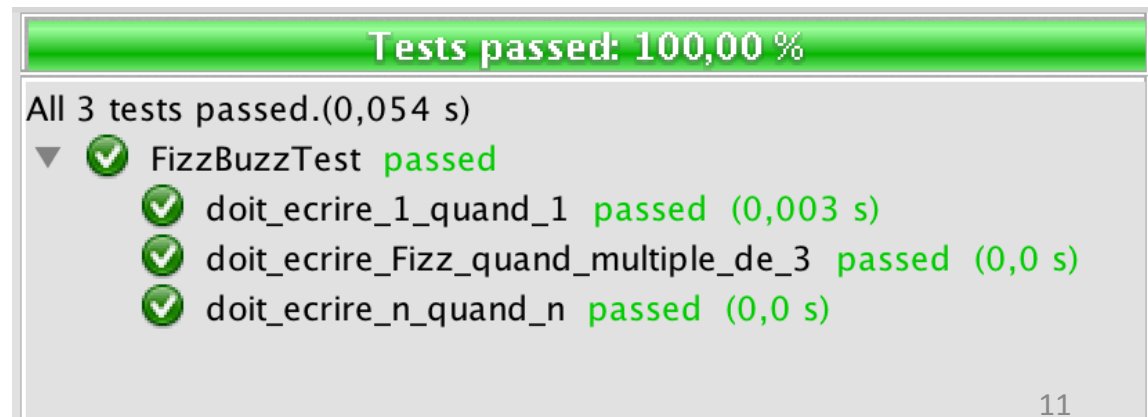


The screenshot shows a test runner window with a red header bar that reads "Tests passed: 66,67 %". Below the header, the text "2 tests passed, 1 test failed.(0,067 s)" is displayed. A tree view shows the test results:

- ▼  FizzBuzzTest **Failed**
 - ✓ doit_ecrire_1_quand_1 **passed** (0,004 s)
 - ▶  doit_ecrire_Fizz_quand_multiple_de_3 **Failed: Expected: is "Fizz" but: was "3"**
 - ✓ doit_ecrire_n_quand_n **passed** (0,0 s)

3- Code pour avoir *Fizz* quand *multiple de 3*

```
class FizzBuzz {  
  
    private final int valeur;  
  
    String afficher() {  
        if (valeur % 3 == 0)  
            return "Fizz";  
        else  
            return Integer.toString(valeur);  
    }  
  
    public FizzBuzz(int nb) {  
        valeur = nb;  
    }  
}
```



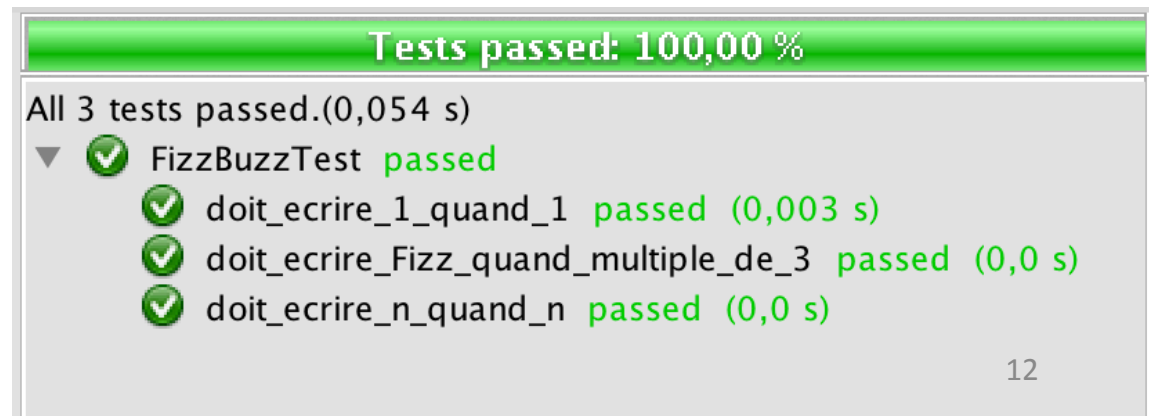
3- Refactoring de code : utiliser une constante pour Fizz

- Clic droit sur "Fizz" :
 - Refactor... - Introduce – Constant...

```
public static final String FIZZ = "Fizz";
```

```
...
```

```
if (valeur % 3 == 0)  
    return FIZZ;
```

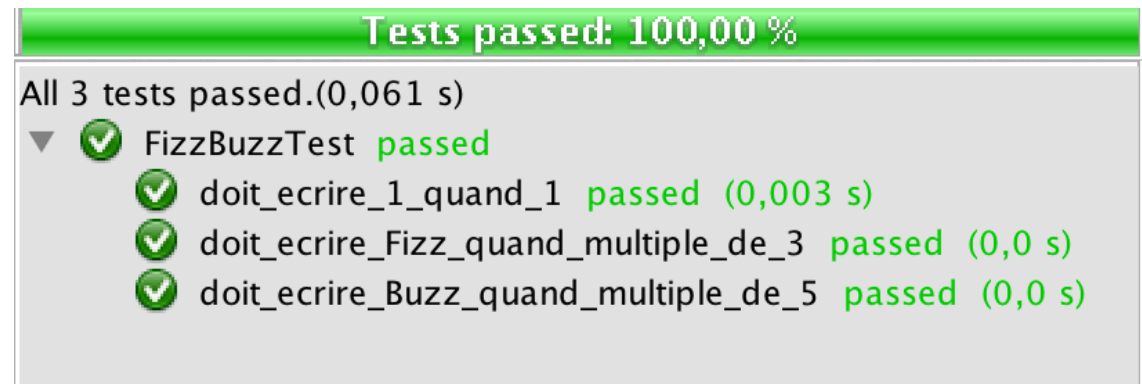


4^{ème} test : affiche Buzz quand multiple de 5

```
@Test
public void doit_ecrire_Buzz_quand_multiple_de_5() {
    // given
    int n = 25;
    // when
    FizzBuzz fb = new FizzBuzz(n);
    String result = fb.afficher();
    // then
    assertThat(result, is("Buzz"));
}
```

4- Code ajouté pour Buzz

```
class FizzBuzz {  
  
    public static final String FIZZ = "Fizz";  
    private final int valeur;  
  
    String afficher() {  
        if (valeur % 3 == 0)  
            return FIZZ;  
        else  
            if (valeur % 5 == 0)  
                return "Buzz";  
            else  
                return Integer.toString(valeur);  
    }  
}
```



4- Refactoring : code plus explicite

- On décide de faire une méthode qui teste si le nb est multiple de 3
 - Lecture du code plus aisée
- Sélection de (valeur % 3 == 0), clic droit :
Refactor – Introduce – Method...
- On l'appelle estMultipleDe3()
 - Il la crée automatiquement :

- (Idem pour 5)

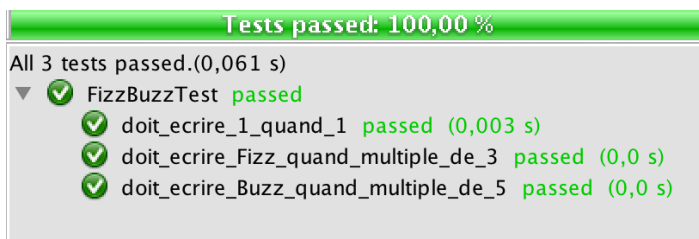
```
public boolean estMultipleDe3() {  
    return (valeur % 3 == 0);  
}
```

4- Refactoring (suite)

On met aussi
« Buzz » en
constante

Et

on reformate le
code
automatique-
ment
(menu Source –
Format)



```
public class FizzBuzz {  
  
    public static final String FIZZ = "Fizz";  
    public static final String BUZZ = "Buzz";  
    private final int valeur;  
  
    public String afficher() {  
        if (estMultipleDe5()) {  
            return BUZZ;  
        } else {  
            if (estMultipleDe3()) {  
                return FIZZ;  
            } else {  
                return Integer.toString(valeur);  
            }  
        }  
    }  
  
    public boolean estMultipleDe5() {  
        return (valeur % 5 == 0);  
    }  
  
    public boolean estMultipleDe3() {  
        return (valeur % 3 == 0);  
    }  
  
    public FizzBuzz(int i) {  
        valeur = i;  
    }  
}
```

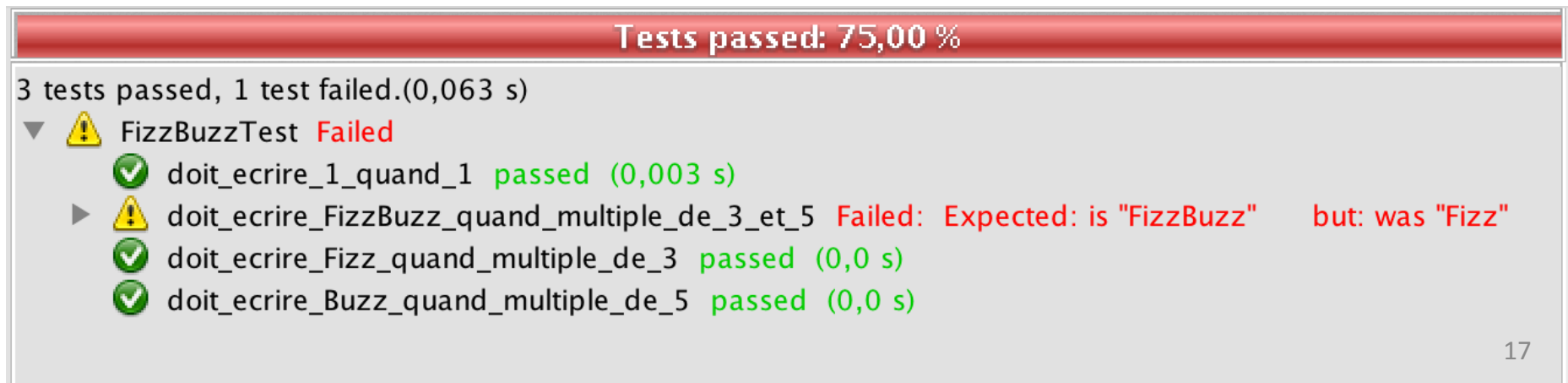

5^{ème} test :

affiche FizzBuzz quand multiple de 3 et de 5



@Test

```
public void doit_ecrire_FizzBuzz_quand_multiple_de_3_et_5() {  
    // given  
    int n = 45;  
    // when  
    FizzBuzz fb = new FizzBuzz(n);  
    String result = fb.afficher();  
    // then  
    assertThat(result, is( "FizzBuzz"));  
}
```

*Normal, on n'a pas encore
écrit le code !*



The screenshot shows a test runner window with a red header bar that reads "Tests passed: 75,00 %". Below the header, it says "3 tests passed, 1 test failed.(0,063 s)". A tree view shows the test results:

- ▼  FizzBuzzTest **Failed**
 - ✓ doit_ecrire_1_quand_1 **passed** (0,003 s)
 - ▶  doit_ecrire_FizzBuzz_quand_multiple_de_3_et_5 **Failed: Expected: is "FizzBuzz" but: was "Fizz"**
 - ✓ doit_ecrire_Fizz_quand_multiple_de_3 **passed** (0,0 s)
 - ✓ doit_ecrire_Buzz_quand_multiple_de_5 **passed** (0,0 s)

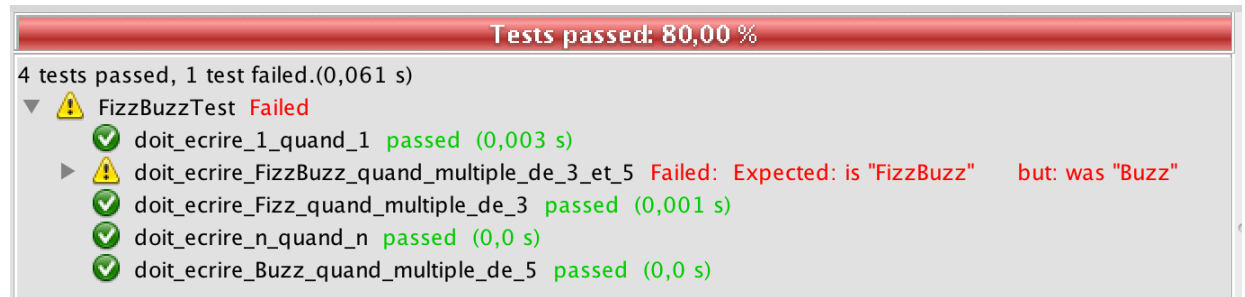
4- Code ajouté pour FizzBuzz

```
class FizzBuzz {
```

Rappel : n = 45

```
// def constantes FIZZ et BUZZ  
private final int valeur;
```

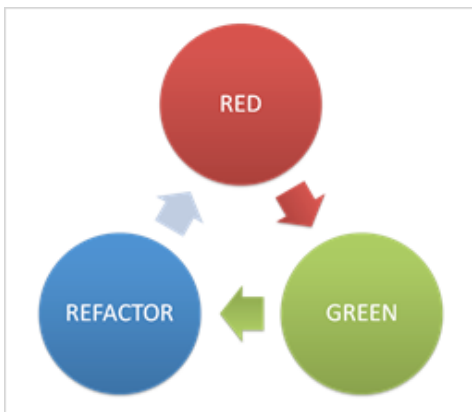
```
String afficher() {  
    if ( estMultipleDe5() )  
        return BUZZ;  
    else  
        if ( estMultipleDe3() )  
            return FIZZ;  
        else  
            if ( estMultipleDe3() && estMultipleDe5() )  
                return "FizzBuzz";  
            else  
                return Integer.toString(valeur);  
}
```



*Normal, on a écrit les conditions
bêtement !*

4- Code *corrigé* pour FizzBuzz

```
public String afficher() {  
    if ( estMultipleDe3() && estMultipleDe5() )  
        return FIZZ_BUZZ;  
    else  
        if (estMultipleDe5())  
            return BUZZ;  
        else  
            if (estMultipleDe3())  
                return FIZZ;  
            else  
                return Integer.toString(valeur);  
}
```



Tests passed: 100,00 %

All 5 tests passed.(0,053 s)

- ✔ FizzBuzzTest passed
 - ✔ doit_ecrire_1_quand_1 passed (0,003 s)
 - ✔ doit_ecrire_FizzBuzz_quand_multiple_de_3_et_5 passed (0,0 s)
 - ✔ doit_ecrire_Fizz_quand_multiple_de_3 passed (0,0 s)
 - ✔ doit_ecrire_n_quand_n passed (0,0 s)
 - ✔ doit_ecrire_Buzz_quand_multiple_de_5 passed (0,0 s)

19

TDD : Je retiens

Sur cet exemple, on a mis en œuvre la démarche du TDD :

- On écrit les tests **avant le code**
 - Un test par fonctionnalité attendue
 - On démarre avec du code trivial
- On **écrit le code** de la fonctionnalité
- On rejoue le test
 - pour qu'il passe au vert
- On **refactore le code** si nécessaire pour l'optimiser
 - Sans régression (sous contrôle du test !)
- Et ainsi de suite avec les autres fonctionnalités



Tests passed: 80,00 %



Tests passed: 100,00 %