Validation

UNIVERSITÉ DE LORRAINE

Analyse et Conception de Logiciels

Horatiu CIRSTEA

Vérification & Validation

Validation

- Est-ce que l'énoncé du problème reflète le vrai problème ?
- Tous les besoins ont été pris en compte ?

Vérification

- Est-ce que l'implémentation correspond à la spécification ?
- Est-ce que le système fait ce qu'il doit faire ?

Techniques

- **Statiques**
  - sans exécuter les programmes
  - inspection du code
  - méthodes formelles: model-checkers, prouveurs automatiques, assistants à la preuves,…
- **Dynamiques**
  - en exécutant les programmes
  - tests, analyseurs de performance, validation de traces, …

Objectifs test logiciels

- **Assurer la confiance dans le logiciel**
  - Trouver des éventuels bugs
  - Analyser si utilisable dans des conditions opérationnelles

Testing is the process of executing a program with the intent of finding errors.

[Myers :1979] The Art of Software Testing

Exemple

```
@Test
public void testAverageRight() {
    population.addPerson(new Person("Alice", 20));
    population.addPerson(new Person("Bob", 30));

    assertSame(25, population.getAverageAge());
}
```

Tests

- **facile à appréhender**
- **proche du code**
- coûteux à mettre en œuvre
- ne garantît pas que le programme est exempt d'erreur

Program testing can be used to show the presence of bugs, but never to show their absence!

[Edsger Dijkstra]

Familles de tests

- **Boîte blanche**
  - on a accès à la structure du code
  - calcul de tous les chemins d'exécution possibles (branches, instructions, ...)

- **Boîte noire**
  - on ne connait pas les détails d'implémentation
  - on vérifie que l'implémentation fournit le résultat attendu par la spécification

Quels tests ?

Right BICEP est un ensemble de bonnes pratiques qui suggèrent des aspects de l'implémentation qui doivent être testés.

Principes Right-BICEP

- **Right**

est-ce que les résultats sont corrects?

- **B (Boundary)**

est-ce que les conditions aux limites sont correctes?

- **I (Inverse)**

est-ce qu'on peut vérifier la relation inverse?

- **C (Cross-check)**

est-ce qu'on peut vérifier le résultat autrement?

- **E (Error condition)**

est-ce qu'on peut forcer l'occurrence d'erreurs?

- **P (Performance)**

est-ce que les performances sont dans les limites acceptables?

Right

On doit pouvoir répondre à la question

Comment sait-on que le programme s'est exécuté correctement ?

Si pas de réponse : spécifications certainement vagues, incomplètes; écrire du code et des tests risque d'être inefficace/inutile.

- la notion de correction peut évoluer au cours du temps (en fonction de l'évolution des besoins, des spécifications,…)

BICEP - Boundary conditions

Un nombre très important de bugs apparaissent au niveau des **comportements extremes** (de la spécification)

- données inconsistantes/anachroniques
- données mal formattées
- données vides ou nulles
- données qui dépassent des valeurs raisonnables
- données dupliquées
- données (dés)ordonnées
- événements ne respectant pas l'ordre
- calculs qui produisent du overflow

BICEP - Boundary conditions

Un nombre très important de bugs apparaissent au niveau des **comportements extremes** (de la spécification)

- Nom de fichier: "!*W@V"
- Adresse mail: **jb@007**
- **0** ou **0:0** ou "" ou **null**
- Age personne : **189**
- Doublons dans une liste d'émargement
- Listes qui doivent (pas) être ordonnées
- Envoyer message avant de se connecter
- **calculs qui produisent du overflow**

Quels tests ?

```
class Population {
    private final List<Person> persons;

    public Population() {
        this.persons = new ArrayList<>();
    }

    public void addPerson(Person person) {
        ...
    }

    public int getAverageAge() {
        ...
    }
}
```

```
class Person {
    private final String name;
    private final int age;

    public String getName();

    public int getAge();
}
```

Quels tests ?

```
public class AppTest {
    Population population = new Population();

    @Test
    public void testAverageRight() {
        population.addPerson(new Person("Alice", 20));
        population.addPerson(new Person("Bob", 30));

        assertSame(25, population.getAverageAge());
    }
}
```

Quels tests ?

```
public class AppTest {
    Population population = new Population();

    @Test
    public void testAddNull() {
        assertThrows(IllegalArgumentException.class, () -> {
            population.addPerson(null);
        });
    }
}
```

AssertionFailedError: Expected java.lang.IllegalArgumentException to be thrown, but nothing was thrown.

```
class Population {
    private final List<Person> persons;

    public void addPerson(Person person) {
            persons.add(person);
    }
}
```

Quels tests ?

```
public class AppTest {
    Population population = new Population();

    @Test
    public void testAddNull() {
        assertThrows(IllegalArgumentException.class, () -> {
            population.addPerson(null);
        });
    }
}

    class Population {
        private final List<Person> persons;

        public void addPerson(Person person) {
            if (person == null) {
                throw new IllegalArgumentException("can't be null");
            }
            persons.add(person);
        }
    }
```

Quels tests ?

```
public class AppTest {
    Population population = new Population();

    @Test
    public void testAverageEmpty() {
        assertSame(0, population.getAverageAge());
    }
}
```

java.lang.ArithmeticException: / by zero

```
class Population {
    private final List<Person> persons;

    public int getAverageAge() {
        int sum = persons.stream().mapToInt(Person::getAge).sum();
        return sum / persons.size();
    }
}
```

Quels tests ?

```
public class AppTest {
    Population population = new Population();

    @Test
    public void testAverageEmpty() {
        assertSame(0, population.getAverageAge());
    }
}
```

```
class Population {
    private final List<Person> persons;

    public int getAverageAge() {
        if (persons.isEmpty()) {
            return 0;
        }
        int sum = persons.stream().mapToInt(Person::getAge).sum();
        return sum / persons.size();
    }
}
```

Quels tests ?

```
expected: <1073741824> but was: <-1073741824>
```

```
public class AppTest {
    Population population = new Population();

    @Test
    public void testAverageTooMany() {
        population.addPerson(new Person("Charlie", Integer.MAX_VALUE));
        population.addPerson(new Person("David", 1));
        assertEquals(1073741824, population.getAverageAge());
    }
}

    class Population {
        private final List<Person> persons;

        public int getAverageAge() {
            ...
            int sum = persons.stream().mapToInt(Person::getAge).sum();
            return sum / persons.size();
        }
    }
```

Quels tests ?

```
public class AppTest {
    Population population = new Population();

    @Test
    public void testAverageTooMany() {
        population.addPerson(new Person("Charlie", Integer.MAX_VALUE));
        population.addPerson(new Person("David", 1));
        assertEquals(1073741824, population.getAverageAge());
    }
}

    class Population {
        private final List<Person> persons;

        public int getAverageAge() {
            ...
            long sum = persons.stream().mapToLong(Person::getAge).sum();
            return (int) (sum / persons.size());
        }
    }
```

C.O.R.R.E.C.T.

- **Conformance**

  est-ce que les données respectent un format bien particulier (format de fichier, email, URL, …) ?

- **Ordering**

  est-ce qu'un un ordre pour les données est attendu ?

- **Range**

  est-ce que la donnée est dans le domaine attendu (int angle ∈ 0..359) ?

- **Reference**

  est-ce que le code utilise des éléments qu'il ne contrôle pas ?

C.O.R.R.E.C.T.

- **Existence**

est-ce que la donnée existe (non-zero, non-null, présente, … (**ex:** facture sans client) **?**

- **Cardinality**

que se passe-t-il quand avec les valeurs “remarquables” (**ex :** 0, 1, -1, MAX_VALUE, MIN_VALUE) **?**

- **Time**

est-ce que tout se passe au moment attendu ? est-ce que l’ordre (relatif/absolu) est respecté ?

B▮CEP - Inverse

Tester des méthodes en appliquant la ***relation logique inverse*** correspondant à l'opération (ex: tester la division en utilisant une multiplication, l'insertion dans une BDD en vérifiant l'existence,…)

```
public void testMySquareRoot() {
    double x = mySquareRoot(4.0);
    assertEquals( 4.0, x*x, 0.0001 );
}
```

BI**C**EP - Cross-check

Identifier des algorithmes équivalents qui permettent de vérifier le comportement de la méthode testée.

⇒ particulièrement utile quand on dispose d'une méthode de confiance mais relativement inefficace

```
public public void testSquareRootUsingStd() {
    double r = mySquareRoot(4.0);
    double rcc = Math.sqrt(4.0);
    assertEquals( rcc, r, 0.0001 );
}
```

BIC**E**P - Error conditions

Dans des conditions d'exécution réelles des erreurs qui pourraient survenir (disque plein, mémoire pleine, perte de connexion, crash, …)

⇒ forcer ce type d'erreur

Scénarios à considerer dans les tests:

- plus de mémoire disponible
- plus d'espace disque disponible
- problèmes avec l'horloge système
- système trop chargé
- palette couleur limitée

BICE**P** - Performance

Est-ce que le système réagit de façon non exponentielle à une montée en charge ?

⇒ L'objectif n'est pas de mesurer les performances mais de réaliser des tests pour s'assurer que la courbe des performance reste stable

Exemples

- vérifier que le temps est linéaire avec la taille de la liste quand on cherche un élément dans une liste
- vérifier que l'application continue de répondre en présence d'une surcharge du système

BICEP - Performance

```
public class AppTest {
    Population population = new Population();

    @Test
    public void testFindPersonPerf() {
        int small = 10000;
        for (int i = 0; i < small; i++) {
            population.addPerson(new Person("Person" + i, i));
        }
        long start = System.currentTimeMillis();
        population.findPerson("Person" + (small - 1));
        long end = System.currentTimeMillis();
        assertTrue(end-start < 1);

        long big = 100000000;
        for (int i = small; i < big; i++) {
            population.addPerson(new Person("Person" + i, i));
        }
        start = System.currentTimeMillis();
        population.findPerson("Person" + (small - 1));
        end = System.currentTimeMillis();
        assertTrue(end-start < 10);
    }
}
```

C.O.R.R.E.C.T.

- **Conformance**

est-ce que les données respectent un format bien particulier ?

- **Ordering**

est-ce qu'un un ordre pour les données est attendu ?

- **Range**

est-ce que la donnée est dans le domaine attendu ?

- **Reference**

est-ce que le code utilise des éléments qu'il ne contrôle pas ?

- **Existence**

est-ce que la donnée existe (non-zero, non-null, présente, …) ?

- **Cardinality**

que se passe-t-il quand avec les valeurs "remarquables" (ex : 0, 1, -1, …) ?

- **Time**

est-ce que tout se passe au moment attendu ? l'ordre est respecté ?

Conformance

Est-ce que le programme réagit correctement quand le format attendu n'est pas respecté?

```
public class AppTest {
    User user = new User("James Bond");

    @Test
    public void testBadMailAddress() {
        assertThrows(IllegalArgumentException.class, () -> {
            user.setMail("James.Bond@007");
        });
    }

    @Test
    public void testIncompleteMailAddress() {
        assertThrows(IllegalArgumentException.class, () -> {
            user.setMail("James.Bond@mi6");
        });
    }

}
```

Conformance

Est-ce que le résultat respecte le format attendu ?

```
public class AppTest {
    User user = new User(“James Bond”,...);

    @Test
    public void testGetMailAddress() {
       String mail = user.getMail();
       assertEquals(2, mail.split(“@“).length);
    }
}
```

Ordering

Quand le résultat dé

```
public static int max(int[] tab) {
   int index, max=0;
   for (index = 0; index < tab.length; index++)
      if (tab[index] > max)
         max = tab[index];
   return max;
}
```

```
public class AppTest {

    @Test
    public void testAscendingOrder() {
        assertEquals(99, Util.max(new int[]{3,7,99}));
    }

    @Test
    public void testDescendingOrder() {
        assertEquals(99, Util.max(new int[]{99,7,3}));
    }

    @Test
    public void testOnlyOne() {
        assertEquals(2, Util.max(new int[]{2}));
    }
}
```

Ordering

Quand le résultat dé

```
public static int max(int[] tab) {
  int index, max = Integer.MIN_VALUE;
  for (index = 0; index < tab.length; index++)
     if (tab[index] > max)
        max = tab[index];
  return max;
}
```

```
public class AppTest {

    @Test
    public void testOnlyPositives() {
        assertEquals(99, Util.max(new int[]{7,3,99}));
    }

     @Test
     public void testOnlyNegatives() {
         assertEquals(-3, Util.max(new int[]{-7,-3,-99}));
    }

     @Test
     public void testEmpty() {
        assertThrows(IllegalArgumentException.class, () -> {
             Util.max(new int[]{});
        });
     }
}
```

Range

Vérifier que les valeurs possibles ne dépassent pas les bornes attendues.

```
public class AppTest {

    @Test
    public void testNegative() {
       assertThrows(IllegalArgumentException.class, () -> {
            Azimuth b = Azimuth.getInstance(-1);
       });
    }
    @Test
    public void testTooBig() {
       assertThrows(IllegalArgumentException.class, () -> {
            Azimuth b = Azimuth.getInstance(Azimuth.MAX+1);
       });
    }
    @Test
    public void testAddTooMuch() {
        Azimuth b = Azimuth.getInstance(270);
        assertEquals(90, b.add(180).getValue());
    }
}
```

Range

Vérifier que les valeurs possibles ne dépassent pas les bornes attendues.

```
public class AppTest {

    @Test
    public void testAddTooMuch() {
        Azimuth b = Azimuth.getInstance(270);
        assertEquals(90, b.add(180).getValue());
    }

    @Test
    public void testAddTooMuch() {
        Azimuth b = Azimuth.getInstance(90);
        b.add(-180);
        assertEquals(270, b.getValue());
    }

}
```

Reference

Le code dépend/fait référence à quelque chose d'externe (dépendances externes, état objet, …) qui n'est pas sous le contrôle direct du code (de la méthode testée) ?

Si des suppositions sont faite par rapport à

- l'état de la classe correspondante,
- l'état d'autres objets,
- l'état global,
- …

il faut s'assurer que le comportement quand les suppositions ne sont pas satisfaites se comporte comme attendu.

Reference

```
@Test
public void remainsInDriveAfterAcceleration() {
    transmission.shift(Gear.DRIVE);
    car.accelerateTo(35);
    assertThat(transmission.getGear(), equalTo(Gear.DRIVE));
}

@Test
public void ignoresShiftToParkWhileInDrive() {
    transmission.shift(Gear.DRIVE);
    car.accelerateTo(30);
    transmission.shift(Gear.PARK);
    assertThat(transmission.getGear(), equalTo(Gear.DRIVE));
}

@Test
public void allowsShiftToParkWhenNotMoving() {
    transmission.shift(Gear.DRIVE);
    car.accelerateTo(30);
    car.brakeToStop();
    transmission.shift(Gear.PARK);
    assertThat(transmission.getGear(), equalTo(Gear.PARK));
}
```

Pragmatic Unit Testing in Java 8 with JUnit
Jeff Langr, with Andy Hunt and Dave Thomas

Reference

Que se passe-t-il quand un certain élément (argument, fichier, connexion réseaux, …) n'existe pas?

Pour les arguments d'une méthode que se passe-t-il si

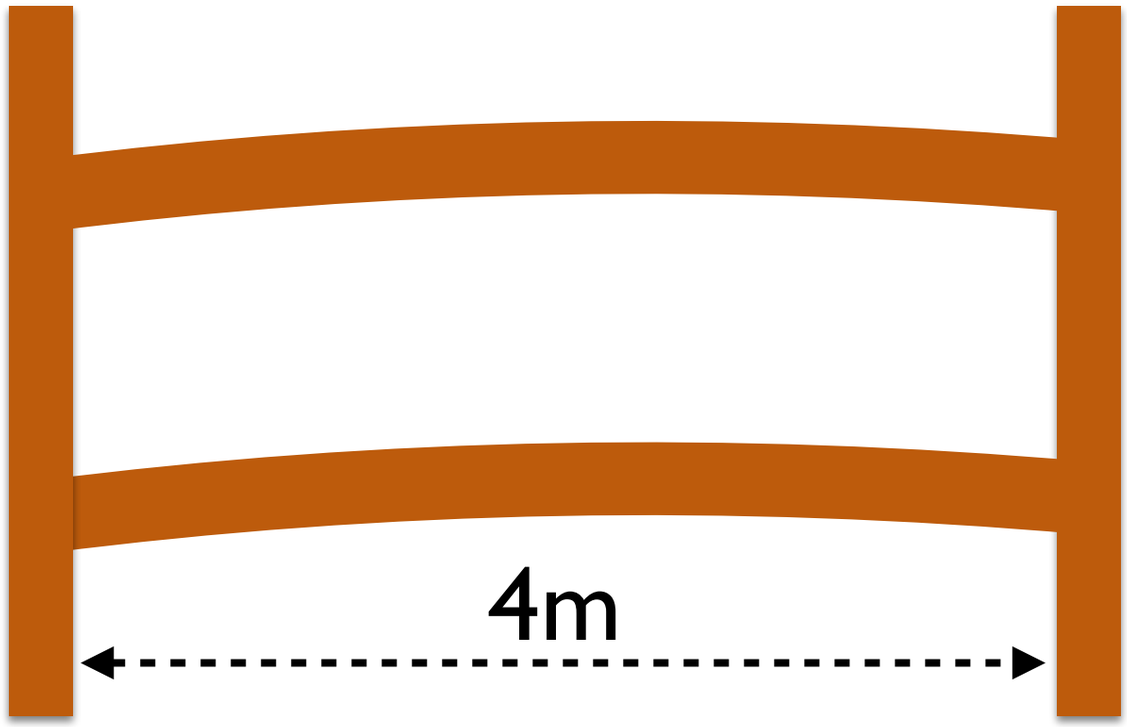
- la valeur reçues est null,
- la valeur reçues est 0,
- la valeur reçues est vide,
- …

Reference

```
public class AppTest {
    User user = new User(“James Bond”);

    @Test
    public void testBadMailAddress() {
        assertThrows(EmptyMailException.class, () -> {
            user.setMail(“”);
        });
    }

    @Test
    public void testIncompleteMailAddress() {
        assertThrows(NullMailException.class, () -> {
            user.setMail(null);
        });
    }

}
```

Cardinality

Une clôture droite de 16m de long est construite en plaçant un poteau à chaque extrémité, puis en plaçant des poteaux entre eux de manière à ce que les poteaux consécutifs soient espacés de 4m. Combien de poteaux sont nécessaires pour construire la clôture ?



fencepost errors

Cardinality

Est-ce que le nombre d'éléments correspond à ce qui est attendu ?

La règle 0-1-n (les cas les plus importants):

- 0 éléments,
- 1 élément,
- plusieurs éléments,

Idée: *si ca marche pour qqch > 1, il y a des chances que ca marche pour toutes les valeurs > 1*.

Cardinality

Est-ce que le nombre d'éléments correspond à ce qui est attendu ?

```
@BeforeEach
public void setUp() {
    jb = new User(“James Bond”, "jb@mi6");
    vv = new User(“Vincent Vega”, "vv@com");
}

@Test
public void testGetEmpty() {
    assertEquals(0, jb.getFriends().size());
}

@Test
public void testAddOneFriend() {
    jb.addFriend(vv);
    Collection<User> friends = jb.getFriends();
    assertEquals(1, friends.size());
    assertTrue(friends.contains(vv));
}
```

Cardinality

```
@Test
public void testAddOneFriendTwice() {
     jb.addFriend(vv); jb.addFriend(vv);
     List<User> friends = jb.getFriends();
     assertEquals(1, friends.size());
     assertTrue(friends.contains(vv));
}

@Test
public void testAddLastFriend() {
     jb.addFriend(vv); jb.addFriend(ap);
     List<User> friends = jb.getFriends();
     assertEquals(2, friends.size());
     assertTrue(friends.contains(vv));
     assertTrue(friends.contains(ap));
}

@Test
public void testAddLastFriend() {
     jb.addFriend(vv); jb.addFriend(ap);
     List<User> friends = jb.getFriends();
     jb.addFriend(mh);
     assertEquals(friends, jb.getFriends()); // ordered comparison!
}
```

Time

Temps relatif

- Tester le comportement quand une méthode est appelée quand l'appel d'une autre méthode était attendu avant
  - `logout()` avant `login()`
  - `close()` (ou `read()`) avant `open()`
- Tester le comportement quand des timeouts sont possibles
  - ressource attendue disponible après un certain temps
  - ressource attendue jamais disponible

Time

Temps relatif

- Tester le comportement quand une méthode est appelée quand l'appel d'une autre méthode était attendu avant
  - `logout()` avant `login()`
  - `close()` (ou `read()`) avant `open()`
- Tester le comportement quand des timeouts sont possibles
  - ressource attendue disponible après un certain temps
  - ressource attendue jamais disponible

Time

Temps absolu

- Tester le comportement du code qui dépend de l'heure exacte
  - avant quelle heure il est 30 minutes après 2:45 ?

Concurrence

- Que se passe-t-il si une ressource est accédée par différents threads/process ?

Organiser les tests

- ARRANGE

  Avant de lancer le code dont on veut tester il faut s'assurer que le système est dans un état qui permet d'exécuter le code (créer des objets, appeler d'autres API, …)

- ACT

  Exécuter le code dont on veut tester

- ASSERT

  Vérifier que le code exécuté s'est comporté comme attendu

- AFTER

  Libérer les éventuelles ressources allouées

Organiser les tests

```
@Test
public void testAverageRight() {
    population = new Population();
    alice = new Person("Alice", 20);
    bob = new Person("Bob", 30);

    population.addPerson(alice);
    population.addPerson(bob);

    assertSame(25, population.getAverageAge());

    population = null;
    alice = null;
    bob = null;
}
```

A

A

A

A

Organiser les tests

```
@BeforeEach
public void setUp() {
    population = new Population();
    alice = new Person("Alice", 20);
    bob = new Person("Bob", 30);
}
@Test
public void testAverageRight() {
    population.addPerson(alice);
    population.addPerson(bob);
    assertSame(25, population.getAverageAge());
}
@AfterEach
public void tearDown() {
    population = null;
    alice = null;
    bob = null;
}
```

A

A

A

A

Bien choisir les noms

- doingSomeOperationGeneratesSomeResult

  `addNullPersonRaisesException`

- someResultOccursUnderSomeCondition

  `raiseExceptionWhenAddingNullPerson`

- givenSomeContextWhenDoingSomeBehaviorThenSomeResultOccurs

  `forEmptyPopulationAverageIsZero`

- whenDoingSomeBehaviorSomeResultOccurs(doingSomeOperationGeneratesSomeResult)

  `addAndRemovePersonLeavesUnchanged`

JUnit

JUnit 3.x

- héritage

JUnit 4.x

- annotations

JUnit 5.x

- flexibilité

Imports (JUnit4 vs JUnit5)

```
import org.junit.Test;
```

- `org.junit.jupiter.api.Test;`

```
import org.junit.Assert.*;
```

- `org.junit.jupiter.api.Assertions.*;`

Annotations (parameters ⇒ functions)

```
@Test(expected = ErreurApp.class)
```

- `assertThrows(ErreurApp.class, () -> {…});`

```
@Test(timeout = 10)
```

- `assertTimeout(Duration.ofMillis(10),`
  `() -> {…});`

Annotations

- `@Before`
  - `@BeforeEach`
- `@After`
  - `@AfterEach`
- `@BeforeClass`
  - `@BeforeAll`
- `@AfterClass`
  - `@AfterAll`
- `@Ignore`
  - `@Disabled`

Imports

```
import org.junit.Test;
```

- `org.junit.jupiter.api.Test;`

```
import org.junit.Assert.*;
```

- `org.junit.jupiter.api.Assertions.*;`

Annotations (parameters ⇒ functions)

```
@Test(expected = ErreurApp.class)
```

- `assertThrows(ErreurApp.class, () -> {…});`

```
@Test(timeout = 10)
```

- `assertTimeout(Duration.ofMillis(10),`
  `() -> {…});`

Asserts

- utilisés pour déterminer si une méthode testée fonctionne correctement ou non
- enregistrent les échecs (lorsque l'assertion est fausse) ou les erreurs (lorsqu'une exception inattendue se produit)

```
assertSame(“Average for 2 persons”,
                25, population.getAverageAge());
```

-
```
assertSame(25, population.getAverageAge()
                “Average for 2 persons”,);
```

Asserts

- `assertTrue(condition);`
- `assertFalse(condition);`
- `assertSame(expected, actual); // ==`
- `assertNotSame(expected, actual); // !=`
- `assertEquals(expected, actual);`
- `assertNull(actual);`
- `assertNotNull(actual);`
- `assertArrayEquals(expected, actual);`

Asserts

- `assertIterableEquals(expected, actual);`
- `assertLinesMatch(expected, actual);`
- `assertThrows(expectedType, executable);`
- `assertTimeout(timeout, supplier);`
- `assertAll(String heading,`
  `Executable… executables);`

```
@Test
public void testAddOneFriend() {
    jb.addFriend(vv);
    Collection<User> friends = jb.getFriends();
    assertAll("Friends",
        () -> assertEquals(1, friends.size()),
        () -> assertTrue(friends.contains(vv)));
}
```

Assumptions

- utilisés pour déterminer si certaines conditions sont satisfaites avant de vérifier les assertions (sinon, le test est sauté)

```
Assume.assumeThat(jb.getFriends().size() == 0);
```

- `Assumtions.assumeThat(jb.getFriends().size() == 0);`

-
```
Assumtions.assumingThat(jb.getFriends().size() == 0,
 () -> {
    jb.addFriend(vv);
    Collection<User> friends = jb.getFriends();
    assertEquals(1, friends.size());
    assertTrue(friends.contains(vv));
 });
```

New in JUnit 5

- tests nommés

```
@DisplayName("Cardinality tests")
public class CardinalityTest {
    @Test
    @DisplayName("Test size when no friends")
    public void testGetEmpty() {
```

New in JUnit 5

- tests imbriqués

```
@DisplayName("Tests for User class")
public class NestedTest {
    User jb, vv, ap, mh;
    @Test
    @DisplayName("Can be instantiated (with no friends)")
    public void testGetEmpty() {
          jb = new User("jb", “jb@mi6");
          assertEquals(0, jb.getFriends().size());
    }
    @Nested
    @DisplayName(“Tests for User class")
    public class TestFriends {
        @BeforeEach
        public void setUp() {...}
        @Test
        public void testAddOneFriendTwice() {
            jb.addFriend(vv); jb.addFriend(vv);
            List<User> friends = jb.getFriends();
            assertEquals(1, friends.size());
            assertTrue(friends.contains(vv));
        }
     ...
```

New in JUnit 5

- tests paramétrés

```
@ParameterizedTest
@ValueSource(strings = {" ", "   ", “ABCD", " AB   CD "})
void isUpperWithEmptyStrings(String input) {
    assertTrue(Util.isUpper(input));
}
```

```
@ParameterizedTest
@EmptySource
void isUpperWithEmptyStrings(String input) {
    assertTrue(Util.isUpper(input));
}

@ParameterizedTest
@NullSource
void isUpperWithEmptyStrings(String input) {
    assertThrows(IllegalArgumentException.class,
                 () -> Util.isUpper(input) );
}
```

- Sources

```
short (shorts)
byte (bytes)
int (ints)
long (longs)
float (floats)
double (doubles)
char (chars)
String (strings)
Class (classes)
```

(Anti-)patterns de test

- Un test contient toujours au moins un assert
  - l'assert peut concerner une exception
  - un test qui affiche juste le résultat est inutile
- Un test doit être relativement simple/court
  - facile à comprendre
  - facile d'identifier les causes des erreurs détectées
  - moins de risque d'erreur dans les tests
  - ne pas vérifier plusieurs propriétés dans le même test

(Anti-)patterns de test

```
@Test
public void testLargestRight() {
        MyArray a = new MyArray();

        a.setTab(new int[]{17,0,9});
        assertSame(17, a.largest());
        a.setTab(new int[]{0,56,0,56});
        assertSame(56, a.largest());
}
```

(Anti-)patterns de test

```
@Test
public void testLargestSimple() {
    MyArray a = new MyArray();
    a.setTab(new int[]{17,0,9});
    assertSame(17, a.largest());
}

@Test
public void testLargestDouble() {
    MyArray a = new MyArray();
    a.setTab(new int[]{0,56,0,56});
    assertSame(56, a.largest());
}
```

(Anti-)patterns de test

- Utiliser l'assert adapté à la propriété à tester

```
assertTrue(17 == a.largest());
```

```
assertSame(17, a.largest());
```

- Il ne faut pas capturer les exceptions

```
@Test
public void testAverageRight() {
    population.addPerson(new Person("Alice", 20));
    population.addPerson(new Person("Bob", 30));

    try {
        assertSame(25, population.getAverageAge());
    } catch (Exception ex) {}
}
```