Chapitre 1

TDD
Test Driven Development

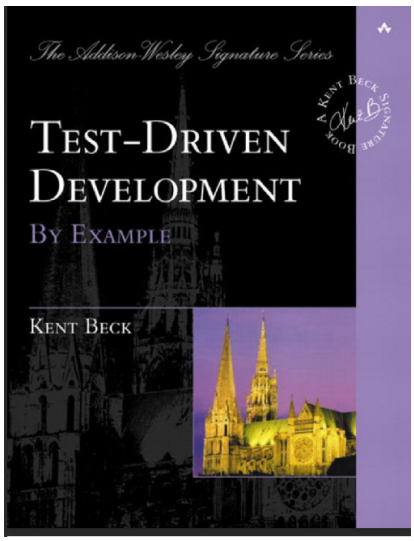
Ressource R5.AD : Qualité de développement

Institut Universitaire de Technologie de Bayonne – Pays Basque
BUT Informatique – Semestre 5 - D.Urruty, P.Dagorret, M.Erritali

v1.4

Origine

- Test Driver Development = Développement Dirigé par les Tests
- Démarche / Technique associée au mouvement eXtreme Programming
- Popularisée par Kent Back (encore lui) dans les années 2000



Approches ‘Test After’ versus ‘Test First’

- Développement et validation sont souvent **décorrélés**
  - Personnes différentes
    … rôle du développeur est de développer…
    … rôle du testeur est de tester (donc bien souvent après le codage)
  - Ecriture des tests **après** le code…
    … si on y pense…
    … si on a le temps…
    …et ces tests sont bien souvent incomplets

  De plus, un défaut détecté tardivement coûte plus cher à corriger
    … il faut arrêter la tâche en cours…
    … il faut se remettre dans le contexte…

- L’eXtreme Programming incite à **rapprocher les tests du développement, et favorise une approche ‘test first’**

Qu’est-ce ce que le TDD ?

Principe

- Positionnement ‘Test First’ : chaque test est écrit **avant** le bout de code qu’il est sensé tester
- Cycle de développement **itératif**
- Chaque cycle se consacre à un « baby step » (pas de bébé), c’est à dire à la mise en place d’une (et une seule) fonctionnalité/propriété de petite granularité

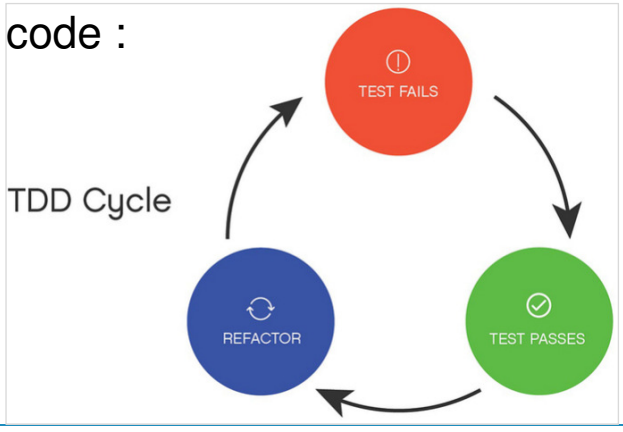
Le cycle du TDD

Pour chaque cycle de développement du code :

1.- écrire un test qui échoue (red)

2.- écrire le code métier MINIMAL qui fait réussir le test (green)

3.- retravailler le code métier pour améliorer sa qualité *interne*, sans ajouter de nouvelles propriétés (blue)



Étapes du TDD (1/2)

Étape 1 – Red : Écrire un test qui échoue

- Il est important de le faire échouer pour s'assurer que le message d'échec est correct (que le test fonctionne bien)
- On écrit le minimum de code de test pour faire échouer le test
- On n'écrit encore aucune ligne de code de production à cette étape !
  Excepté le minimum pour faire passer la compilation
- On écrit un seul test à cette étape !

Étape 2 – Green : Faire passer le test

- Un test qui échoue est une situation inconfortable…
  … alors on le fait passer au vert le plus vite possible…
  … c'est-à-dire avec le code le plus minimal possible (même moche !)

Des remarques sur cette approche par rapport à votre pratique ?

Étapes du TDD (2/2)

Étape 3 – Blue (en option) : Refactor

- Une fois au vert, on peut prendre le temps de remanier le code
- Cette étape n'est pas obligatoire
- Il est même recommandé d'avoir suffisamment de tests pour commencer le remaniement du code
- Attention, le refactoring ne concerne que l'amélioration de la structure du code, et ne concerne pas le fonctionnel…

Mais avant tout cela : Étape 0 – Think !

- Bien comprendre le besoin et les différents scénarios
- Etablir une liste de tests associés à ces scénarios
- Le TDD est compatible avec une conception (souvent légère) en amont

Apports du TDD (1/3)

Couverture de Code

- Si l'on suit le cycle rigoureusement, on obtient automatiquement une couverture de code maximale
- Chaque branche de code est normalement couverte par un test dédié

Documentation

- Les tests aident à la (parfois servent de) documentation pour le module testé
- Chaque scénario / cas d'usage est représenté par un test
- Il est donc important de bien nommer les tests, c'est-à-dire de les nommer en référence au scénario testé et au résultat attendu

Apports du TDD (2/3)

Confiance

- Voir les tests passer au vert donne confiance dans ce que l'on développe
- L'avancement des tests permet d'estimer l'avancement du codage

Code fonctionnel à chaque étape

- A la fin de chaque itération du cycle, on a du code qui fonctionne et qui correspond à de vrais cas d'usage
- Cela permet de montrer des résultats au client, même incomplets

Débogage diminué

- L'approche TDD diminue en général le besoin d'utiliser le débogueur
- Si un bug est découvert, on démarre un nouveau cycle :
  - écrire un test qui échoue,
  - corriger le bug,
  - puis refactor si besoin

Apports du TDD (3/3)

Conception de Qualité

- Les tests obligent à penser à l'interface (API) du module testé : on doit en effet utiliser le module avant même de l'avoir implémenté
- Ils mettent une pression continue sur la conception
- Ils mènent à du code plus modulaire, flexible
- S'il est compliqué d'écrire un test, c'est probablement que la conception est perfectible
- Il faut donc « écouter » les tests : ce sont eux qui indiquent si la conception est bonne

Non régression

- Tous les tests accumulés lors des différents cycles constituent une barrière de sécurité de non régression

Exemple d'application

Problème

- Etant donnée une chaîne de caractères, produire une nouvelle chaîne dont les 2 derniers caractères ont été échangés par rapport à la chaine initiale.
- Exemples : "" → "" , "A" → "A", "AB" → "BA", "RIEN" → "RINE"

Rappel java : str.charAt(i) donne accès au caractère situé au rang i de la chaîne (en commençant à 0)

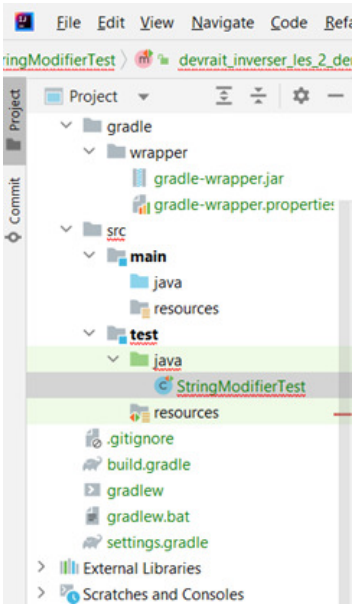
Scénarios à tester (dans cet ordre)

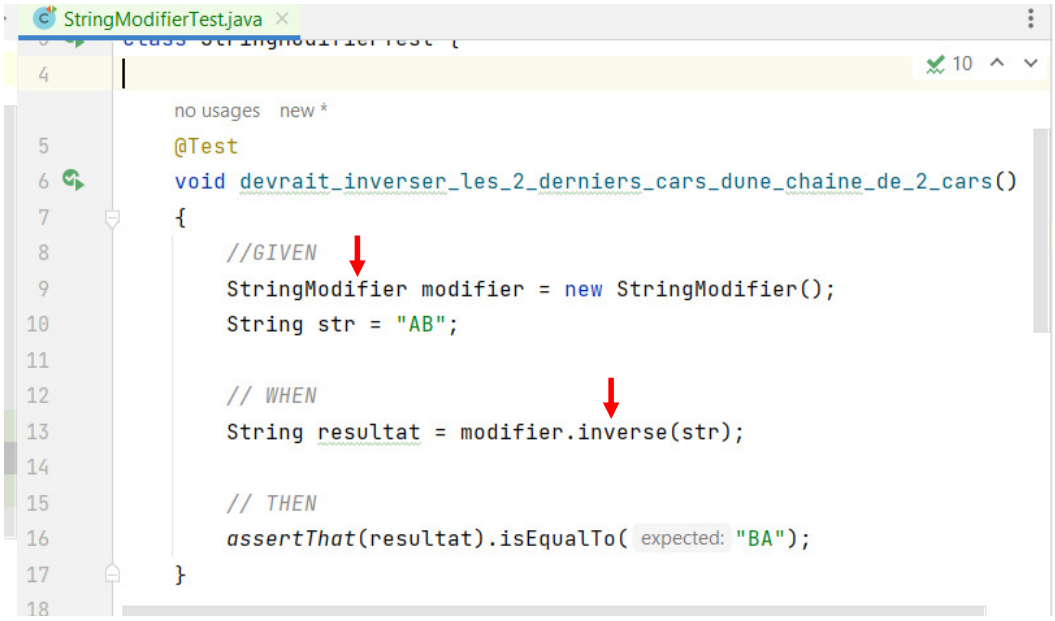
"AB" → "BA" (chaîne minimale non particulière)
"RIEN" → "RINE", (généralisation : chaîne quelconque avec plus de 2 cars)
"A" → "A", (un cas particulier)
"" → "" (un cas particulier)

Exemple d'application – Cycle 1

1) Création d'une classe de test, puis du 1er test correspondant au premier scénario : "AB" → "BA"

@Test

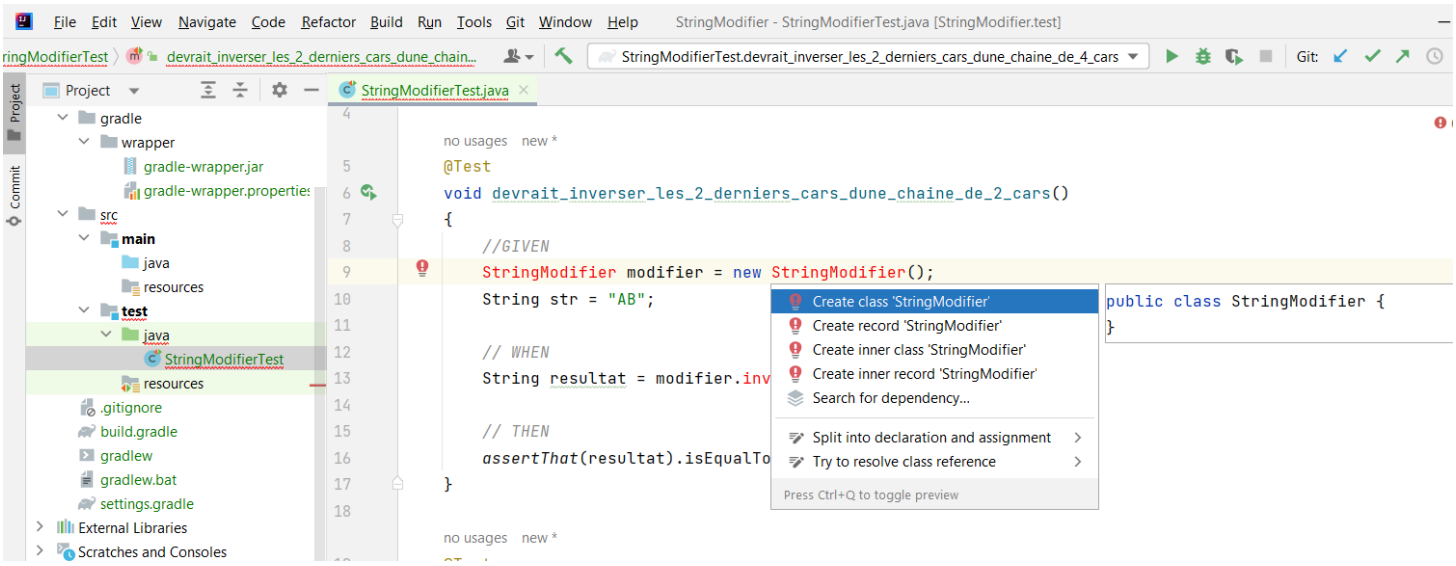




Exemple d'application – Cycle 1

Problème compilation : **classe** StringModifier inconnue

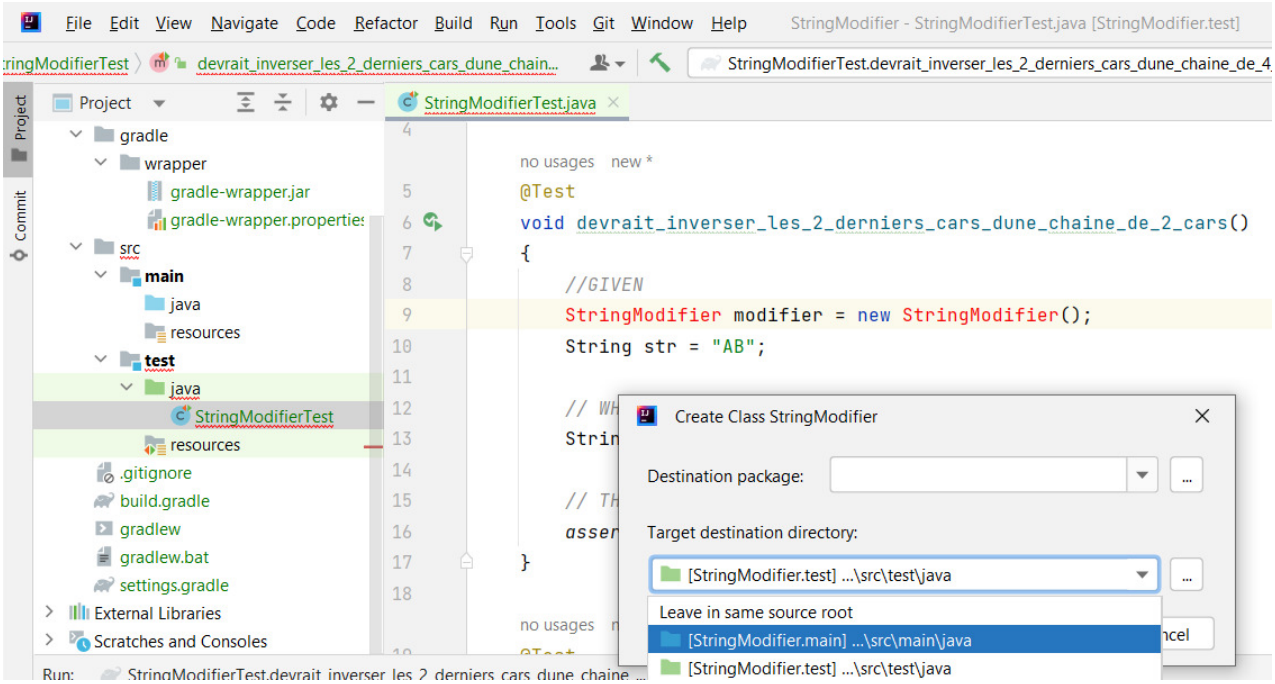
@Test



Exemple d'application – Cycle 1

2) Création de la classe StringModifier dans src/main/java

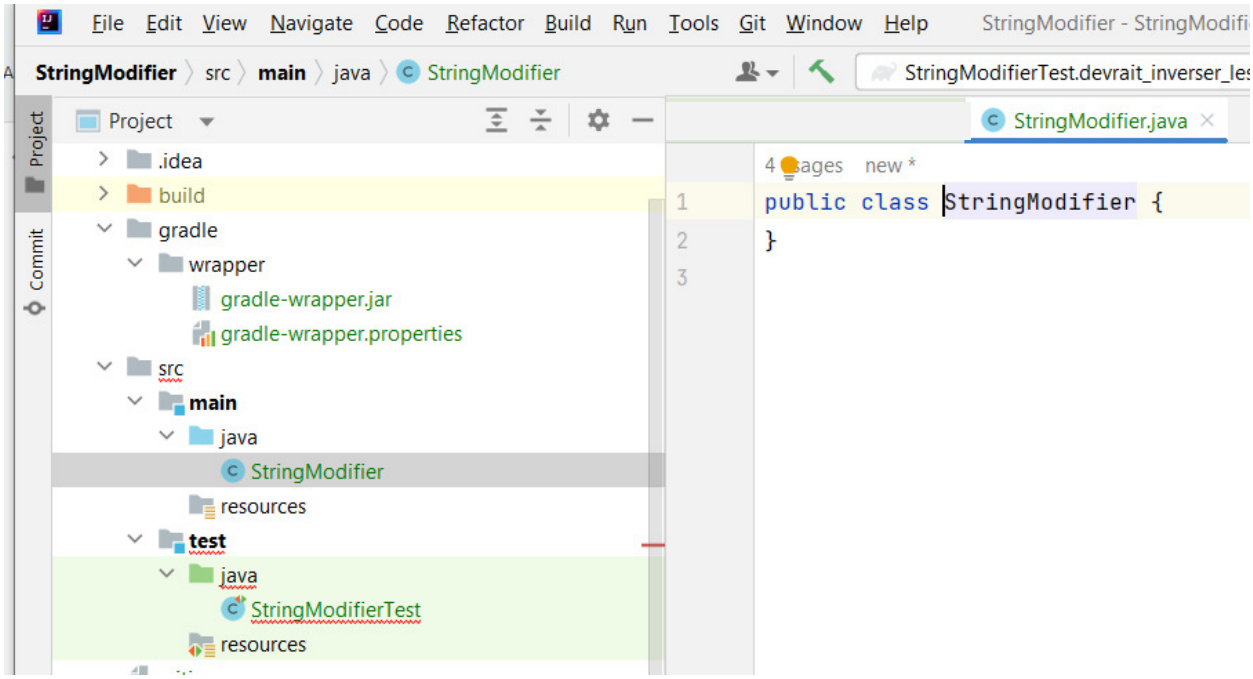
@Test



Exemple d'application – Cycle 1

2) Création de la classe StringModifier + Compilation

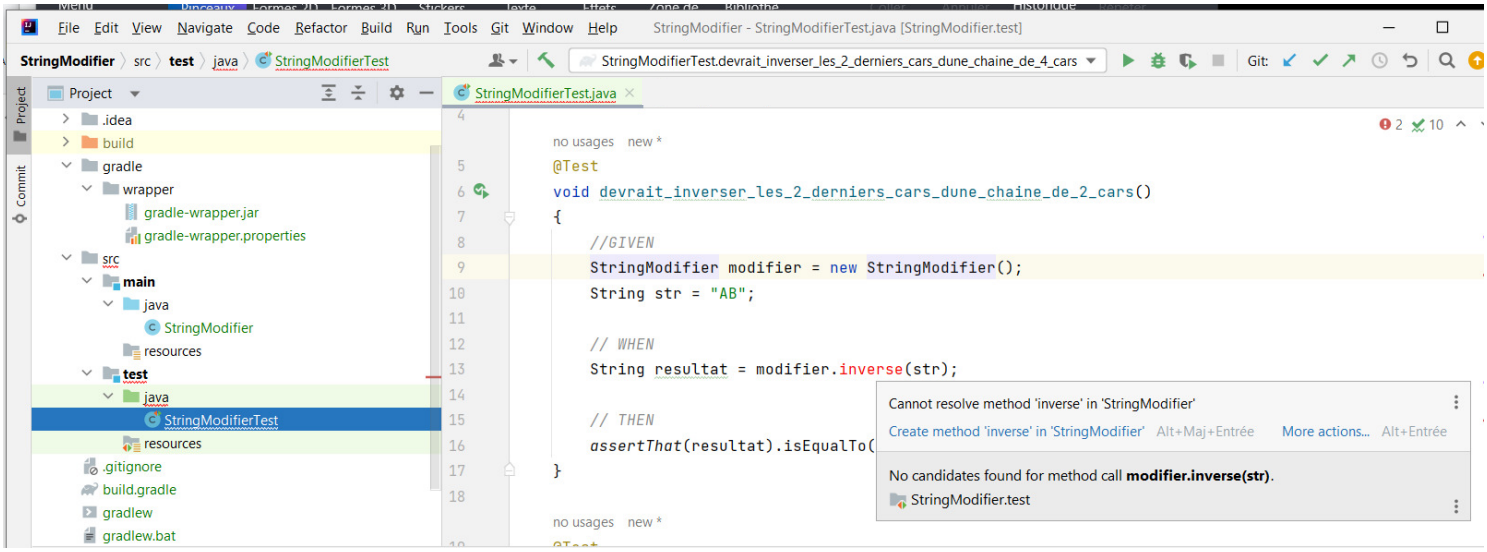
@Classe



Exemple d'application – Cycle 1

Problème compilation : **méthode** `inverse()` inconnue

@Test 1 v1

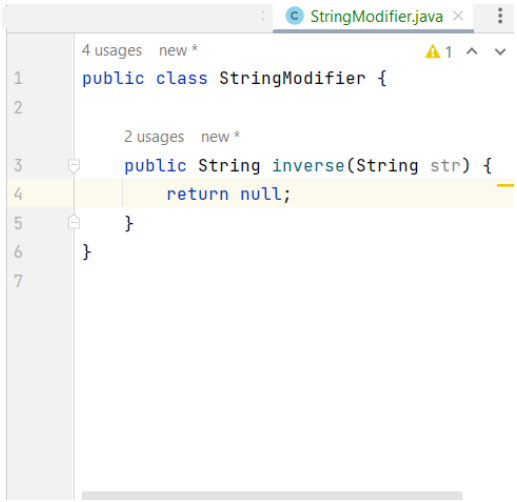


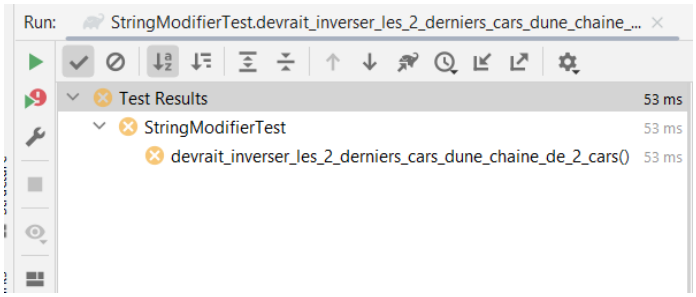
Exemple d'application – Cycle 1

3) Création méthode `inverse()` + Exécution test

@Test 1 v1

inverse() v1





Exemple d'application – Cycle 1

4) Modification (minimale) méthode `inverse()` + Exécution test

@Test 1 v1

```java
no usages  new *
@Test
void devrait_inverser_les_2_derniers_cars_dune_chaine_de_2_cars()
{
    //GIVEN
    StringModifier modifier = new StringModifier();
    String str = "AB";

    // WHEN
    String resultat = modifier.inverse(str);

    // THEN
    assertThat(resultat).isEqualTo( expected: "BA");
}
```

inverse() v2

```java
4 usages  new *
public class StringModifier {
    2 usages  new *
    public String inverse(String str) {
        char premierCar = str.charAt(0);
        char secondCar = str.charAt(1);

        return ("" + secondCar + premierCar);
    }
}
```

Run: StringModifierTest.devrait_inverser_les_2_derniers_cars_dune_chaine_...

Test Results 48 ms
StringModifierTest 48 ms
devrait_inverser_les_2_derniers_cars_dune_chaine_de_2_cars() 48 ms

✓

Exemple d'application – Cycle 1

Remaniement (Refactor) de la méthode `inverse()` ? **NON**

inverse() v2

```java
4 usages  new *
public class StringModifier {
    2 usages  new *
    public String inverse(String str) {
        char premierCar = str.charAt(0);
        char secondCar = str.charAt(1);

        return ("" + secondCar + premierCar);
    }
}
```

Exemple d'application – Cycle 2

1) Création du test correspondant au second scénario : "RIEN" → "RINE"

@Test 2 **v1**

```
19      @Test
20      void devrait_inverser_les_2_derniers_cars_dune_chaine_de_4_cars()
21      {
22          //GIVEN
23          StringModifier modifier = new StringModifier();
24          String str = "RIEN";
25
26          // WHEN
27          String resultat = modifier.inverse(str);
28
29          // THEN
30          assertThat(resultat).isEqualTo( expected: "RINE");
31      }
```


Exemple d'application – Cycle 2

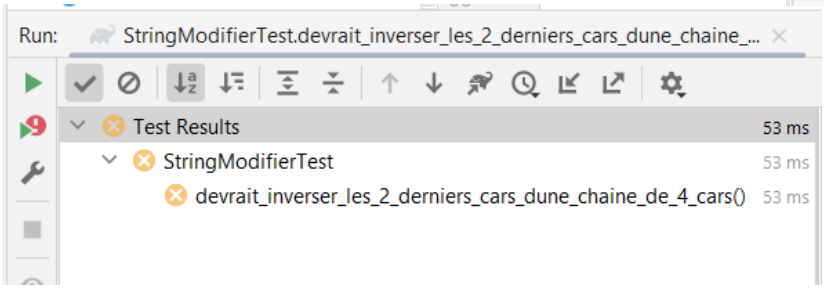
2) Exécution du test ***sans modifier*** la méthode `inverse()`

@Test 2 **v1** **inverse()** **v2**

```
19      @Test
20      void devrait_inverser_les_2_derniers_cars_dune_chaine_de_4_cars
21      {
22          //GIVEN
23          StringModifier modifier = new StringModifier();
24          String str = "RIEN";
25
26          // WHEN
27          String resultat = modifier.inverse(str);
28
29          // THEN
30          assertThat(resultat).isEqualTo( expected: "RINE");
31      }
```

StringModifier.java

```
4 usages   new *
1   public class StringModifier {
        2 usages   new *
2   @       public String inverse(String str) {
3               char premierCar = str.charAt(0);
4               char secondCar = str.charAt(1);
5
6               return ("" + secondCar + premierCar);
7       }
8   }
```



Exemple d'application – Cycle 2

3) Modification (minimale) méthode `inverse()`, noter les changements de noms des variables + Exécution test

@Test 2 v1

```java
@Test
void devrait_inverser_les_2_derniers_cars_dune_chaine_de_2_cars()
{
    //GIVEN
    StringModifier modifier = new StringModifier();
    String str = "AB";

    // WHEN
    String resultat = modifier.inverse(str);

    // THEN
    assertThat(resultat).isEqualTo( expected: "BA");
}
```

inverse() v3

```java
public class StringModifier {
    2 usages  new *
    public String inverse(String str) {
        int longueur = str.length();

        char avantDernierCar = str.charAt(longueur-2);
        char dernierCar = str.charAt(longueur - 1);

        return ("" + dernierCar + avantDernierCar);
    }
}
```

Run: StringModifierTest.devrait_inverser_les_2_derniers_cars_dune_chaine_... Tests failed: 1 of 1 test – 58 ms

Test Results 58 ms
StringModifierTest 58 ms
devrait_inverser_les_2_derniers_cars_dune_chaine_de_4_cars() 58 ms

```
expected: "RINE"
 but was: "NE"
org.opentest4j.AssertionFailedError:
expected: "RINE"
 but was: "NE"
```


Exemple d'application – Cycle 2

4) Modification (minimale) méthode `inverse()` + Exécution test

@Test 2 v1

```java
@Test
void devrait_inverser_les_2_derniers_cars_dune_chaine_de_2_car
{
    //GIVEN
    StringModifier modifier = new StringModifier();
    String str = "AB";

    // WHEN
    String resultat = modifier.inverse(str);

    // THEN
    assertThat(resultat).isEqualTo( expected: "BA");
}
```

inverse() v4

```java
4 usages  new *
public class StringModifier {

    2 usages  new *
    public String inverse(String str) {
        int longueur = str.length();
        String racine = str.substring(0, longueur-2);

        char avantDernierCar = str.charAt(longueur-2);
        char dernierCar = str.charAt(longueur - 1);

        return (racine + dernierCar + avantDernierCar);
    }
}
```

Run: StringModifierTest.devrait_inverser_les_2_derniers_cars_dune_chaine_... Tests passed: 1 of 1 t

Test Results 62 ms
StringModifierTest 62 ms
devrait_inverser_les_2_derniers_cars_dune_chaine_de_4_cars() 62 ms

Exemple d'application – Cycle 2

Remaniement (Refactor) de la méthode `inverse()` ? **NON**

inverse() v4

```
4 usages  new *
public class StringModifier {

    2 usages  new *
    public String inverse(String str) {
        int longueur = str.length();
        String racine = str.substring(0, longueur-2);

        char avantDernierCar = str.charAt(longueur-2);
        char dernierCar = str.charAt(longueur - 1);

        return (racine + dernierCar + avantDernierCar);
    }
}
```


Exemple d'application – Cycle 2

Remaniement (Refactor) des tests ? **OUI**

5) Introduction des méthodes de montage / démontage dans les tests

@Tests

StringModifierTest.java

```
import static org.assertj.core.api.Assertions.assertThat;

no usages  new *
class StringModifierTest {

    4 usages
    private StringModifier modifier;

    no usages  new *
    @BeforeEach
    void setUp() { modifier = new StringModifier(); }

    no usages  new *
    @AfterEach
    void tearDown() { modifier = null; }
```

Exemple d'application – Cycle 2

Remaniement (Refactor) des tests ? **OUI**

5) Introduction des méthodes de montage / démontage dans les tests

… et Refactor de @Test1 et @Test2 **v2**

```
14      @Test
15      void devrait_inverser_les_2_derniers_cars_dune_chaine_de_2_cars()
16      {
17          //GIVEN
18          String str = "AB";
19          // WHEN
20          String resultat = modifier.inverse(str);
21          // THEN
22          assertThat(resultat).isEqualTo( expected: "BA");
23      }
25      @Test
26      void devrait_inverser_les_2_derniers_cars_dune_chaine_de_4_cars()
27      {
28          //GIVEN
29          String str = "RIEN";
30          // WHEN
31          String resultat = modifier.inverse(str);
32          // THEN
33          assertThat(resultat).isEqualTo( expected: "RINE");
34      }
```


Exemple d'application – Cycle 3

1) Création du test d'inversion d'une chaîne (générale) de 10 caractères : "ABCDEFGHIJ" → "ABCDEFGHJI"

@Test 3 **v1**

```
35      @Test
36      void devrait_inverser_les_2_derniers_cars_dune_chaine_de_10_cars()
37      {
38          //GIVEN
39          String str = "ABCDEFGHIJ";
40          // WHEN
41          String resultat = modifier.inverse(str);
42          // THEN
43          assertThat(resultat).isEqualTo( expected: "ABCDEFGHJI");
44      }
```

Exemple d'application – Cycle 3

Exécution du test sans changer la méthode `inverse()`

@Test 3 **v1**

```
@Test
void devrait_inverser_les_2_derniers_cars_dune_chaine_de_10_cars()
{
    //GIVEN
    String str = "ABCDEFGHIJ";
    // WHEN
    String resultat = modifier.inverse(str);
    // THEN
    assertThat(resultat).isEqualTo( expected: "ABCDEFGHJI");
}
```

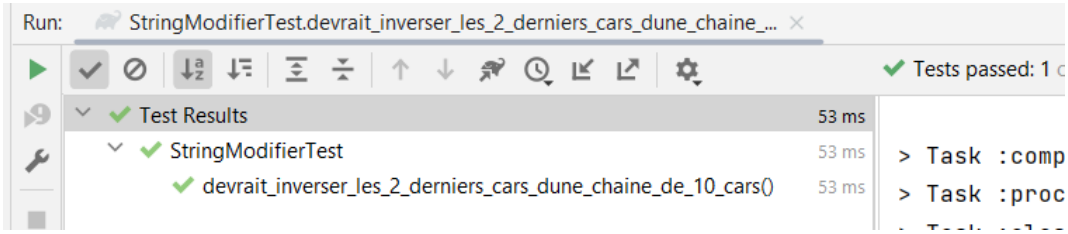
inverse() **v4**

```
4 usages  new *
public class StringModifier {

    2 usages  new *
    public String inverse(String str) {
        int longueur = str.length();
        String racine = str.substring(0, longueur-2);

        char avantDernierCar = str.charAt(longueur-2);
        char dernierCar = str.charAt(longueur - 1);

        return (racine + dernierCar + avantDernierCar);
    }
}
```



Exemple d'application – Cycle 4

1) Création du test correspondant au scénario : "A" → "A"

@Test 4 **v1**

```
@Test
void devrait_retourner_la_meme_chaine_de_1_car()
{
    //GIVEN
    String str = "A";
    // WHEN
    String resultat = modifier.inverse(str);
    // THEN
    assertThat(resultat).isEqualTo( expected: "A");
}
```

Exemple d'application – Cycle 4

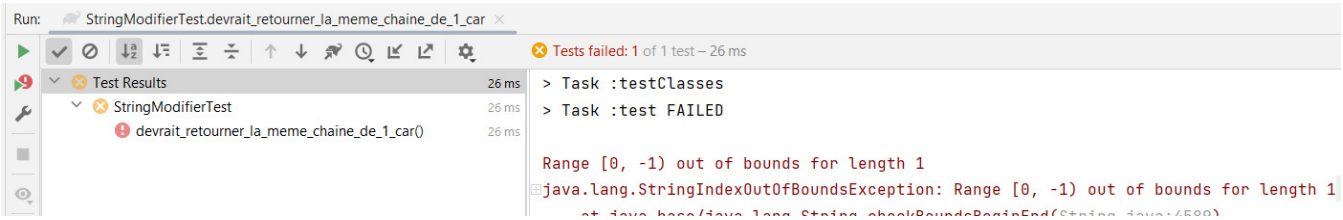
Exécution du test sans changer la méthode `inverse()`. Problème d'indice qui dépasse les limites de la taille de la chaîne

@Test 4 v1

```
46      @Test
47      void devrait_retourner_la_meme_chaine_de_1_car()
48      {
49          //GIVEN
50          String str = "A";
51          // WHEN
52          String resultat = modifier.inverse(str);
53          // THEN
54          assertThat(resultat).isEqualTo( expected: "A");
55      }
```

inverse() v4

```
4 usages   new *
1   public class StringModifier {
2
        2 usages   new *
3   @   public String inverse(String str) {
4           int longueur = str.length();
5           String racine = str.substring(0, longueur-2);
6
7           char avantDernierCar = str.charAt(longueur-2);
8           char dernierCar = str.charAt(longueur - 1);
9
10          return (racine + dernierCar + avantDernierCar);
11      }
12  }
```



Exemple d'application – Cycle 4

2) Modification (minimale) méthode inverse() : la méthode retourne la chaîne initiale lorsque sa longueur est < 2 + Exécution test

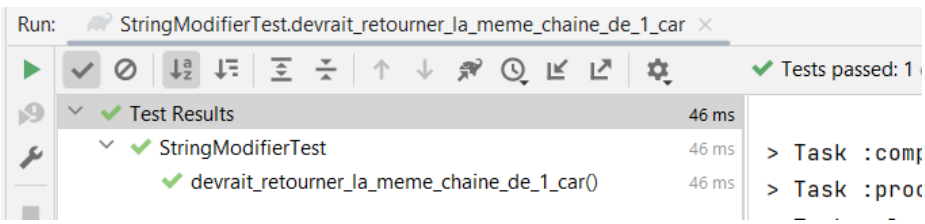
@Test 4 v1

```
46      @Test
47      void devrait_retourner_la_meme_chaine_de_1_car()
48      {
49          //GIVEN
50          String str = "A";
51          // WHEN
52          String resultat = modifier.inverse(str);
53          // THEN
54          assertThat(resultat).isEqualTo( expected: "A");
55      }
```

inverse() v5

StringModifier.java

```
2   @   public String inverse(String str) {
3           int longueur = str.length();
4
5           if (longueur < 2) return str;
6
7           String racine = str.substring(0, longueur-2);
8
9           char avantDernierCar = str.charAt(longueur-2);
10          char dernierCar = str.charAt(longueur - 1);
11
12          return (racine + dernierCar + avantDernierCar);
13      }
```



Exemple d'application – Cycle 4

Remaniement (Refactor) de la méthode `inverse()` ? **OUI**
(mais pas forcément obligatoire, cf. Robert C. Martin - *Uncle Bob*[1])

3) Un seul return dans la méthode

inverse() **v6**

StringModifier.java

```
4 usages  new *
public String inverse(String str) {
    String chaineARetourner = str;      // initialisation
    int longueur = str.length();

    if (longueur >=  2) {               // modification si longueur >= 2
        String racine = str.substring(0, longueur-2);

        char avantDernierCar = str.charAt(longueur-2);
        char dernierCar = str.charAt(longueur - 1);

        chaineARetourner = racine + dernierCar + avantDernierCar;
    }
    return chaineARetourner;
}
```

(1) Selon les principes de programmation structurée (E. Dijktra), une fonction doit avoir un seul return. MAIS cette règle peut être ignorée si la fonction est très courte. Dans ce cas, les multiples instruction return peuvent donner lieu à un code plus simple et expressif (Clean Code – Robert C.Martin)

Exemple d'application – Cycle 5

1) Création du test correspondant au dernier scénario : "" → ""

@Test 5 **v1**

StringModifierTest.java

```
no usages  new *
@Test
void devrait_retourner_la_meme_chaine_vide()
{
    //GIVEN
    String str = "";
    // WHEN
    String resultat = modifier.inverse(str);
    // THEN
    assertThat(resultat).isEqualTo( expected: "");
}
```

Exemple d'application – Cycle 5

Exécution du test sans changer la méthode `inverse()`

@Test 5 **v1**

StringModifierTest.java

```
57  @Test
58  void devrait_retourner_la_meme_chaine_vide()
59  {
60      //GIVEN
61      String str = "";
62      // WHEN
63      String resultat = modifier.inverse(str);
64      // THEN
65      assertThat(resultat).isEqualTo( expected: "");
66  }
```

inverse() **v6**

StringModifier.java

```
2   public String inverse(String str) {
3       String chaineARetourner = str;      // initialisation
4       int longueur = str.length();
5
6       if (longueur >=  2) {               // modification si longueur >= 2
7           String racine = str.substring(0, longueur-2);
8
9           char avantDernierCar = str.charAt(longueur-2);
10          char dernierCar = str.charAt(longueur - 1);
11
12          chaineARetourner = racine + dernierCar + avantDernierCar;
13      }
14      return chaineARetourner;
15  }
```

Run: StringModifierTest.devrait_retourner_la_meme_chaine_vide

Tests passed: 1

Test Results 50 ms
StringModifierTest 50 ms
devrait_retourner_la_meme_chaine_vide() 50 ms

```
> Task :comp
> Task :proc
```


Exemple d'application – Bilan des tests

La méthode `inverse()` vérifie tous les scénarios.
Elle est donc terminée, fonctionnelle et correcte.
La liste des tests, quand ils sont bien nommés, documente son comportement.

Run: StringModifierTest

Tests passed: 5 of 5 tests – 63 ms

Test Results 63 ms
StringModifierTest 63 ms
devrait_inverser_les_2_derniers_cars_dune_chaine_de_2_cars() 2 ms
devrait_inverser_les_2_derniers_cars_dune_chaine_de_4_cars() 1 ms
devrait_inverser_les_2_derniers_cars_dune_chaine_de_10_cars() 7 ms
devrait_retourner_la_meme_chaine_de_1_car() 52 ms
devrait_retourner_la_meme_chaine_vide() 1 ms

```
> Task :compileJava UP-TO-DATE
> Task :processResources NO-SOURCE
> Task :classes UP-TO-DATE
> Task :compileTestJava UP-TO-DATE
> Task :processTestResources NO-SOURCE
> Task :testClasses UP-TO-DATE
> Task :test
BUILD SUCCESSFUL in 746ms
```

Refactoring & Transformations

Position du problème

- L'activité TDD consiste à développer du code en démarrant de 'rien' et en lui appliquant une série de modifications.
- Ces modifications sont de 2 natures :
- *Transformation*
  - Il s'agit d'une modification du code qui **change le comportement du code**
  - **Les transformations sont appliquées uniquement dans le but de faire passer un test au vert**
- *Refactoring*
  - Il s'agit d'une modification du code qui améliore sa qualité interne sans changer son comportement. Elle peut être omise (!).
  - Cela veut notamment dire qu'après une opération de refactoring, un test qui est rouge reste rouge.

Le mantra du TDD

As the tests get more specific, the code gets more generic

Robert C. Martin

Quand on fait du TDD, le code produit se modifie selon une séquence de transformations.

Ces transformations sont tout d'abord *spécifiques*, puis évoluent de sorte que le code devient de plus en plus *générique*.

Les Transformations (1/2)

La Liste connue à ce jour

- **{} → null** — pas de code du tout → code utilisant null
- **null → constant**
- **constant → constant++**
- **constant → variable / attribut** — remplacer une constante par une variable / attribut
- **statement → statements** — ajouter des instructions non conditionnelles
- **unconditional → if** — séparer / éclater le graphe d'exécution
- **éléments simples → tableaux**
- **tableaux → containers**
- **instruction → récursivité**
- **if → while**
- **expression → fonction** — remplacer une expression par une fonction

- **Elles orientent toutes le code dans la même direction** : celle de le transformer de spécifique à plus générique

Les Transformations (2/2)

Ordre de *présentation* des transformations :

- Les transformations sont ***présentées*** par ordre de *complexité* **croissant** :
  - Les transformations situées en haut de liste sont plus simples, et leur application risque moins de faire échouer le test
  - Les transformations situées en bas de liste sont plus complexes, et leur application risque davantage de faire échouer le test

Ordre *d'application* des transformations :

- Les transformations doivent être ***appliquées*** selon une *priorité* **inverse** à celle de la liste
  - L'application des transformations doit respecter cet ordre :
    - Transformations situées en **haut** de liste : à appliquer en **premier**
    - Transformations situées en **bas** de liste : à appliquer en **dernier**

Exemples (1/7)

{} → null

PrimeFactors

But : Retourner la liste des diviseurs premiers du paramètre entier fourni

Cycle 1 - Test 1 : `generate(1)` devrait retourner une liste vide

```
5   public class FacteursPpremiers {
        1 usage
6   @   public static List<Integer> generate(int nbre) {
7           return null;
8       }
```

✘

```
7   @   public static List<Integer> generate(int nbre) {
8           return new ArrayList<Integer>();
9       }
```

✓

Dans cet exercice, la situation {} n'était pas possible, car il s'agit de développer le code d'une fonction, qui doit forcément retourner une valeur résultante.

Return null correspondait donc à la première étape possible.

Exemples (2/7)

null → constant

Personnage

But : Retourner la nouvelle orientation d'un personnage après l'avoir fait tourner un nombre de fois (de ¼ de tour) fourni en paramètre

Cyle 1 - Test 1 : Partant de l'orientation `NORD`, `tourner(1)` devrait retourner la valeur `EST`

```
3   public class Personnage {
        2 usages
4       public Orientation tourner(int nbre) {
5           return Orientation.EST;
6       }
7   }
```

De manière analogue, dans cet exercice, la première situation possible consistait à retourner une valeur de type Orientation. C'est donc une constante.

Exemples (3/7)

constant → variable / attribut

PrimeFactors

Cycle 2 - Test 3 : `generate(3)` devrait retourner la liste `{3}`

```java
public static List<Integer> generate(int nbre) {
    List <Integer> facteurs = new ArrayList<Integer>();
    if (nbre > 1) {
        facteurs.add(2);
    }
    return facteurs;
}
```

```java
public static List<Integer> generate(int nbre) {
    List <Integer> facteurs = new ArrayList<Integer>();
    if (nbre > 1) {
        facteurs.add(nbre);
    }
    return facteurs;
}
```

Ici, la valeur littérale 2 ajoutée à la liste est bien remplacée par une variable, en l'occurrence le paramètre de la méthode, *car une variable est bien la généralisation d'une constante*.

Exemples (4/7)

unconditional → if

Personnage

Cycle 2 - Test 2 : Partant de l'orientation `NORD`, `tourner(2)` devrait retourner la valeur `SUD`

```java
public Orientation tourner(int nbre) {
    return Orientation.EST;
}
```

```java
public Orientation tourner(int nbre) {
    if (nbre > 1) {
        return Orientation.SUD;
    }
    return Orientation.EST;
}
```

Cette transformation, consistant à remplacer une instruction inconditionnelle par une condition if, augmente bien le nombre de chemins d'exécution possibles (il passe de 1 à 2).
C'est aussi une manière de généraliser l'algorithme, puisque l'on traite davantage de cas.

Exemples (5/7)

éléments simples → tableaux

ArabicToRoman

But : Retourner la version écrite en chiffres romaines du nombre entier, compris entre 1 et 50, passé en paramètre

2.17 Cycle 8 - Test 8 : `convert(5)` devrait retourner la chaîne `"V"`

```
4 @   public static String convert(int arabicNbre) {
5         StringBuilder romanNbre = new StringBuilder();
6         int reste = arabicNbre;
7
8         while (reste >=10 ) {
9               romanNbre.append("X");
10              reste = reste - 10;
11          }
12        while (reste >=1 ) {
13            romanNbre.append("I");
14            reste = reste - 1;
15        }
16        return romanNbre.toString();
17    }
```

```
3   public class ArabicToRoman {
        1 usage
4       private static final String[] symboles = {"X", "V", "I"};
        2 usages
5       private static final int[] valeurs = {10, 5, 1};
6
7       2 usages
        static class ResultatBuilder {...}
36
37      7 usages
        public static String convert(int arabicNbre) {
38          ResultatBuilder resultat = new ResultatBuilder(arabicNbre);
39
40          // Décrémentation de arabicNbre des valeurs et concaténation du
41          // symbole romain correspondant
42          for (int i = 0; i < valeurs.length; i++) {
43              resultat.compute(symboles[i], valeurs[i]);
44          }
45
46          return resultat.toString();
47      }
```

La fabrication du nombre en chiffres romains à partir de valeurs littérales est désormais réalisée à partir de valeurs d'un tableau de symboles.
La solution est plus générale, car on augmente le nombre de valeurs traitées.

Exemples (6/7)

if → while

PrimeFactors

Cycle 6 - Test 6 : `generate(8)` devrait retourner la liste `{2, 2, 2}`

```
7 @   public static List<Integer> generate(int nbre) {
8         List <Integer> facteurs = new ArrayList<Integer>();
9         if (nbre % 2 ==0) {
10            facteurs.add(2);
11            nbre = nbre / 2;
12        }
13        if (nbre > 1) {
14            facteurs.add(nbre);
15        }
16        return facteurs;
17    }
```

```
7 @   public static List<Integer> generate(int nbre) {
8         List <Integer> facteurs = new ArrayList<Integer>();
9         while (nbre % 2 ==0) {
10            facteurs.add(2);
11            nbre = nbre / 2;
12        }
13        if (nbre > 1) {
14            facteurs.add(nbre);
15        }
16        return facteurs;
17    }
```

Généralisation du nombre de valeurs composant la solution.

Exemples (7/7)

expression → fonction

ArabicToRoman

Utilisation d'une classe builder pour construire la solution, car elle manipule 2 éléments et 2 expressions

```java
public class ArabicToRoman {
    1 usage
    public static String[] symboles = {"X", "I"};
    3 usages
    public static int[] valeurs = {10, 1};

    7 usages
    public static String convert(int arabicNbre) {
        StringBuilder romanNbre = new StringBuilder();
        int reste = arabicNbre;

        for (int i = 0; i < valeurs.length; i++) {
            while (reste >= valeurs[i]) {
                romanNbre.append(symboles[i]);
                reste = reste - valeurs[i];
            }
        }
        return romanNbre.toString();
    }
}
```

avec

```java
static class ResultatBuilder {
    // Bonne pratique : classe qui regroupe les éléments modifiés pour
    //                  la production de la solution
    4 usages
    int reste;      // arabicNbre au départ, sans cesse décrémenté des valeurs
                    // du tableau valeurs.
    3 usages
    StringBuilder romanNbre = new StringBuilder();  // l'équivalent
                    // en chiffres romains, fabriqué par concaténation

    public void compute(String symbole, int valeur) {
        while (reste >= valeur) {
            romanNbre.append(symbole);
            reste = reste - valeur;
        }
    }
```

```java
public static String convert(int arabicNbre) {
    ResultatBuilder resultat = new ResultatBuilder(arabicNbre);

    // Décrémentation de arabicNbre des valeurs et concaténation du
    // symbole romain correspondant
    for (int i = 0; i < valeurs.length; i++) {
        resultat.compute(symboles[i], valeurs[i]);
    }

    return resultat.toString();
}
```

La solution est plus générale, car la production de la solution est encapsulée dans une fonction (ici une classe et une fonction)

Conclusion

Contextualiser le TDD dans les techniques d'Extreme Programming

- Le TDD est une méthode de développement logiciel dans lequel l'écriture de tests automatisés dirige l'écriture du code source.
- C'est une méthode très efficace pour livrer des logiciels **avec une suite de tests de non-régression**.
- La pratique du TDD est à la base du développement Agile qui met l'accent sur la livraison rapide et fréquente de composants logiciels fonctionnels, même lorsque la couverture fonctionnelle n'est pas complète (l'application ne fait pas encore tout ce qui est demandé)

Maîtriser la pratique du TDD

- Faire du TDD demande beaucoup de pratique pour acquérir de la maîtrise
- Cela requiert aussi une certaine discipline pour respecter le cycle
- Peuvent aider : coding dojo, kata, la programmation en binôme…

Chapitre 1

TDD
Test Driven Development

Ressource R5.AD : Qualité de Développement

Merci pour votre attention !