Test fonctionnel
Test unitaire

JUnit

Jean-Marie Mottu jean-marie.mottu@univ-nantes.fr

Lanoix – Le Traon – Baudry - Sunye

Test fonctionnel

Objectif : obtenir un ensemble de données de test

- Potentiellement il y en a une infinité
  - a = b + c sur $\mathbb{N}$
    - Combien de combinaisons :
      - 2^32*2^32 = 2^64 = un nombre avec plus de 18 chiffres !
- Il faut choisir des données
  - Au mieux :
    - Suffisamment
    - Suffisamment répartie
    - Suffisamment efficace

Boite noire

- Le code n'est pas connu
- Basé sur la spécification
  - Règles, modèles (formels ou non)
  - Domaine d'entrée et de sortie.

Domaine d'entrée

- Plusieurs niveaux
  - type des paramètres d'une méthode
  - pré-condition sur une méthode
  - ensemble de commandes sur un système
  - grammaire d'un langage
  - ...
- On ne peut pas tout explorer, il faut délimiter
  - Génération aléatoire
  - Analyse partitionnelle
  - Test aux limites
  - Graphe causes - effets

Technique 1: Analyse partitionnelle

- A partir de la spécification
  - déterminer le domaine d'entrée du programme
- Partitionner le domaine d'entrée en classes d'équivalences
  - identifier des classes d'équivalence pour chaque donnée
  - les classes d'équivalence forment une partition du domaine de chaque donnée en entrée
  - choisir une donnée dans chacune

Méthodologie

- Si la valeur à tester appartient à un intervalle :
  - une classe pour les valeurs inférieures
  - une classe pour les valeurs supérieures
  - n classes valides
- Si la donnée est un ensemble de valeurs :
  - une classe avec l'ensemble vide
  - une classe avec trop de valeurs
  - n classes valides
- Si la donnée est une contrainte/condition:
  - une classe avec la contrainte respectée
  - une classe avec la contrainte non-respectée

Exemple du nombre de jours

- Soit à tester la méthode :

public static int nbJoursDansMois(int mois, int année)

(la spécification précise que la méthode ne couvre que le XXlième siècle.)

- Mois
  - [-2^31; 0]
  - [1; 12]
  - [13; 2^31 - 1]

Année

[-2^31; 2000]

[2001; 2100]

[2101; 2^31 - 1]

- DTs = (-2, 2010), (3, 2010), (15, 2010), (6, 1880), (6, 3000), (-2, 1880),
- (-2, 3000), (15, 1880), (15, 3000)

Amélioration fonctionnelle

- Mois
  - [-2^31; 0]
  - {1; 3; 5; 7; 8; 10; 12}
  - {4; 6; 9; 11}
  - 2
  - [13; 2^31 -1]

- Année
  - [-2^31; 2000]

  AnneesBissextiles =

  {x ∈ [2001; 2100] : (x mod 4 =0 et x mod 100 !=0) ou (x mod 400 = 0)}
  - AutresAnnees = [2001; 2100] \ AnneesBissextiles
  - [2101; 2^31 -1]

Table de décision

- Élargir l'analyse partitionnelle pour former des cas de test
  - Étude du domaine de sortie (oracle)
  - Mise en relation des partitions du domaine d'entrée et celles du domaine des résultats.

Mois / Année		$[-2^{31}, 2000]$ invalide	$[2001, 2100]$ valide	$[2101, 2^{31}-1]$ invalide
$[-2^{31}, 0]$	invalide	(-4,-10)	(-4,2010)	(-4,2222)
$[1, 12]$	valide	(9,-10)	(9, 2010)	(9,2222)
$[13, 2^{31}-1]$	invalide	(15,-10)	(15,2010)	(15,2222)

Exemple du nombre de jour

- Table de décision

Entrees	mois	$[-2^{31}, 0]$	X									
		$\{1, 3, 5, 7, 8, 10, 12\}$					X	X				
		$\{4, 6, 9, 11\}$							X	X		
		2									X	X
		$[13, 2^{31} - 1]$		X								
	annee	$[-2^{31}, 2001]$			X							
		AnneesBissextiles					X		X		X	
		AutresAnnees						X		X		X
		$[2100, 2^{31} - 1]$				X						
Sortie	31						X	X				
	30								X	X		
	29										X	
	28											X
	entrees invalides		X	X	X	X						

Caractéristiques et Limitation

- le choix des partitions est critique
- possible non prise en compte d'éventuelles différences fonctionnelles entre les éléments appartenant à la même partition
  - l'identification des problèmes/erreurs dépend de ce choix
- partitions hors limites (invalide) : tests de robustesse
- partitions dans limites : tests nominaux
- explosion combinatoire des cas de test
  - soit n données d'entrées, et 5 classes : 5n cas de tests

Technique 2 : Test aux limites

- Intuition:
  - de nombreuses erreurs se produisent dans les cas limites
- Pour chaque donnée en entrée
  - déterminer les bornes du domaine
  - prendre des valeurs sur les bornes et juste un peu autour
- Exemple
  - pour un intervalle [1, 100]
  - 1, 100, 2, 99, 0, 101

Sélection des valeurs

- si x appartient à un intervalle [a; b], prendre
  - les deux valeurs aux limites (a, b)
  - les quatre valeurs a+/-μ, b+/-μ, où μ est le plus petit écart possible
  - une/des valeur(s) dans l'intervalle
- si x appartient à un ensemble ordonne de valeurs, prendre
  - les première, deuxième, avant-dernière, et dernière valeurs
- si x définit un nombre de valeurs, prendre
  - Prendre le minimum de valeurs, le maximum, le minimum-1, le max+1

Type de donnée

- Booléen : True/false
- Logique Floue: Complètement, Pas du tout, pas complètement, un peu.

Dépendance entre paramètres et partitionnement

- Plusieurs paramètres d'entrée peuvent être partitionnés séparément
- Ces paramètres sont peut-être contraints entres-eux.

- Exemple des triangles:
  - Équilatéral sur]0;5] : (5,5,5), (1,1,1), (1,5,1), etc.

- Contrainte explicite ou implicite
  - Fonctionnel

Pairwise testing

- Problématique : explosion combinatoire
  - Méthode sous test avec plusieurs paramètres
    - 5 entiers => chaque entier 3 valeurs => 3^5 valeurs = 243
    - 5 entiers => au limite chaque entier 7 valeurs => 7^5 valeurs = 16807
    - À cela on ajoute l'état du système
  - Au-delà des plages de valeurs des données, la combinatoire complexifie le test

Pairwise testing

- Principe : tester un minimum de fois chaque pair de valeur
- Résultat : réduction du nombre de combinaison

- Combinatoire non définie, approximation :
  - O(nm) quand n et m sont les nombres de possibilités des deux valeurs en ayant le plus

Pairwise testing - exemple

Marque	Carburant	Gamme	Porte
Renault	Essence	Citadine	3
Peugeot	Diesel	Berline	5
Citroen	GPL	Monospace	

- Toutes les combinaisons : 3*3*3*2 = 54
- Toutes les paires : 9

Pairwise testing - exemple

- 9 données de test

DT	Marque	Carburant	Gamme	Porte
1	Renault	Essence	Citadine	3
2	Renault	Diesel	Berline	5
3	Renault	GPL	Monospace	3
4	Peugeot	Essence	Monospace	5
5	Peugeot	Diesel	Citadine	3
6	Peugeot	GPL	Berline	5
7	Citroen	Essence	Berline	3
8	Citroen	Diesel	Monospace	5
9	Citroen	GPL	Citadine	3

  - Toutes les paires

Pairwise testing

- Intuition : la majorité des fautes seront détectées par des combinaisons de 2 variables.
  - Ce n'est pas un postulat
  - Il restera des bugs nécessitant une combinaison exacte de toutes les variables
- Réduction importante, surtout avec beaucoup de variable
- Déclinable avec des triplets, …, jusqu'à la combinatoire
- Largement outillé :
  - http://www.pairwise.org/tools.asp

Test aléatoire

- Fonction aléatoire
- Efficace pour les fautes provoquant systématiquement des défaillances

- Adapté pour le test de robustesse
  - Combiné avec du partitionnement et de la combinatoire complète

Test statistique

- Utilisation d'une loi statistique
  - Fonction de Gauss …
  - Echantillonnage
- Plusieurs objectifs
  - Trouver là où sont le plus souvent les erreurs
  - Vérifier surtout les valeurs qui seront réellement utilisées.

Bilan techniques fonctionnelles

- Efficacité, facilité de mise en œuvre
- Attention au gain surestimé :
  - Plus on *génère* de données de test, plus on *écrira* des oracles

Le test unitaire OO

Plan

- Introduction au test unitaire
- 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?

Problème pour l'oracle

- Encapsulation : les attributs sont souvent privés
- Difficile de récupérer l'état d'un objet
- Penser à la testabilité au moment de la conception:
  - prévoir des accesseurs en lecture sur les attributs privés
  - des méthodes pour accéder à l'état de l'objet

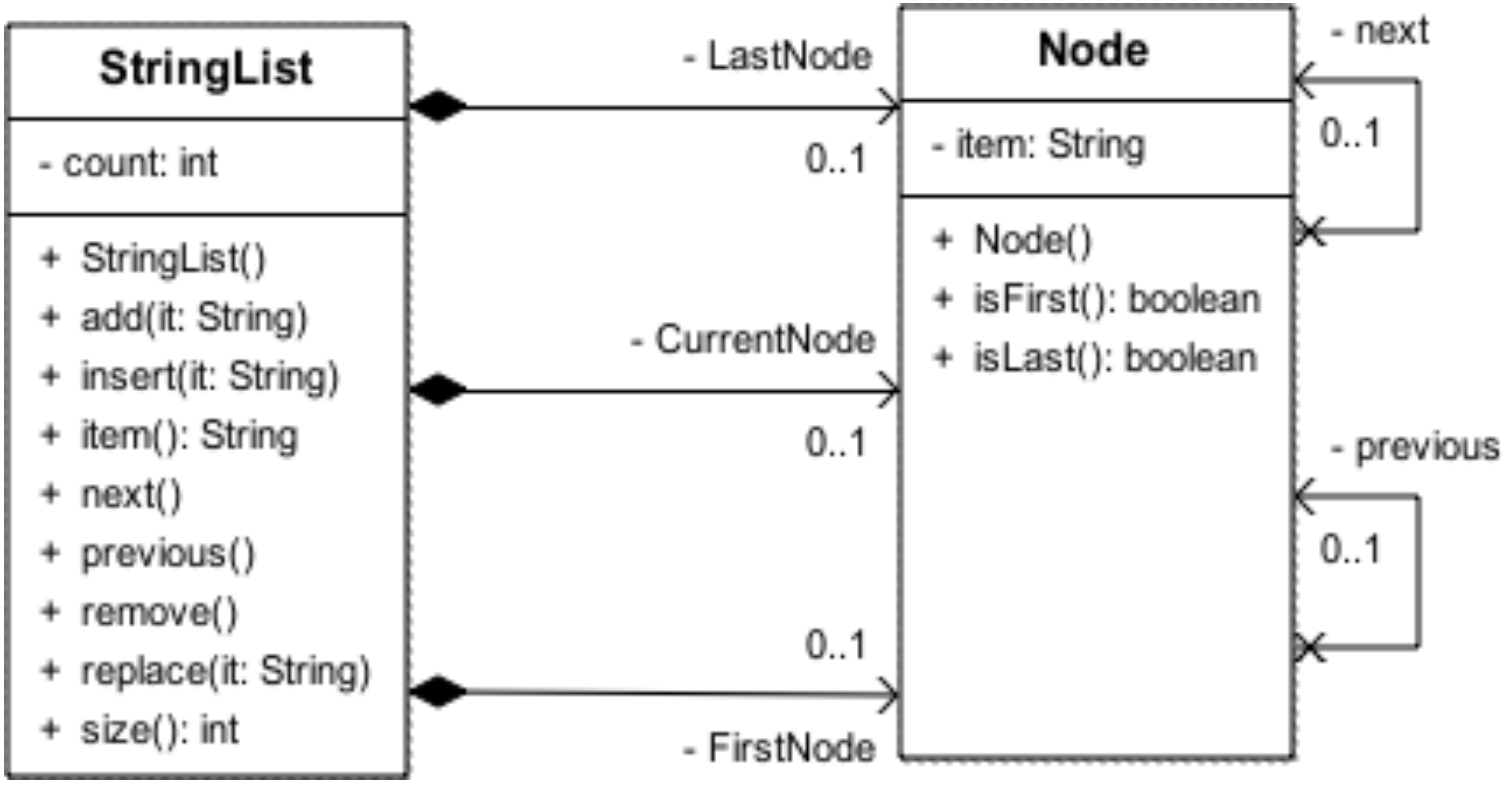
Classes de test

- Méthode = Cas de test
- Classe = Suite de tests
- Une classe de test par classe testée
  - Regroupe les cas de test
  - Il peut y avoir plusieurs classes de test pour une classe testée

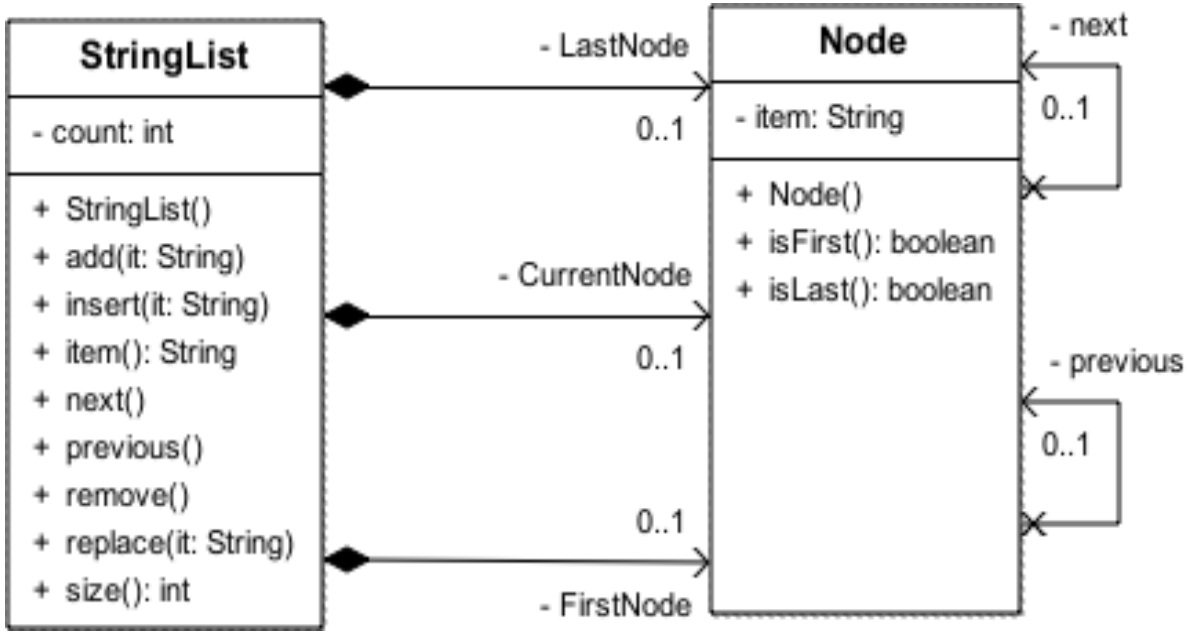
Corps de la méthode

- Configuration initiale:
  - 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

Exemple: la classe StringList

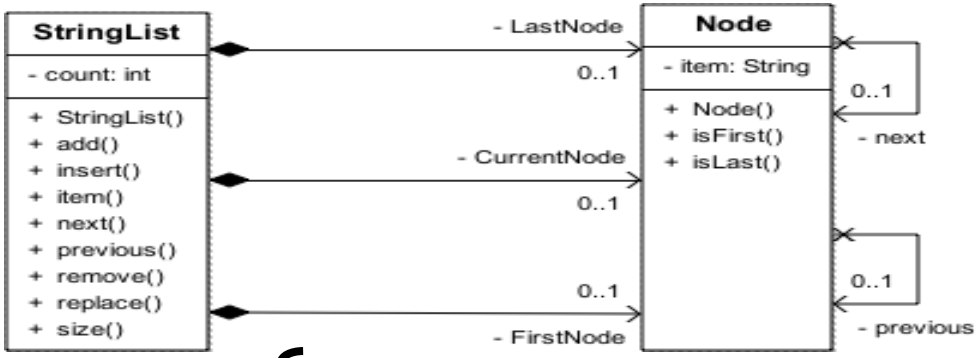


Test de StringList



- 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
@Test public void testInsert1(){
```

initialisation

```
        StringList list = new StringList();
        list.add("first");
        list.add("second");
```

appel avec donnée de test

```
        list.insert("third");
```

oracle

```
        assert list.size() == 3 ;
        assert "third".equals(list.item());
}
```

Outillage

- Jenny
  - http://www.burtleburtle.net/bob/math/jenny.html
- …

JUnit

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

(Framework)

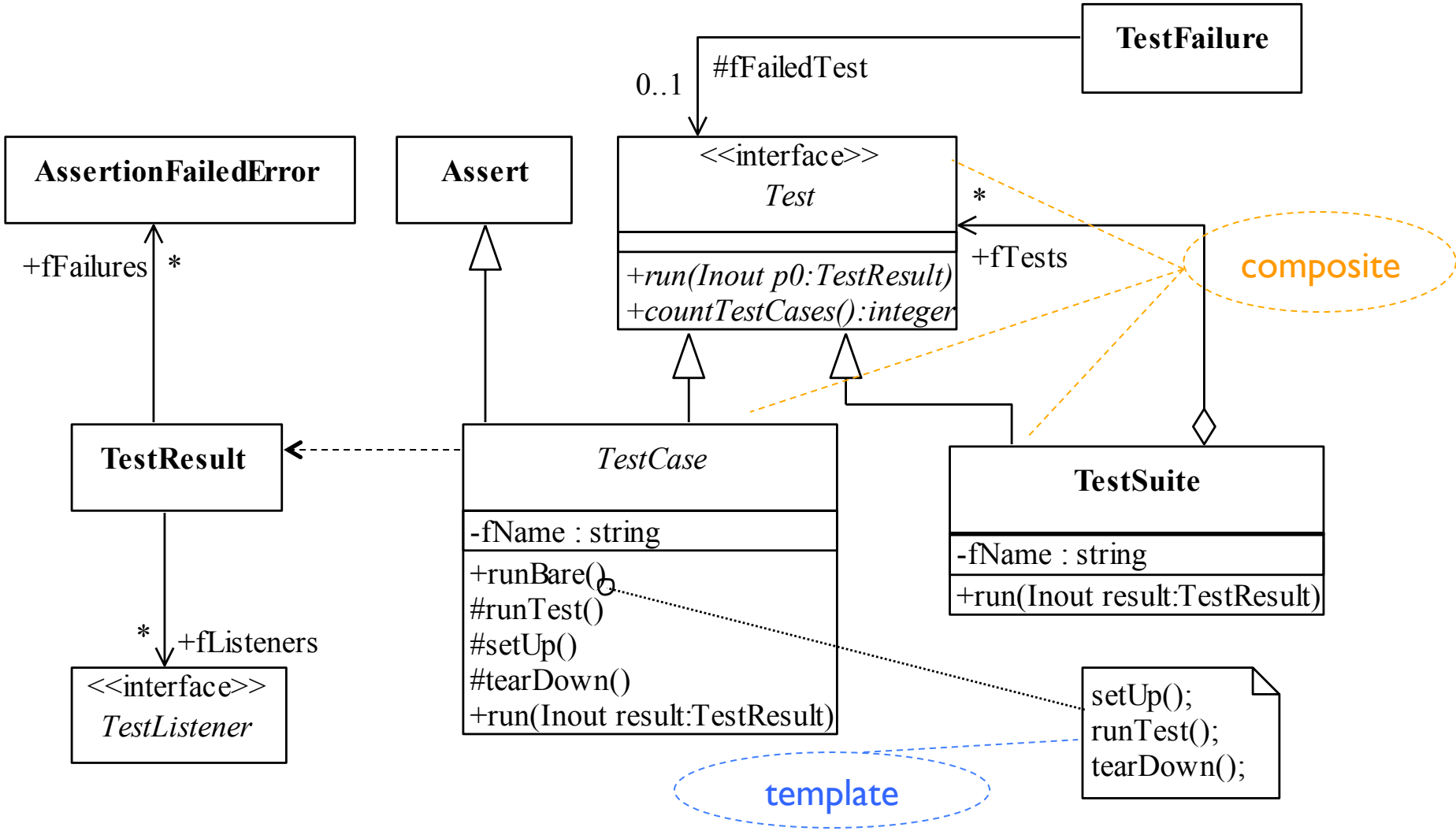
- Un framework est un ensemble de classes et de collaborations entre les instances de ces classes.

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

(Junit: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.
- S'utilise en important les librairies de Junit 4 et en utilisant des annotations
  - @Test
  - @Before, @BeforeClass
  - @After, @AfterClass

Junit : l'ancien Framework



Codage JUnit4 (1/6)

- Organisation du code des tests
  - cas de Test : TestCase
    - setUp() et tearDown() annoté avec @Before @After
    - les méthodes de test annotées avec @Test
  - suite de Test : TestSuite
    - Méthodes de test
    - Cas de test
    - Suite de Test
  - Initialisation d'une classe de test
    - @BeforeClass
    - @AfterClass

Codage (2/6)

```
//Une classe qui défini une liste de chaines de caractères

public class StringList {
        private int count;
        private Node lastNode;
        private Node firstNode;

        public StringList(){
                count=0;
        }

        public String item(){...}
        public int size(){...}
        public void add (String it){...}
        public void insert (String it){...}
...
}
```

Codage (2/6)

- Codage d'un « TestCase »:
  - déclaration de la classe:

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

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

        //@Before setUp()
        //@After tearDown()

        //@Test méthodes de test

}
```

Codage (3/6)

- la méthode setUP avec @Before:
  - //appelée avant chaque cas de test

```
@Before
public void setUp() throws Exception {
        list = new StringList();
}
```

- la méthode tearDown avec @After:
  - //appelée après chaque cas de test

```
@After
public void tearDown() throws Exception {
        super.tearDown();
}
```

Codage (4/6)

- les méthodes de test:
- caractéristiques:
  - nom préfixé par « test »
  - Annotation @Test
  - contient une assertion minimum (de préférence maximum)
    - Obligatoire sauf pour le test des levées d'eceptions

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

Test paramétrique 5/6

- constatation : beaucoup de repetition de code
- définition de patrons de code pour les classes de test, pour éviter la duplication du code
- exécution des tests avec l'exécuteur Parameterized
- les données de tests (DTs) sont issues d'une méthode annotée avec @Parameters, puis injectes dans les différentes méthodes de test

Test paramétrique 5/6 - exemple (1/2)

```
1 import org.junit.* ;
2 import static org.junit.Assert.*  ;
3 import java.util.*  ;
4 import org.junit.runner.RunWith ;
5 import org.junit.runners.Parameterized ;
6 import org.junit.runners.Parameterized.Parameters ;
7
8 @RunWith ( Parameterized.class) public class TestCalcul3 {
9
10 private Calcul maCalculatrice ;
11 private int arg1 ;
12 private int arg2 ;
13 private int res ;
14
15 public TestCalcul3 ( int arg1 , int arg2 , int res ) {
16      this.arg1 = arg1 ;
17      this.arg2 = arg2 ;
18      this.res = res ;
19 }
```

Test paramétrique 5/6 - exemple (2/2)

```
1 @Before public void nouvelleCalculatrice( ) {
2       maCalculatrice = new Calcul ( ) ;
3 }
4
5 @Parameters public static Collection data ( ) {
6       Object[][] data = {
7               {0 ,0 ,0} ,
8               {0 ,1 ,0} ,
9               {1 ,1 ,1} ,
10              {1 ,2 ,2} ,
11              {2 ,2 ,4} ,
12              {2 ,3 ,6} ,
13              {3 ,1 ,3} ,
14              {3 ,3 ,9} ,
15      } ;
16      return Arrays.asList(data);
17 }
18
19@Test public void testMultiplication ( ) {
20      assertEquals( "Test "+arg1+"  " +arg2 ,      res ,
21              maCalculatrice.mul(arg1 , arg2));
22 }
```

Suite de test

- Le lancement classe de test par classe de test n'est pas adapté aux tests de non-régression, ni aux projets de taille importante.

```
1  import org.junit.runner.RunWith ;
2  import org.junit.runners.Suite ;
3  import org.junit.runners.Suite.SuiteClasses ;
4
5  @RunWith (Suite.class)
6  @SuiteClasses( { TestCalcul.class ,
7                       TestCalcul2.class, TestCalcul3.class})
9
10 public classAllTests {
11
12 public static void main ( String args [] ) {
13      org.junit.runner.JUnitCore.main ( " AllTests " ) ;
14 }
15
16 }
```

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);
         }
```

Codage (6/6)

- JUnit utilise les mécanismes de réflexion de Java (java.lang.reflect.*) pour exécuter les méthodes de test.

Les assertions

- assertEquals (String msg, Object expected, Object actual)
- assertSame(String msg, Object exp, Object act)
- assertEquals (String msg, Object exp [], Object act [])
- assertEquals (String msg, oat exp, oat act , oat delta)
- assertTrue (String msg, boolean b)
- assertNotNull (String msg, Object o)
- fail (String msg)

- + des variantes (sans msg), etc.

Les exceptions

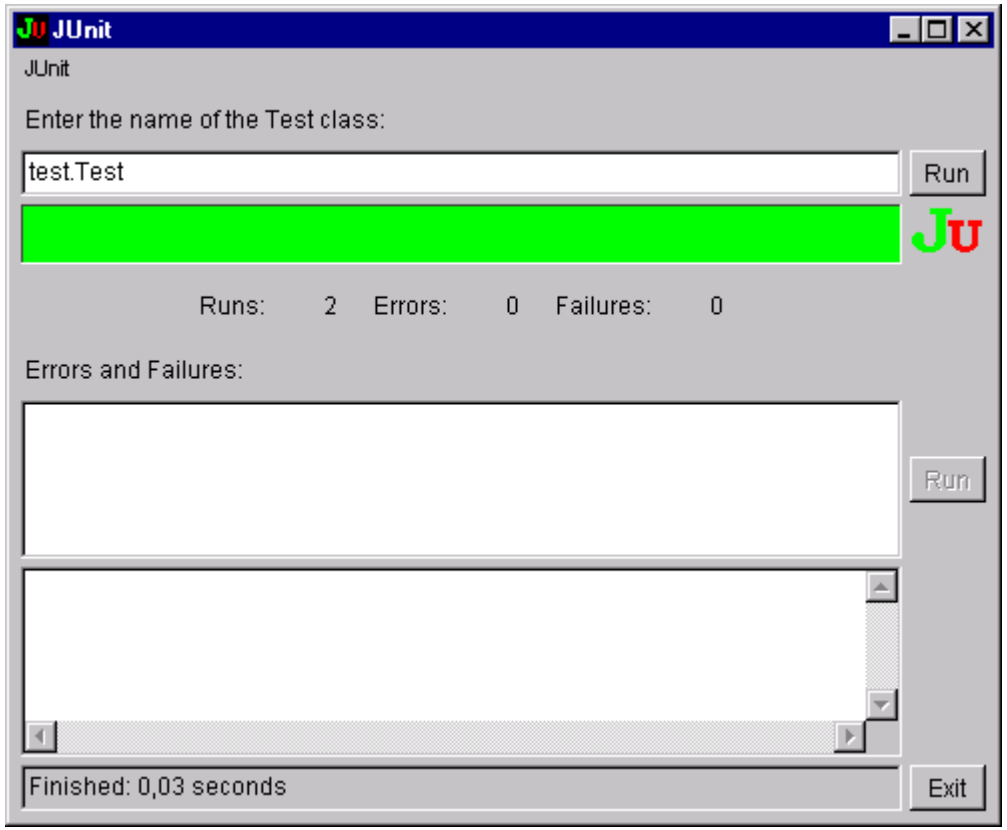
- Mecanisme permettant d'ecrire des tests vérifiant la levée des exceptsions
- Pas de bloc try .. catch , mais
- Utilisation de l'annotation parametree

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

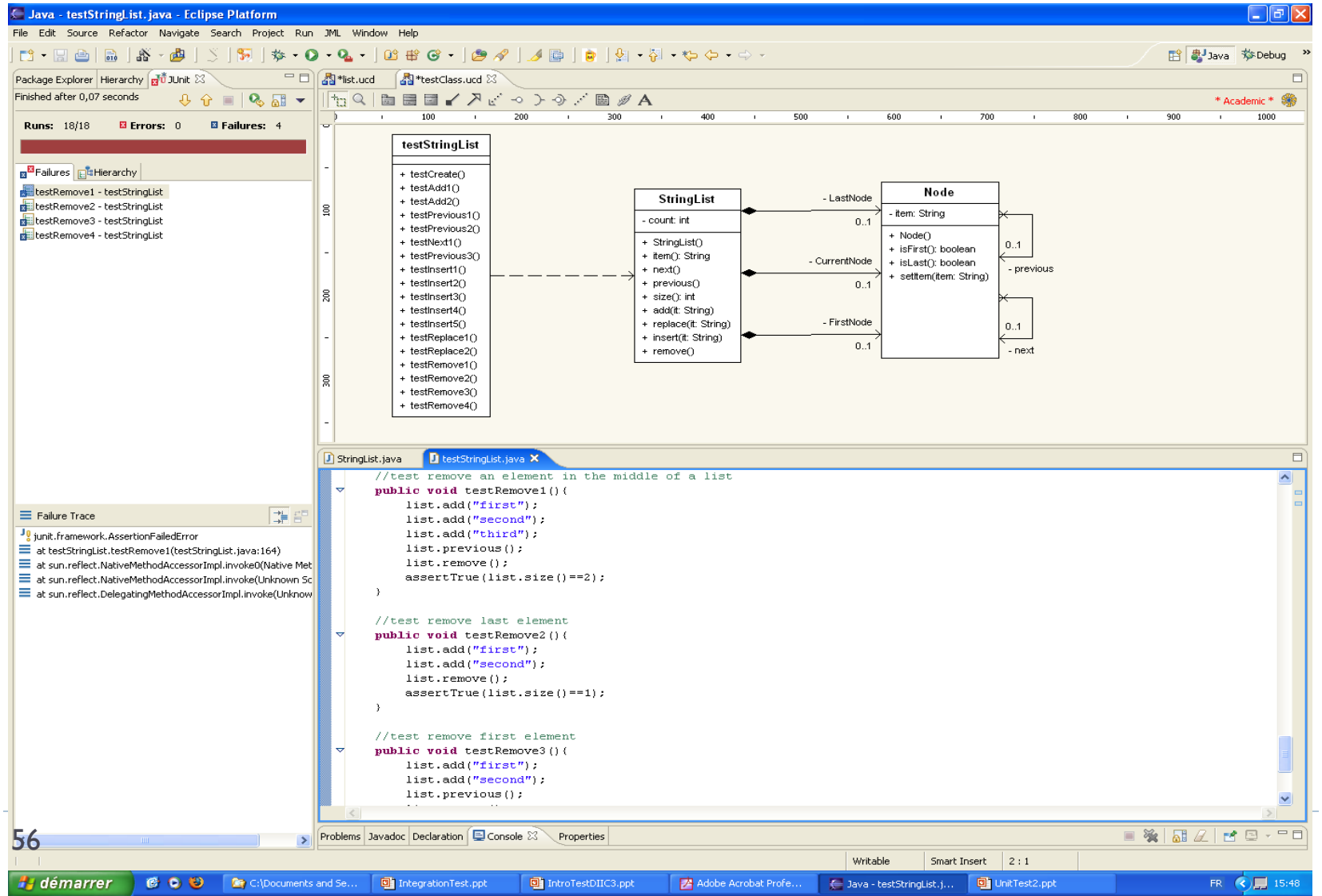
Les exceptions

```
1    @Test(expected=FactureException.class)
2    public void testModificationFactureValide()
3      throws FactureException{
4          maFacture.valide();
5          maFacture.add(newArticle("article3",1,20) ) ;
6    }
```

TestRunner


JUnit
JUnit
Enter the name of the Test class:
test.Test
Run
Runs: 2 Errors: 0 Failures: 0
Errors and Failures:
Run
Finished: 0,03 seconds
Exit

Intégration dans Eclipse



Intégration dans Maven

- Plug-in Surefire
- La commande «mvn test» exécute toutes les classes de test appelées "*Test*.java" qui se trouvent dans le répertoire "./src/test/java".

Points forts (1/2)

- Simple à comprendre: méthode, classe et suites de test.
- Simple à utiliser: @Test, @Before, @After, etc.
- Gratuit, intégré à Eclipse, à Maven, etc.
- Structuré: cas de test, suite de tests.

Points forts (2/2)

- Permet de sauvegarder les cas de test:
  - important pour la non régression.
  - quand une classe évolue, on ré-exécute les cas de test.
- Plusieurs extensions: tests de BD et IHM, rapports, etc.
- Généralisation des concepts (XUnit):
  - http://www.testdriven.com

Points faibles (1/2)

- Exploitation des résultats (pas d'historique...).
- Génération de données de tests.
- Impossibilité de configuration de l'exécution (se fait par édition et compilation des cas de test).
- Impossibilité de réaliser des tests en parallèle.

Points faibles (2/2)

- Adapté à certaines méthodes: publiques, types de paramètres simples, résultat simple à exploiter :
  - public Integer add(Integer);
- Moins adapté à d'autres:
  - public void doSomethingVeryComplicate();
  - private Boolean checkDate(Date);
  - public void retrieveData(DBConnection);
- Absence de notion de dépendance entre méthodes de test.