Test dynamique

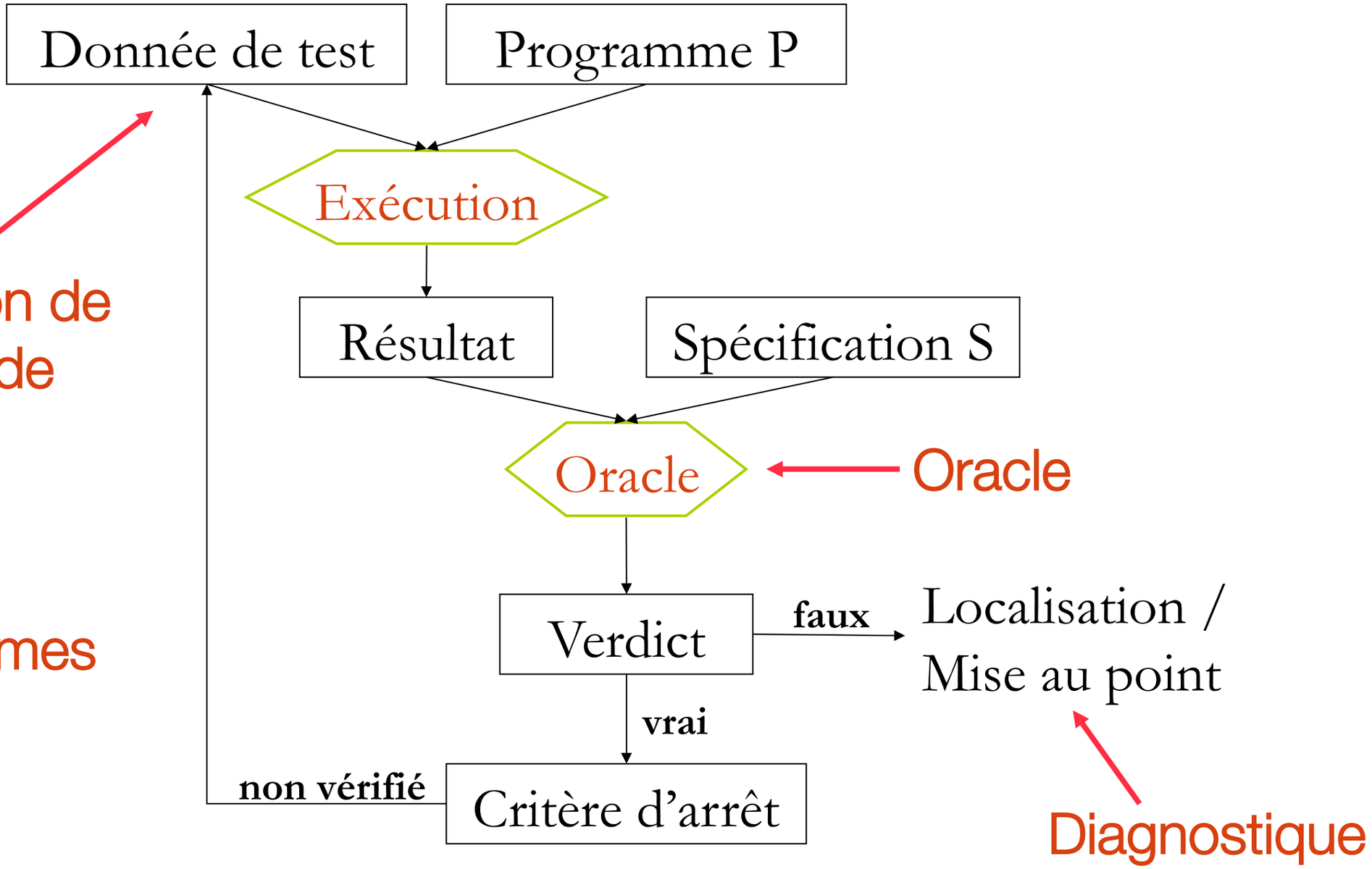
Plan

1. Le test dynamique
    1. Définition
    2. Test fonctionnel
    3. Test structurel

Test dynamique

Exécuter le programme avec un ensemble de valeurs en entrée dans le but de révéler des erreurs dans l'implantation

Le test dynamique : processus



Génération de données de test

Problèmes

Diagnostique

Le test dynamique

- Soit D le domaine d'entrée d'un programme P spécifié par S, on voudrait pouvoir dire
  - Soit D le domaine de P: ∀ x∈D P(x) = S(x)

- Test exhaustif impossible dans la plupart des cas
  - Domaine D trop grand, voire infini
  - Trop long et coûteux

Le test dynamique

- On cherche alors un ensemble de données de test T tel que
  - $T \subset D$
    - si $\forall x \in T\ P(x) = S(x)$ alors $\forall x \in D\ P(x) = S(x)$
- Critère d'arrêt pour la génération de données de test
  - {données de test} = T

Génération de test

- Génération déterministe
  - « à la main »
- Génération automatique aléatoire
- Génération automatique aléatoire contrainte
  - mutation
  - test statistique
- Génération automatique guidée par les contraintes
- Génération de test avancée (à partir des exigences, de modèles : *Model Based Testing*)

Génération de test

- Reste à savoir quand on a suffisamment testé
  - critères de test structurels, fonctionnels
  - analyse de mutation
- Choisir le bon niveau pour le test

La notion de couverture et de critère d'arrêt

- Critère d'arrêt = conditions objectives, mesurables, qui font qu'on peut considérer qu'une série de tests est suffisante
- Test fonctionnel (boîte noire)
  - Pour arrêter les tests, on se base sur des critères externes
    - Couverture des exigences
    - Couverture des fonctions des interfaces
    - Taux de défaillances par jour inférieurs à un seuil
- Test structurel (boîte blanche)
  - Pour arrêter les tests, on se base sur des critères internes
    - Couverture de la structure du système

      Chaque composant, chaque dépendance entre composants
    - Couverture du code
- Le test structurel permet d'être plus précis dans l'élaboration du critère d'arrêt des tests

Oracle

- Fonction qui évalue le résultat d'un cas de test
- Plus formellement
  - soit un programme P: Dom(P) → Ran(P)
  - une spécification S: Dom(P) → Ran(P)
  - une donnée de test $X \in Dom(P)$
  - oracle O: $Dom(P) \times Ran(P) \rightarrow bool$

$$O(X, P(X)) = true \text{ iff } P(X) = S(X)$$

- Problème : comment comparer P(X) et S(X)
  - plus S est formalisé plus on peut automatiser

Oracle

- Oracle manuel: on « regarde » le résultat et un humain évalue s'il est bon
- Construire le résultat exactement attendu
  - Comparer le résultat obtenu avec le résultat construit (diff)
- Assertions
  - dans le code (programmation défensive)
  - aux interfaces (design by contract)

    set_current_node (cnode: Node)

    pre : cnode != null

    post : currentNode = cnode
  - dans les cas de test (par ex. JUnit)
- …

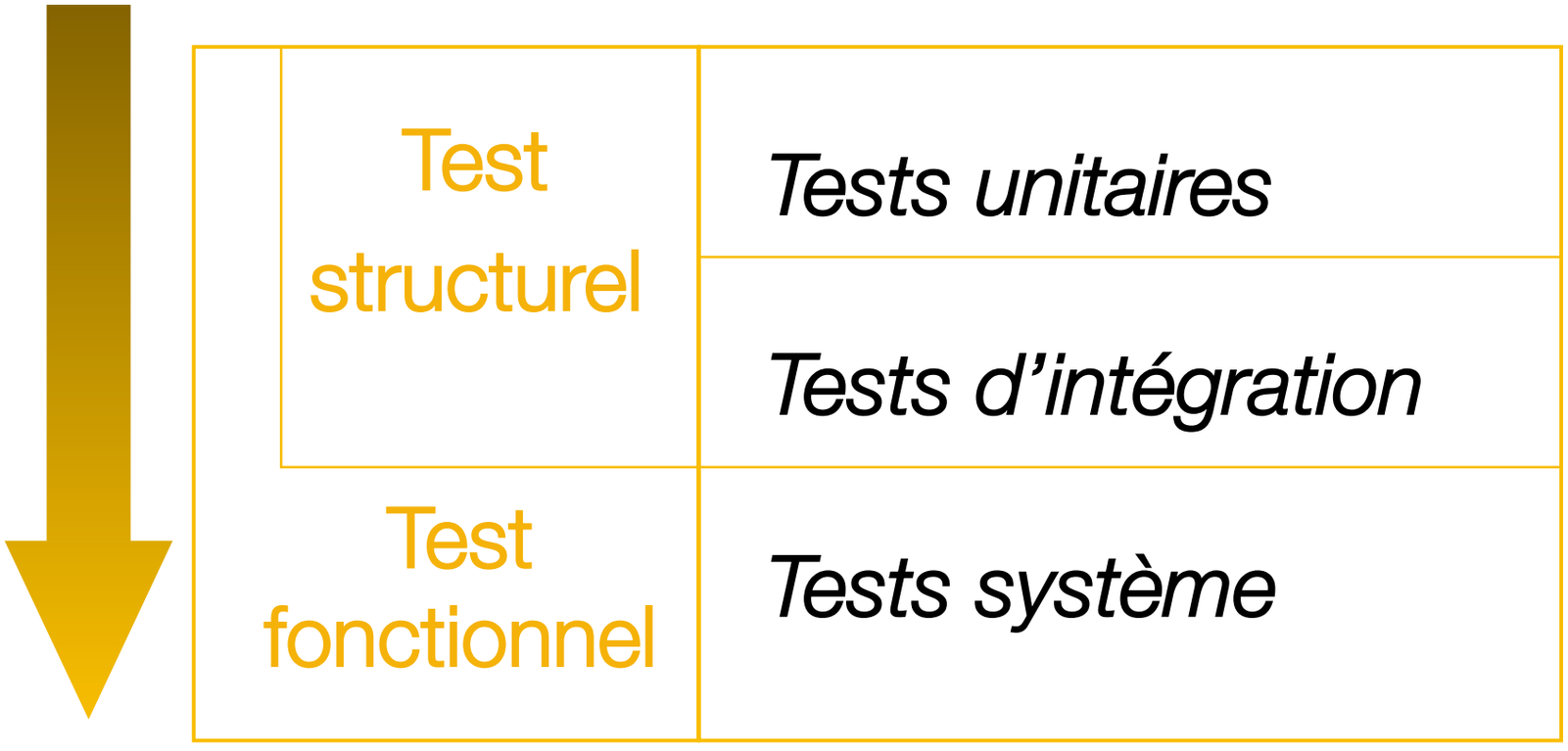
Test fonctionnel

- Spécification formelle
  - Modèle B, Z
  - Automate, système de transitions
- Description en langage naturel
- UML
  - Use cases
  - Diagramme de classes (+ contrats)
  - Machines à états / diagramme de séquence

Test structurel

- A partir d'un modèle du code
  - modèle de contrôle (conditionnelles, boucles...)
  - modèle de données
    - modèle de flot de données (définition, utilisation...)
- Utilisation importante des parcours de graphes
  - critères basés sur la couverture du code

Etapes et hiérarchisation des tests

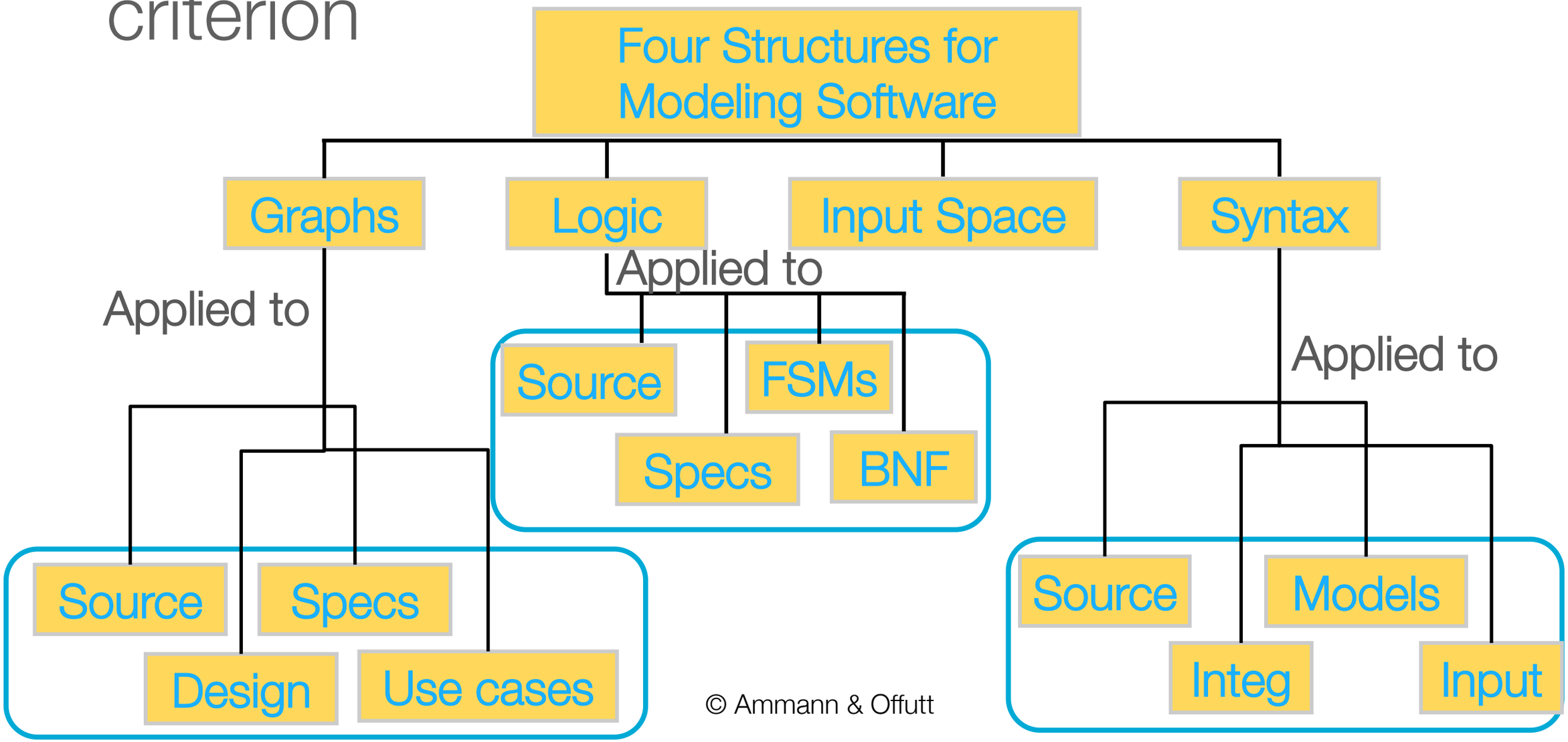


Dynamic testing in brief

- Testers run the program to reveal the presence of errors
- 4 main activities
  - TEST CASE DESIGN
    - select data and scenario to run the program
  - TEST ORACLE DESIGN
    - express properties that must hold on these runs
  - TEST RUN
    - set the exectution environment for test run
  - STOPPING CRITERION
    - ideally when there are no more errors
    - approximation, w.r.t coverage criteria

Test case qualification

- extract a model of a software artifact
- define metrics on the model: coverage criterion



Test automation

- Some tools exist to automate the following tasks
  - data/scenario generation
  - oracle checking
  - test case execution
  - coverage checking

=> + details in the specification
+ can be automated

JUNIT

Test unitaire OO

- Tester une unité isolée du reste du système
- L'unité est la classe
  - Test unitaire = test d'une classe
- Test du point de vue client
  - les cas de tests appellent les méthodes depuis l'extérieur
  - on ne peut tester que ce qui est public
  - Le test d'une classe se fait à partir d'une classe extérieure
- Au moins un cas de test par méthode publique
- Il faut choisir un ordre pour le test
  - quelles méthodes sont interdépendantes?

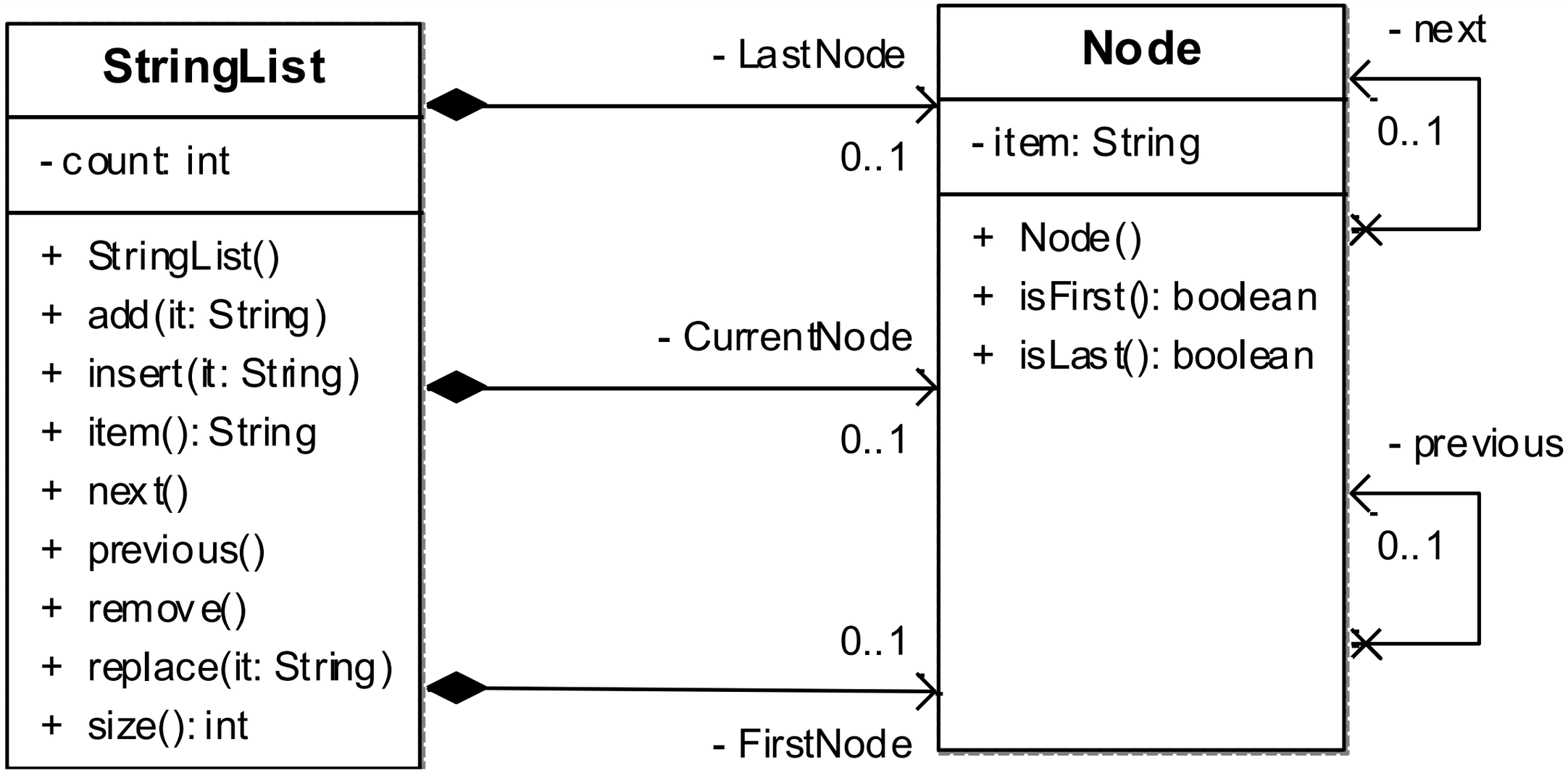
Test unitaire OO

- Problème pour l'oracle :
  - Encapsulation : les attributs sont souvent privés
  - Difficile de récupérer l'état d'un objet
- Penser au test au moment du développement (« testabilité »)
  - prévoir des accesseurs en lecture sur les attributs privés
  - des méthodes pour accéder à l'état de l'objet

Cas de test unitaire

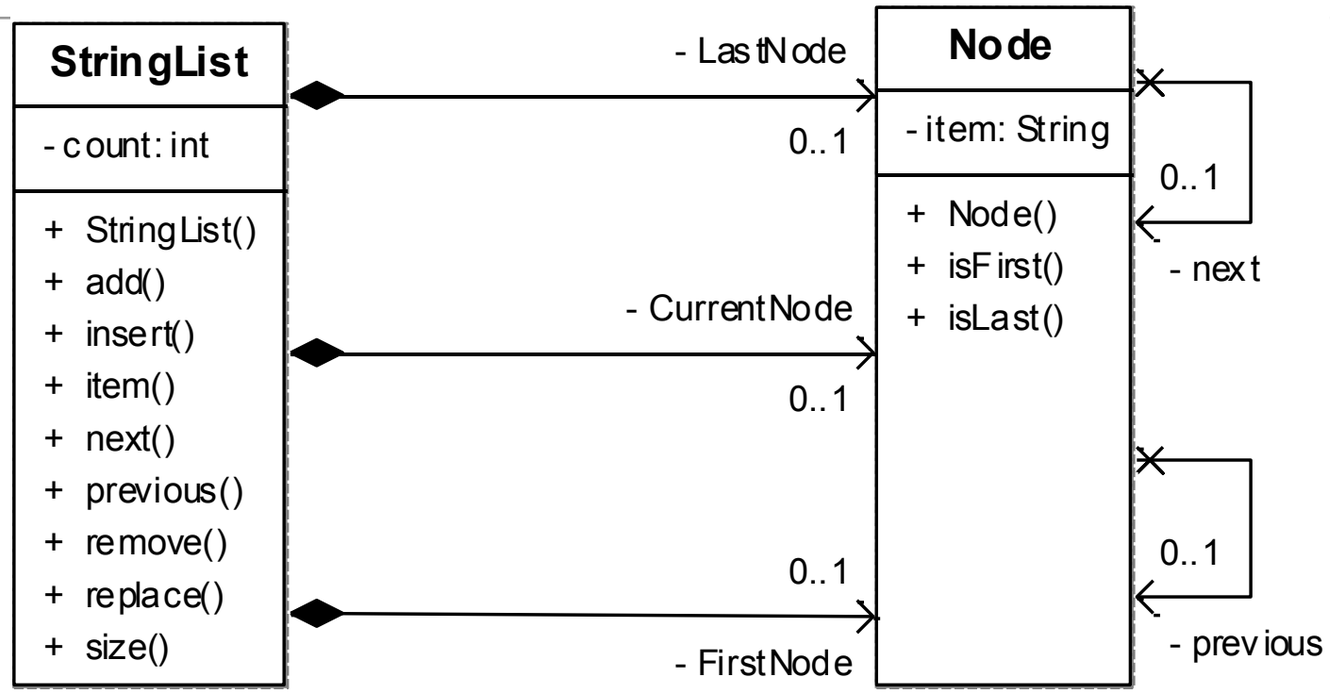
- Cas de test = une méthode
- Corps de la méthode
  - Configuration initiale
  - Une donnée de test
    - un ou plusieurs paramètres pour appeler la méthode testée
  - Un oracle
    - il faut construire le résultat attendu
    - ou vérifier des propriétés sur le résultat obtenu
- Une classe de test pour une classe testée
  - Regroupe les cas de test
  - Il peut y avoir plusieurs classes de test pour une classe testée

Exemple : test de StringList



- Créer une classe de test qui manipule des instances de la classe StringList
- Au moins 9 cas de test (1 par méthode publique)
- Pas accès aux attributs privés : count, LastNode, CurrentNode, FirstNode

Exemple : insertion dans une liste



spécification du cas de test

```
//first test for insert: call insert
//and see if current element is the
//one that's been inserted
public void testInsert1(){
```

initialisation

```
    list.add("first");
    list.add("second");
    list.insert("third");
```

appel avec donnée de test

```
    assertTrue(list.size()==3);
```

oracle

```
    assertTrue(list.item()=="third");
}
```

Plan

1. Introduction au test unitaire
2. JUnit, une bibliothèque Java de test unitaire
3. Test-driven development

JUnit

- Origine
  - Xtreme Programming (test-first development)
  - framework de test écrit en Java par E. Gamma et K. Beck
  - open source: www.junit.org
- Objectifs
  - test d'applications en Java
  - faciliter la création des tests
  - tests de non régression

JUnit : un framework

« Un framework est un ensemble de classes et de collaborations entre les instances de ces classes. »

http://st-www.cs.uiuc.edu/users/johnson/frameworks.html

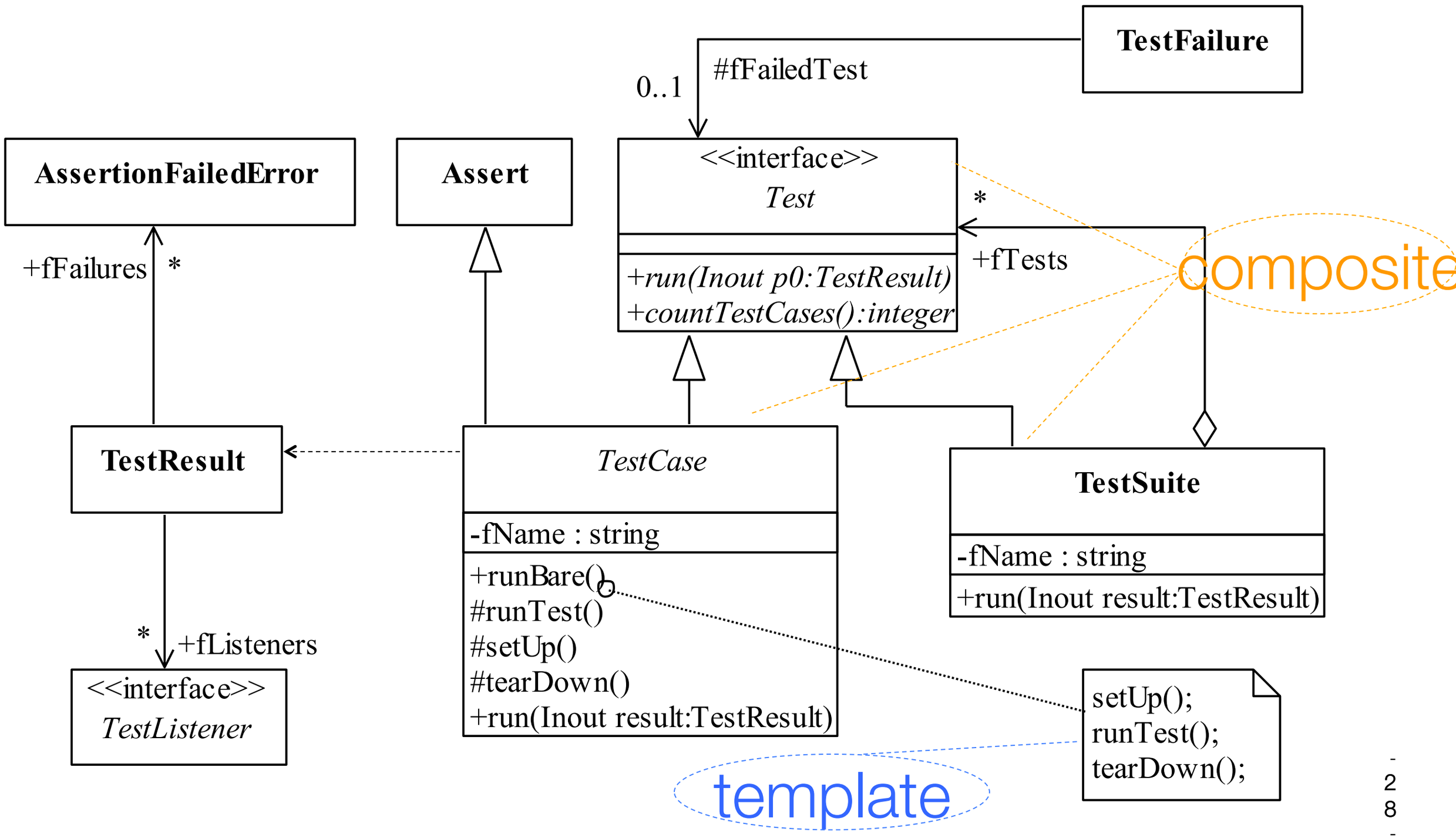
JUnit : un framework

- Le source d'un framework est disponible
- Ne s'utilise pas directement: il se spécialise

  Ex: pour créer un cas de test on hérite de la classe TestCase

➔ Un framework peut être vu comme un programme à « trous » qui offre la partie commune des traitements et chaque utilisateur le spécialise pour son cas particulier.

JUnit : un framework



JUnit v3 : codage (1/5)

- Organisation du code des tests
  - cas de Test: TestCase
    - setUp() et tearDown()
    - les méthodes de test
  - suite de Test: TestSuite
    - Méthodes de test
    - Cas de test
    - Suite de Test

Exemple : une classe StringList

```
public class StringList {

	private Node currentNode;
	private Node firstNode;
	private int count;

	public StringList(){
		count=0;
	}

	public String item(){
		return this.currentNode.getItem();
	}

	public void next(){...}
	public void previous(){ ... }
	public int size(){...}
	public void add (String it){...}
	public void replace (String it){...}

}
```

JUnit v3 : codage (2/5)

- Codage d'un « TestCase »:
  - déclaration de la classe
  - Hérite de `JUnit.framework.TestCase`

```
public class TestStringList extends TestCase {
        //déclaration des instances
        private StringList list;

        //setUp()
        //tearDown()
        //méthodes de test
        //main()
}
```

JUnit v3 : codage (3/5)

- la méthode setUp:

```
//appelée avant chaque cas de test
//permet de factoriser la construction de « l'état du monde »

protected void setUp() throws Exception {
        list = new StringList();
}
```

- la méthode tearDown:

```
//appelée après chaque cas de test
//permet de défaire « l'état du monde »

protected void tearDown() throws Exception {
        super.tearDown();
}
```

JUnit v3 : codage (4/5)

- les méthodes de test:

```
//test add two elements
public void testAdd2(){
        list.add("first");
        list.add("second");
        assertTrue(list.size()==2);
        assertTrue(list.item()=="second");
}
```

- caractéristiques:
  - nom préfixé par « test »
  - publique, type de retour **`void`**
  - contient des assertions (définie l'oracle)

Les assertions de JUnit

- assertTrue(...), assertFalse(...)
- assertEquals(Object, Object)
- assertSame(Object, Object)
- assertNull(...)
- assertEquals(double expected, double actual, double delta)

⇨ Voir la liste des méthodes static de la classe Assert

JUnit v3 : codage (5/5)

- regroupement des méthodes de test:

```
public static Test suite() {
    TestSuite suite = new TestSuite();
    suite.addTest(newTestedClass(“testAdd2()”);
    suite.addTest(TestStringList.suite());
    suite.addTestSuite(TestStringList.class)
    return suite;
}
```

JUnit v3 : lancement des tests

- On utilise un TestRunner graphique ou textuel.
  - graphique : « *keep the bar green to keep the code clean* »
    `java junit.swingui.TestRunner`
  - textuel : affichage des résultats sur la console

```
junit.textui.TestRunner.run(<ma suite de
 test>);

// main

junit.textui.TestRunner <ma suite de Test>

// ligne de commande
```

JUnit v3 : détail d'implémentation

- Pour exécuter une suite de tests, JUnit utilise l'introspection

```
public TestSuite (final Class theClass){
        …
        Method[] = theClass.getDeclaredMethods
        …
}
private boolean isTestMethod(Method m) {
        String name= m.getName();
        Class[] parameters= m.getParameterTypes();
        Class returnType= m.getReturnType();
        return parameters.length == 0 && name.startsWith("test") &&
returnType.equals(Void.TYPE);
         }
```

JUnit version 4

- Fonctionne avec Java 5+
- Utilisation intensive des annotations
- Plus de runner graphique (laissé au soin des IDEs)
- Paquetage `org.junit`
- Nouvelle architecture
- Tests paramétrés, timeouts, etc

JUnit v4 : classe et méthode de test

- Classe de test :
  - `import org.junit.Test;`
  - `import static org.junit.Assert.*;`
- Méthodes de test :
  - Nom de méthode quelconque
  - Annotation **`@Test`**
  - Publique, type de retour **`void`**
  - Pas de paramètre, peut lever une exception
  - Annotation **`@Test(expected = Class)`** pour indiquer l'exception attendue
  - Annotation **`@Ignore`** pour ignorer un test

JUnit v4 : classe et méthode de test

- Méthodes avec annotations `@Before` ou `@After`
  - `import org.junit.Before;, etc`
  - Publiques (et plus protected)
  - Exécutées avant/après chaque méthode de test
  - Possibilité d'annoter plusieurs méthodes (ordre d'exécution indéterminé)
- Méthodes avec annotations `@BeforeClass` et `@AfterClass`
  - Publiques et statiques
  - Exécutées avant (resp. après) la première (resp. dernière) méthode de test
  - Une seule méthode pour chaque annotation

JUnit v4 : suite de test

- utilisation des annotations
- utilisation d'un autre runner que celui par défaut, le `org.junit.runners.Suite`
- changer de runner : annotation `@RunWith(Class)`
- classe vide annotée `@RunWith(Suite.class)`
- pour indiquer comment former la suite de test : annotation `@SuiteClasses(Class[])`

```
@RunWith(Suite.class)
@SuiteClasses({TestSuitePartie1.class, TestSuitePartie2.class})
public class TestSuite {
}
```

JUnit v4 : test paramétrés

- Pour factoriser les données de test :
  - runner `org.junit.runners.Parameterized`
  - Données fournies par une méthode publique statique qui retourne une collection et est annotée par `@Parameters`
  - constructeur public pour la classe de test qui prend en paramètre les données et le résultat attendu
  - produit en croix des données de test et des méthodes de test

JUnit v4 : test paramétrés

```
@RunWith(Parameterized.class)
public class FibonacciTest {
    @Parameters
    public static Collection<Object[]> data() {
        return Arrays.asList(new Object[][] {
                {0, 0}, {1, 1}, {2, 1}, {3, 2}, {4, 3}, {5, 5},{6, 8}});
    }

    private int fInput;

    private int fExpected;

    public FibonacciTest(int input, int expected) {
        fInput= input;
        fExpected= expected;
    }

    @Test
    public void test() {
        assertEquals(fExpected, Fibonacci.compute(fInput));
    }
}
```

JUnit

- Permet de structurer les cas de test
  - cas de test / suite de test
- Permet de sauvegarder les cas de test
  - important pour la non régression
  - quand une classe évolue on ré-exécute les cas de test

JUnit

- www.junit.org
- Avantages
  - gratuit
  - simple
  - intégré à Eclipse
- Inconvénients
  - exploitation des résultats (pas d'historique...)
- Généralisation des concepts (XUnit)
  - www.testdriven.com

Ecrire 4 cas de test pour

```java
package test;

public class List {

	private int[] tab = new int[10];
	private int current = -1;

	public void add(int i) {
		current++;
		tab[current]=i;
	}

	public int length() {
		return current+1;
	}

	public int current() {
		return tab[current];
	}

	public int get(int i) {
		return tab[i];
	}

}
```

QUELQUES EXEMPLES

Beacoup de JUnit disponible

- De bons exemples
  - de tests de différents types de comportement
  - de checkers
- Et aussi des moins bons
  - The Free Ride / Piggyback
  - Happy Path
  - The Hidden Dependency
  - http://stackoverflow.com/questions/333682/unit-testing-anti-patterns-catalogue

Regular test case

```
// In
org.apache.commons.math3.linear.RealVectorFormatAbstra
ctTest

    @Test
    public void testSimpleWithDecimals() {
        ArrayRealVector c = new ArrayRealVector(new
double[] {1.23, 1.43, 1.63});
        String expected =
            "{1"     + getDecimalCharacter() +
            "23; 1" + getDecimalCharacter() +
            "43; 1" + getDecimalCharacter() +
            "63}";
        String actual = realVectorFormat.format(c);
        Assert.assertEquals(expected, actual);
    }
```

Well-designed test case

```
// In org.apache.commons.math3.linear.BlockFieldMatrixTest

    /** test copy functions */
    @Test
    public void testCopyFunctions() {
        Random r = new Random(666363289960021);
        BlockFieldMatrix<Fraction> m1 = createRandomMatrix(r,
47, 83);
        BlockFieldMatrix<Fraction> m2 = new
BlockFieldMatrix<Fraction>(m1.getData());
        Assert.assertEquals(m1, m2);
        BlockFieldMatrix<Fraction> m3 = new
BlockFieldMatrix<Fraction>(testData);
        BlockFieldMatrix<Fraction> m4 = new
BlockFieldMatrix<Fraction>(m3.getData());
        Assert.assertEquals(m3, m4);
    }
```

Strong test case

```
// In org.apache.commons.math3.random.UnitSphereRandomVectorGeneratorTest

    @Test
    public void test2DDistribution() {

        RandomGenerator rg = new JDKRandomGenerator();
        rg.setSeed(173992254321);
        UnitSphereRandomVectorGenerator generator = new UnitSphereRandomVectorGenerator(2, rg);

        // In 2D, angles with a given vector should be uniformly distributed
        int[] angleBuckets = new int[100];
        int steps = 1000000;
        for (int i = 0; i < steps; ++i) {
            final double[] v = generator.nextVector();
            Assert.assertEquals(2, v.length);
            Assert.assertEquals(1, length(v), 1e-10);
            // Compute angle formed with vector (1,0)
            // Cosine of angle is their dot product, because both are unit length
            // Dot product here is just the first element of the vector by construction
            final double angle = FastMath.acos(v[0]);
            final int bucket = (int) (angleBuckets.length * (angle / FastMath.PI));
            ++angleBuckets[bucket];
        }
        // Simplistic test for roughly even distribution
        final int expectedBucketSize = steps / angleBuckets.length;
        for (int bucket : angleBuckets) {
            Assert.assertTrue("Bucket count " + bucket + " vs expected " + expectedBucketSize,
                              FastMath.abs(expectedBucketSize - bucket) < 350);
        }
    }
```

Strong test data

```
//In org.apache.commons.math3.special.GammaTest

    /**
     * Reference data for the {@link Gamma#logGamma(double)} function. This data
     * was generated with the following <a
     * href="http://maxima.sourceforge.net/">Maxima</a> script.
     *
     * <pre>
     * kill(all);
     *
     * fpprec : 64;
     * gamln(x) := log(gamma(x));
     * x : append(makelist(bfloat(i / 8), i, 1, 80),
     *     [0.8b0, 1b2, 1b3, 1b4, 1b5, 1b6, 1b7, 1b8, 1b9, 1b10]);
     *
     * for i : 1 while i <= length(x) do
     *     print("{", float(x[i]), ",", float(gamln(x[i])), "},");
     * </pre>
     */
    private static final double[][] LOG_GAMMA_REF = {
        { 0.125 , 2.0194183575537960 },
        { 0.25 , 1.2880225246980770 },
        { 0.375 , .86307398227064750 },
        { 0.5 , .57236494292470010 },
        { 0.625 , .36082949548894020 },
        { 0.75 , .20328095143129540 },
        { 0.875 , .08585870722533433 },
        { 0.890625 , .07353860936979656 },
        ...124 more lines
    };
```

Strong checker

```
// org.apache.commons.math3.geometry.euclidean.twod.PolygonsSetTest
    private void checkVertices(Vector2D[][] rebuiltVertices,
                               Vector2D[][] vertices) {
        // each rebuilt vertex should be in a segment joining two original vertices
        for (int i = 0; i < rebuiltVertices.length; ++i) {
            for (int j = 0; j < rebuiltVertices[i].length; ++j) {
                boolean inSegment = false;
                Vector2D p = rebuiltVertices[i][j];
                for (int k = 0; k < vertices.length; ++k) {
                    Vector2D[] loop = vertices[k];
                    int length = loop.length;
                    for (int l = 0; (! inSegment) && (l < length); ++l) {
                        inSegment = checkInSegment(p, loop[l], loop[(l + 1) % length], 1.0e-10);
                    }
                }
                Assert.assertTrue(inSegment);
            }
        }
        // each original vertex should have a corresponding rebuilt vertex
        for (int k = 0; k < vertices.length; ++k) {
            for (int l = 0; l < vertices[k].length; ++l) {
                double min = Double.POSITIVE_INFINITY;
                for (int i = 0; i < rebuiltVertices.length; ++i) {
                    for (int j = 0; j < rebuiltVertices[i].length; ++j) {
                        min = FastMath.min(vertices[k][l].distance(rebuiltVertices[i][j]),
                                           min);
                    }
                }
                Assert.assertEquals(0.0, min, 1.0e-10)
            }}}
```

Interesting checker

```
// In org.apache.commons.math3.TestUtils

    /**
     * Verifies that the relative error in actual vs. expected is less than or
     * equal to relativeError.  If expected is infinite or NaN, actual must be
     * the same (NaN or infinity of the same sign).
     *
     * @param msg  message to return with failure
     * @param expected expected value
     * @param actual  observed value
     * @param relativeError  maximum allowable relative error
     */
    public static void assertRelativelyEquals(String msg, double expected,
            double actual, double relativeError) {
        if (Double.isNaN(expected)) {
            Assert.assertTrue(msg, Double.isNaN(actual));
        } else if (Double.isNaN(actual)) {
            Assert.assertTrue(msg, Double.isNaN(expected));
        } else if (Double.isInfinite(actual) || Double.isInfinite(expected)) {
            Assert.assertEquals(expected, actual, relativeError);
        } else if (expected == 0.0) {
            Assert.assertEquals(msg, actual, expected, relativeError);
        } else {
            double absError = FastMath.abs(expected) * relativeError;
            Assert.assertEquals(msg, expected, actual, absError);
        }
    }
```

Antipattern: Piggybacking

```
// In org.apache.commons.lang3.ArrayUtilsRemoveMultipleTest

    @Test
    public void testRemoveAllObjectArray() {
        Object[] array;
        array = ArrayUtils.removeAll(new Object[] { "a" }, 0);
        assertArrayEquals(ArrayUtils.EMPTY_OBJECT_ARRAY, array);
        assertEquals(Object.class, array.getClass().getComponentType());
        array = ArrayUtils.removeAll(new Object[] { "a", "b" }, 0, 1);
        assertArrayEquals(ArrayUtils.EMPTY_OBJECT_ARRAY, array);
        assertEquals(Object.class, array.getClass().getComponentType());
        array = ArrayUtils.removeAll(new Object[] { "a", "b", "c" }, 1, 2);
        assertArrayEquals(new Object[] { "a" }, array);
        assertEquals(Object.class, array.getClass().getComponentType());
        array = ArrayUtils.removeAll(new Object[] { "a", "b", "c", "d" }, 1, 2);
        assertArrayEquals(new Object[] { "a", "d" }, array);
        assertEquals(Object.class, array.getClass().getComponentType());
        array = ArrayUtils.removeAll(new Object[] { "a", "b", "c", "d" }, 0, 3);
        assertArrayEquals(new Object[] { "b", "c" }, array);
        assertEquals(Object.class, array.getClass().getComponentType());
        array = ArrayUtils.removeAll(new Object[] { "a", "b", "c", "d" }, 0, 1, 3);
        assertArrayEquals(new Object[] { "c" }, array);
        assertEquals(Object.class, array.getClass().getComponentType());
        array = ArrayUtils.removeAll(new Object[] { "a", "b", "c", "d", "e" }, 0, 1, 3);
        assertArrayEquals(new Object[] { "c", "e" }, array);
        assertEquals(Object.class, array.getClass().getComponentType());
        array = ArrayUtils.removeAll(new Object[] { "a", "b", "c", "d", "e" }, 0, 2, 4);
        assertArrayEquals(new Object[] { "b", "d" }, array);
        assertEquals(Object.class, array.getClass().getComponentType());

        ...
}
```

Antipattern: happy path

```
// In org.easymock.tests.CapturesMatcherTest

    @Test
    public void test() throws Exception {

        matcher.appendTo(buffer);
        assertEquals("capture(Nothing captured yet)", buffer.toString());

        assertTrue(matcher.matches(null));
        matcher.validateCapture();

        clearBuffer();
        matcher.appendTo(buffer);
        assertEquals("capture(null)", buffer.toString());

        assertTrue(matcher.matches("s"));

        matcher.validateCapture();

        clearBuffer();
        matcher.appendTo(buffer);
        assertEquals("capture([null, s])", buffer.toString());
}
```

Antipattern: useless assert

```
// In org.apache.commons.math3.genetics.RandomKeyTest

    @Test
    public void testRandomPermutation() {
        // never generate an invalid one
        for (int i=0; i<10; i++) {
            DummyRandomKey drk = new
DummyRandomKey(RandomKey.randomPermutation(20));
            Assert.assertNotNull(drk);
        }
    }
```

Flaky test

```
// In org.easymock.tests2
public class ThreadingTest {

    private static final int THREAD_COUNT = 10;

    @Test
    public void testThreadSafe() throws Throwable {

        final IMethods mock = createMock(IMethods.class);
        expect(mock.oneArg("test")).andReturn("result").times(THREAD_COUNT);
        replay(mock);

        final Callable<String> replay = new Callable<String>() {
            public String call() throws Exception {
                return mock.oneArg("test");
            }
        };
        final ExecutorService service = Executors.newFixedThreadPool(THREAD_COUNT);
        final List<Callable<String>> tasks = Collections.nCopies(THREAD_COUNT, replay);
        final List<Future<String>> results = service.invokeAll(tasks);
        for (final Future<String> future : results) {
            assertEquals("result", future.get());
        }
        verify(mock);
    }
}
```

Root cause

```
//In org.easymock.internal.ReplayState

    public Object invoke(final Invocation invocation)
                                                throws Throwable {

        behavior.checkThreadSafety();
        if (behavior.isThreadSafe()) {
            // If thread safe, synchronize the mock
            lock.lock();
            try {
                return invokeInner(invocation);
            }
        }

        return invokeInner(invocation);
    }
```

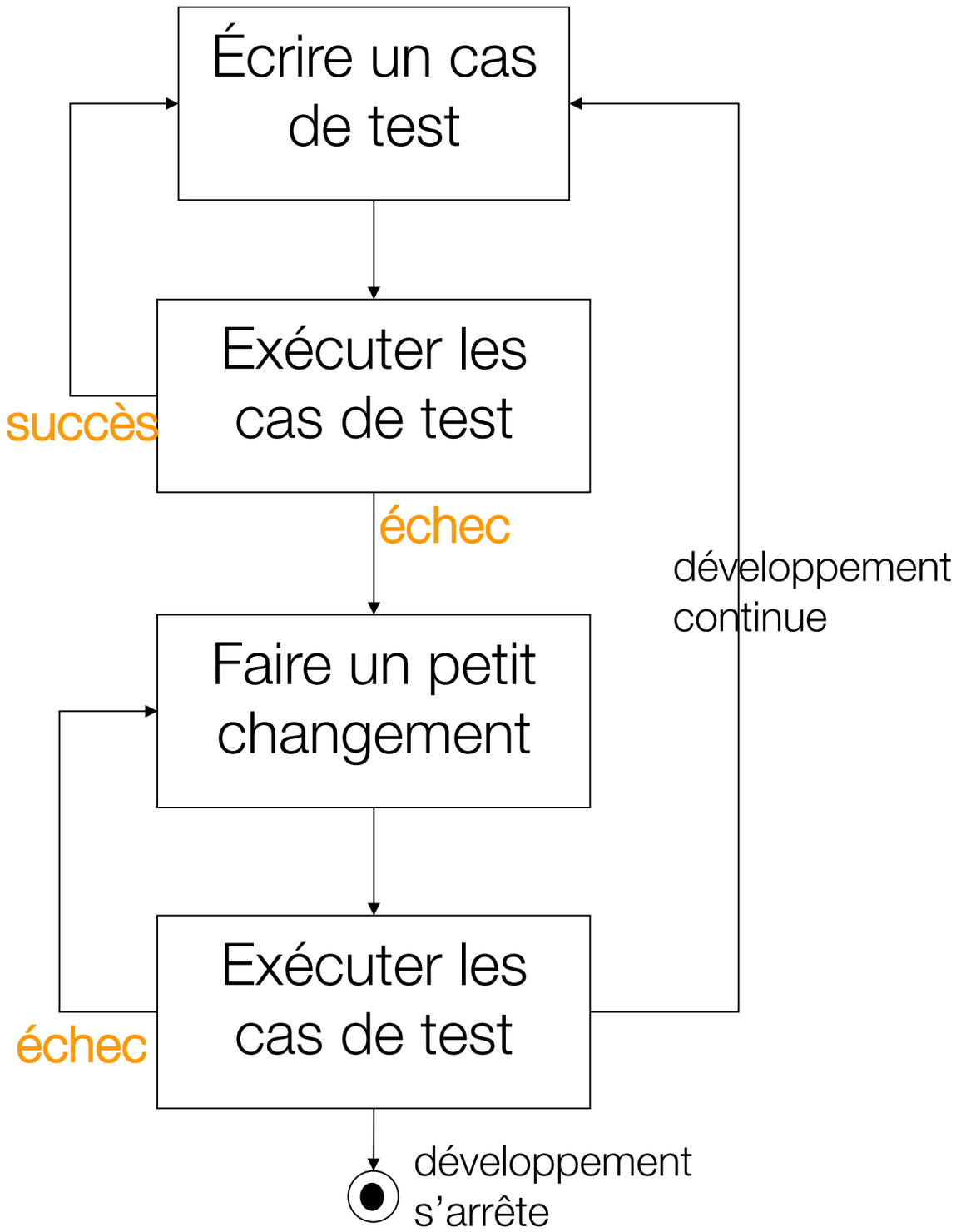
Plan

1. Introduction au test unitaire
2. JUnit, une bibliothèque Java de test unitaire
3. Test-driven development

Test-first development

- Xtreme programming
- Écrire les cas de test avant le programme
  - les cas de test décrivent ce que le programme doit faire
  - avant d'écrire le code, les cas de test échouent
  - quand on développe, les cas de test doivent passer
- Ensuite on développe la partie du programme qui fait passer le cas de test

Test-first development



Exemple : ajout dans une liste chainée

```
public void testAdd(){
        list.add("first");
        assertTrue(list.size()==1);
}
public void add (String it){
public void add (String it){}
        Node node = new Node();
        node.setItem(it);
        node.setNext(null);
        if (firstNode == null) {
                node.setPrevious(null);
                this.setFirstNode(node);
        }
        lastNode = node;
        this.setCurrentNode(node);
}
```

Test-first development

- Les cas de test servent de support à la documentation
  - des exemples d'utilisation du code
- Tester avec une intention précise
  - qu'est-ce qu'on teste?
  - pourquoi on le teste?
  - quand est-ce assez testé?
- Retours rapides sur la qualité du code
  - itérations courtes dans le cycle de développement
  - on exécute le code tout de suite (avant même de l'avoir écrit)
  - On ne code que quand un test a échoué

Test-first development

- Les cas de test spécifient ce que le programme doit faire mais pas comment
  - il faut associer TDD à des refactorings fréquents
    - revoir la structure du code
    - ne pas oublier la conception
- Grande importance du test de non-régression
  - quand on refactore les cas de test qui passaient doivent continuer à passer

Test-first development

- Importance d'un environnement pour l'exécution automatique des tests
  - pouvoir exprimer facilement des test unitaires
  - pouvoir les exécuter rapidement
  - pouvoir réexécuter les cas de test
- JUnit + d'autres outils pour automatiser le test et permettre TDD