Analyse et Conception Logiciel
Tests Logiciel

Pascal Urso

Sources : Ian Sommerville

Lionel Seinturier

License Creative Commons

- Cette création est mise à disposition selon le Contrat *Paternité - Pas d'Utilisation Commerciale - Partage des Conditions Initiales à l'Identique 2.0 France* disponible en ligne http://creativecommons.org/licenses/by-nc-sa/2.0/fr/ ou par courrier postal à Creative Commons, 171 Second Street, Suite 300, San Francisco, California 94105, USA.

Introduction

Contexte : Ingénierie logicielle

- Discipline professionnelle couvrant tous les aspects de la production de logiciels
  - Méthodes,
  - Outils,
  - Activités
- Basé sur de la théorie et beaucoup de pratique
- Maitrise des coûts
- "*Software engineering is concerned with cost-effective software development*" *[Ian Sommerville]*

Activités de production logicielle

- **Spécification** : où (les clients et) les professionnels définissent le logiciel devant être produit et les contraintes sur ses opérations (besoins fonctionnels et non-fonctionnels)
- **Développement** : où le logiciel est conçu et programmer
- **Validation** : où l'on vérifie avec les utilisateurs que le logiciel obtenu est celui désiré
- **Evolution** : où le logiciel est modifié en fonction des changements dans les besoins des utilisateurs et dans le marché

Objectifs du test logiciel

- Activitée de développement
  - Objectif général : vérification
- Démontrer qu'un morceau de logiciel fait ce pour quoi il est prévu
  - Spécification technique fonctionnelle et non-fonctionnelle
  - Documentation technique du logiciel
- Assurer la confiance dans le logiciel
  - Travail incrémental
  - Travail en équipe
- Détecter des erreurs dans un logiciel
  - *"Testing is the process of executing a program with the intent of finding errors " [Myers :1979] The Art of Software Testing*

Exemple de test

```
@Test
public void testInsertMid() {
    Document d = new Document("Bonjour amis");
    d.insert(8,"les ");
    assertEquals(d.toString(), "Bonjour les amis");
}

@Test
public void testInsertEnd() {
    Document d = new Document("Bonjour les");
    d.insert(11," amis");
    assertEquals(d.toString(), "Bonjour les amis");
}

@Test(expected=IndexOutOfBoundsException.class)
public void testInsertOutEnd() {
    Document d = new Document("Bonjour les");
    d.insert(12," amis");
    fail("Insert out of bound not detected");
}
```

Autres techniques de vérification

- Autres techniques pour vérifier un programme
  - preuve : modélisation formelle + établissement de théorèmes
  - vérification automatique : ex model-checking
- Tests :
  - + facile à appréhender,
  - + proche du code,
  - - coûteux à mettre en œuvre
- Mais
  - ne garantît pas que le programme est exempt d'erreur
  - on ne teste que ce à quoi on a pensé
  - *"Program testing can be used to show the presence of bugs, but never to show their absence!" [Edsger Dijkstra]*

Vérification != Validation

- Quelle que soit la méthode, on s'en remet à une spécification
  - Qui vérifie la spécification ?
- Validation : inspection logicielles
  - Activités statiques dépendant des
    - Besoins
    - Utilisateurs
    - Marché
  - Cf. Analyse des besoins

Importance des tests

- Les tests sont importants (tout le monde d'accord)
  - mais les tests tendent à être repoussés à la fin d'un projet
  - plus une erreur est découverte tard, plus elle coûte cher
- *Best practice* : Tests systématiques
  - développement conduit par les tests
- L'écriture de test est une tâche dévolue
  - ni aux utilisateurs, ni aux managers, ni aux chefs de projet
  - c'est vraiment une activité de programmeur
  - mais pas forcement de celui qui à écrit le code à tester

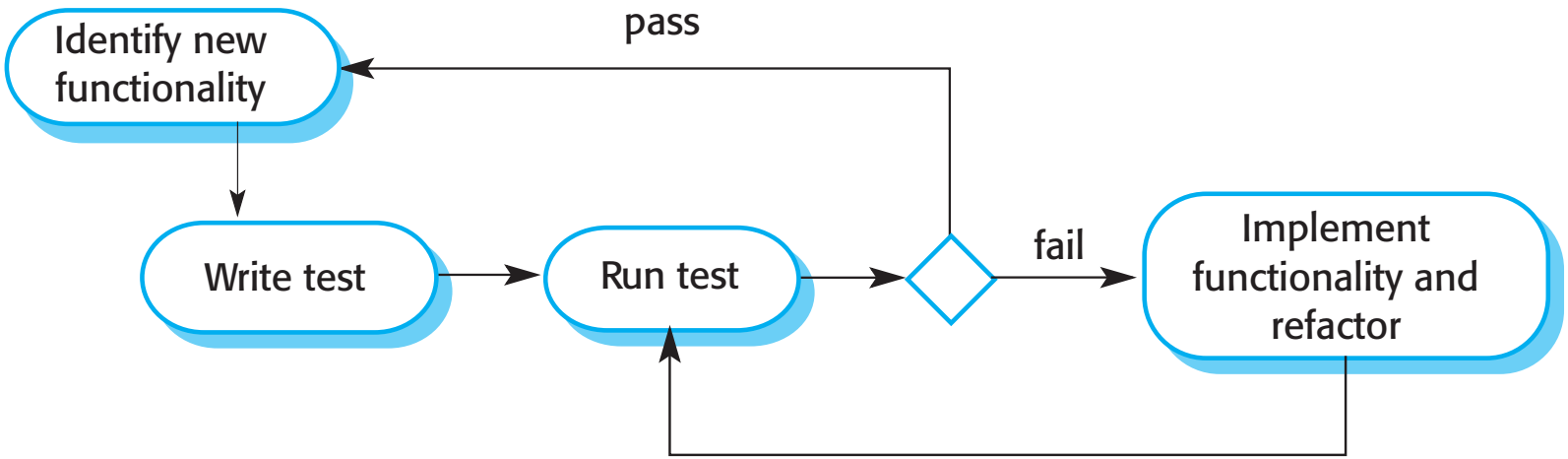
Bénéfice du test systématique

- Couverture du code
  - Chaque méthode est testée
- Suite de test de régression
  - Une batterie de test développé incrémentalement
- Débug simplifié
  - Une nouvelle erreur a beaucoup de chance de provenir d'un nouveau développement
- Documentation
  - Bien moins ambiguë que du langage naturel

Développement conduit par les tests

- "Test-driven development" (TDD)
- Ecrire le test avant le code
- Le but du développeur devient "faire passer les tests" !
- Développement incrémental
- Issu des méthodes agiles (eXtrem Programing) mais adapté aux méthodes sur plan
- Bénéfice supplémentaire
  - Pas de "mauvaise tentation"

Test-driven development



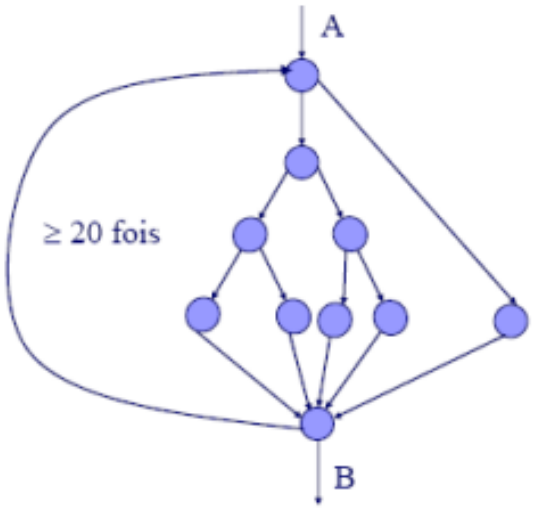
Catégorisation

Deux "familles" de tests

- Tests boîte blanche
- Tests boîte noire

Tests boîte blanche

- On a accès à la structure du code
  - ex. : calcul de tous les chemins d'exécution possibles (branches, instructions, …)
  - tests sur chacun de ces chemins
- avantage :
  - on peut détecter des situations extrêmes qui ne se manifestent que dans certaines branches
- inconvénients
  - calcul des chemins souvent long
  - il faut avoir accès au code source



Tests boîte noire

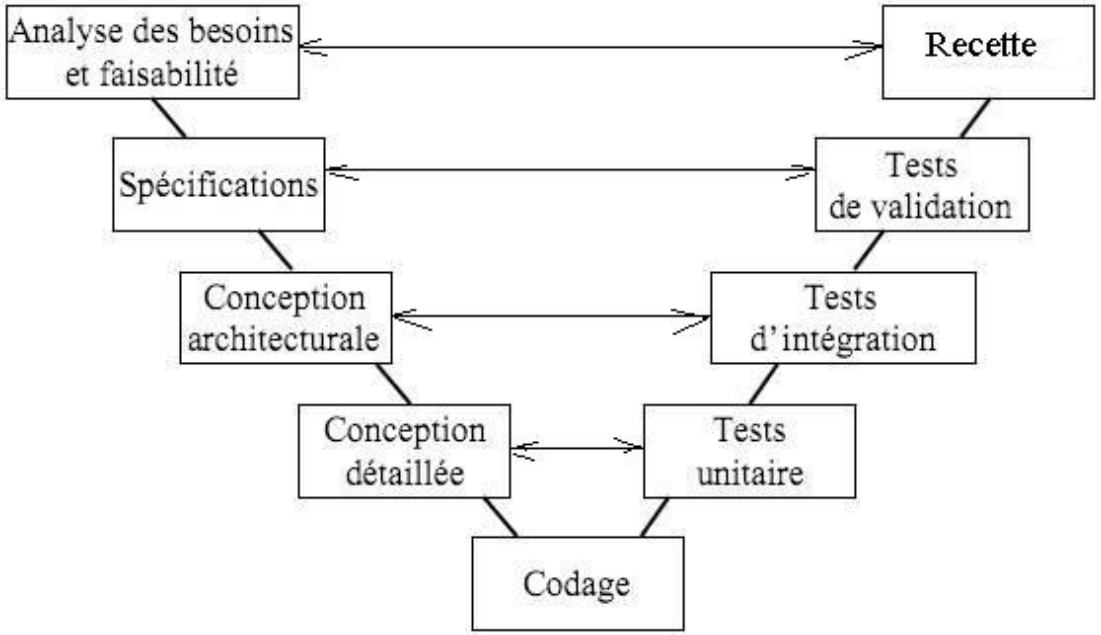
- Les + courants
- Aucune connaissance des détails d'implémentation n'est nécessaire
- On s'en remet à la spécification
- On vérifie que l'implémentation fournit bien le résultat attendu par la spéc
- *EX :ASSERT(cinq.plus(trois).equals(deux))*

D'habitude…

- Dans les deux cas on a besoin d'une spécification
- Dans le cas B.B. la spécification est souvent compliquée
  - source d'erreurs
  - coûts
- *En général*, on a accès au code mais on fait du B.N.
  - On se sert du code pour "valider" les tests, *ex : taux de couverture*

Quatre "granularités" de tests

- Tests unitaires
- Tests d'intégration
- Tests systèmes (aka tests de validation)
- Recette



Tests unitaires

- Les plus fréquents
- Unité = composant logiciel
  - méthode,
  - mais aussi classe, package, web service, servlet, client, serveur, programme, …
- Principe = tester le composant en isolation
  - le résultat du test doit montrer le disfonctionnement du seul composant
  - ne doit pas utiliser de composant non-fiable
    - java.util.ArrayList OK
    - fr.uhp.MaListeGéniale KO
- Doit être automatisé
  - xUnit

Tests d'intégration

- Test d'un assemblage d'instance de différents composants
  - déjà testé unitairement
  - en isolation du système
- Exemple :
  - Listener + Component Swing
  - Modèle + HTTPServlet
- Doit être automatisé aussi
  - xUnit aussi
  - Mais peut être couteux
- Intégration continue

Tests système et recette

- Tests système
  - Test de l'intégration d'une application complète avec son env. d'exécution (OS, VM, librairies, serveurs, ...)
    - Environnement de développement
    - != Serveur de test
    - != Serveur de production
  - Plutôt des test de performance, stress test
  - Monté par une autre équipe que l'équipe de développement
- Recette
  - Effectuée en présence du client afin de valider l'application complète

Directives

Ecriture d'un test

- Tout est du code
  - Comportement : code
  - Test : code
- Code du test
  - Plus petit que le code à tester
  - Ne contient pas de complexité algorithmique
    - source d'erreur
  - Fournit un retour explicite
    - ~~assertTrue(4 == x)~~ FAILED : AssertionFailedError
    - assertEquals(4, x) FAILED : expected:<4> but was:<0>

Principe d'un test

- Action avec effet de bord
  - Un objet dans un état
  - On fait l'action
  - On vérifie l'état après l'action
- Calcul sans effet de bord
  - Des données paramétrées
  - On fait le calcul
  - On vérifie le résultat
- Action et calcul = méthode, requête HTTP, programme, …

Logiciel testable

- Prévoir l'accès à l'état
  - ajouter des getter pour voir le résultat
  - ajouter des setter pour paramétrer les données ou l'état
- Séparer la fonctionnalité de l'effet
- Injection de dépendance (coming soon)

Quelles unités tester ?

- Grave question !
  - impact sur les coûts
  - expérience du développeur
- A la base toute méthode
  - modulo les "too small to fail"
    - getter/setter
    - méthodes de 2 lignes
    - bean "pure"
- Reproduction d'un bug ayant déjà eu lieu
  - 1 bug => 1 test

Quelles valeurs tester ?

- Tester les "grandes catégories" (classes d'équivalence)
  - cas exception
  - cas retour null
  - cas standard
- Tester les conditions aux limites
  - paramètre null, tableau vide, valeurs max/min
- Tester les invariants du programme
  - cohérence des données
  - intégrité référentielle

Mesure de la qualité des tests

- Notion de couverture de code (coverage)
  - % de code couvert par des tests (unitaires, intégration, ...)
  - % de méthodes, branches, lignes, instructions
  - ne garantît pas la correction
  - indicateur sur le "sérieux" d'un ensemble de tests
- Outils
  - Cobertura, EMMA, Patchwork, Quilt,
- Peut entrer en conflit avec la notion de "*too small to fail*"

Principe Right-BICEP

- R (Right)
  - est-ce que les résultats sont corrects ?
- B (Boundary)
  - est-ce que les conditions aux limites sont correctes ?
- I (Inverse)
  - est-ce que l'on peut vérifier la relation inverse ?
- C (Cross-check)
  - est-ce que l'on peut vérifier le résultat autrement ?
- E (Error condition)
  - est-ce que l'on peut forcer l'occurrence d'erreurs ?
- P (Performance)
  - est-ce que les performances sont prévisibles ?

Right

- Validation des résultats en fonction de ce que définit la spécification
- 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
- la notion de correction peut évoluer au cours du temps
  - en fonction de l'évolution des besoins, des spécifications, ...

Boundary conditions (BICEP)

- Identifier les conditions aux limites de la spécification
- Que se passe-t-il lorsque les données sont
  - anachroniques — ex. : !*W@\/"
  - mal formattées — ex.:fred@foobar
  - vides ou nulles — ex. : 0, 0.0, "", null
  - extraordinaires — ex. : 10000 pour un âge
  - dupliquées — ex. : doublon dans un Set
  - non conformes — ex. : listes ordonnées qui ne le sont pas
  - désordonnées — ex. : imprimer avant de se connecter
- principe "CORRECT"
  - conformance, ordering, range, reference, existence, cardinality, time

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

- Conformance
  - données doivent souvent respecter un format bien particulier
    - format de fichier, email (voir RFC 822), URL, …
- Ordering
  - l'ordre des données dans une structure peut avoir de l'importance
- Range
  - situation où une donnée a moins de valeurs légales que son type (ex. int angle ∈ 0..359)
- Reference
  - les dépendances de la méthode à tester avec le reste de l'application

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

- Existence
  - que se passe-t-il lorsque la valeur attendue n'existe pas ? ex : pas de Client associé à une Facture
- Cardinality
  - que se passe-t-il avec les valeurs "remarquables" ? ex : 0, 1, -1, Integer.MAX_VALUE, Integer.MIN_VALUE,
- Time
  - les cas où l'ordre dans le temps à de l'importance ex. : pas de logout() avant login()
  - cf. les diagrammes états/transitions => tester ce qui n'est pas dans le diagramme

Inverse - Cross-check (BICEP)

- **Inverse** : identifier les relations inverses

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

- **Cross-check** : identifier les algorithmes équivalents qui permettent de vérifier le comportement

```
public void testSquareRootUsingStd() {
   double x1 = mySquareRoot(4.0);
   double x2 = Math.sqrt(4.0);
   assertEquals( x2, x1, 0.0001 );
}
```

Error condition – Performance (BICEP)

- **Error** : Que se passe-t-il en cas de
  - disque, mémoire, ... plein
  - perte de connexion réseau
- **Performance :** Est-ce que le système réagit de façon non exponentielle à une montée en charge ?
  - ex. : vérifier qu'un élément n'est pas dans une liste
    - vérifier que le temps est linéaire avec la taille de la liste
  - application continue de répondre en présence d'une surcharge du système ?
- Test particuliers (frameworks d'injection de charge)

JUnit

JUnit

- Framework de test pour Java
  - http://www.junit.org
- Adapté
  - à tout test automatique
  - aux tests unitaires
  - aux tests d'intégration
- JUnit 3.x
  - héritage
- JUnit 4.x
  - annotation

Exemple de code à tester

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

- Que manque-t-il ?

Exemple de code testable

```
public class MyArray {

  ...

  public int[] getTab() {
      return tab;
  }
  public void setTab(int[] tab) {
      this.tab = tab;
  }
}
```

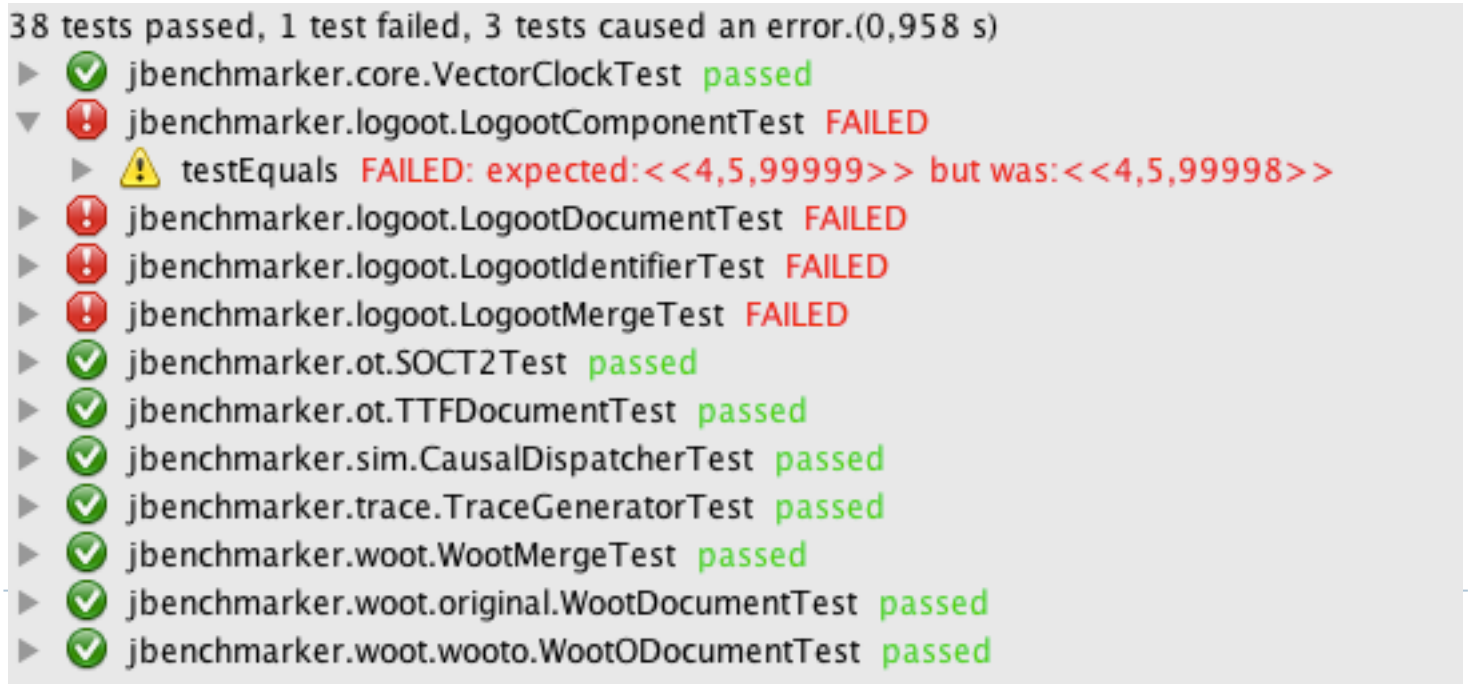
Classe de test JUnit 3.x

- Hérite de junit.framework.TestCase
  - méthodes assertXXX()
  - tests = méthodes publiques avec un nom commençant par test

```
public class MyArrayTest extends TestCase {
  public void testRight() {
      MyArray a = new MyArray();
      a.setTab(new int[]{17,0,9});
      assertEquals(17, a.largest());
  }
}
```

Lancement du test

- ligne de commande :
  - java junit.textui.TestRunner TestLargest
- GUI swing
  - java junit.swingui.TestRunner
- Environnement de développement (eclipse, NetBean, ...)



Méthodes AssertXXX de TestCase

- Voir API
  - assertSame(["message",] expected, actual); // ==
  - assertNotSame(["message",] expected, actual); // !=
  - assertEquals(["message",] expected, actual); // equals
  - assertNull(["message",] actual);
  - assertNotNull(["message",] actual);
  - assertTrue, assertFalse, assertArrayEquals
- Fail (pour les autres cas)
  - if (condition) fail(["message"]);

Pour chaque méthode de test

- Méthodes protected
  - setUp exécutée avant
  - tearDown exécutée après

```
public class TestDB extends TestCase {
  private Connection cx;
  protected void setUp() {
       cx = new Connection( "mysql", "localhost", "9001",
  "bob, "");
       cx.connect();
  }
  protected void tearDown() {
       cx.disconnect();
  }
  public void test1() { ... /* cx = une connection */ ... }
  public void test2() { ... /* cx = une autre connection
  */ ... }
}
```

Initialisation globale pour toutes les méthodes de test

- Méthodes protected
  - static void oneTimeSetUp()
  - static void oneTimeTearDown()
- Ordonnancement
  - oneTimeSetUp()
  - setUp()
  - test1()
  - tearDown()
  - setUp()
  - test2()
  - tearDown()
  - oneTimeTearDown()

JUnit 4.x

- Package org.junit
- Annotation @Test
  - méthode de test
  - paramètre expected et timeout (en ms)
  - @Test(expected=SomeThrowable, timeout=100)
- Annotation @Ignore
  - méthode de test à ignorer
  - @Ignore("test not ready")

Example JUnit 4.x

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

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

JUnit 4.x

- Initialisation
  - setUp() : une méthode annotée @Before
  - tearDown() : une méthode annotée @After
  - oneTimeSetUp() : une méthode annotée @BeforeClass
  - oneTimeTearDown() : une méthode annotée @AfterClass
- Lancement des tests
  - java org.junit.runner.JUnitCore ClassTest
  - environnement de développement !

FAIL : bug ou spécification ?

Fail

```
@Test public void negativeValues() {
    MyArray a = new MyArray();
    a.setTab(new int[]{-7,-3,-99});
    assertEquals(-3, a.largest());
}
```

- **Résultat**

```
There was 1 failure:
1) negativeValues(MyArrayTest)
junit.framework.AssertionFailedError:
expected:<-3> but was:<0>
```

BUG ?

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

Ou spécification ?

```
/**
 * Returns the largest positive integer, 0 if none
 **/
public int largest(int[] list)
```

- Test

```
@Test public void negativeValues() {
      MyArray a = new MyArray();
      a.setTab(new int[]{-7,-3,-99});
      assertEquals(0, a.largest());
  }
```