IFT3912 – Développement et maintenance de logiciels

Tests

Bruno Dufour
Université de Montréal
dufour@iro.umontreal.ca

L'importance des tests

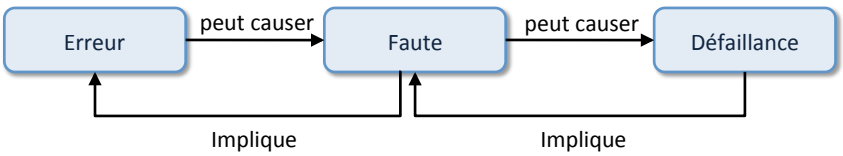
- L'erreur est humaine (« *errare humanum est* »), presque tous les programmes contiennent des erreurs
- Les erreurs sont difficiles à identifier:
  - Grande complexité des logiciels
  - Invisibilité du système développé
  - Pas de principe de continuité
- L'activité de test est essentielle au développement de logiciels de qualité

Définitions

- **Erreur (*error*)**: commise par un développeur
  - Erreur de programmation
  - Erreur de logique
- **Faute** ou **défault (*fault, defect*)**: état interne invalide d'un logiciel après l'activation d'une erreur
- **Défaillance (*failure*)**: manifestation externe d'une faute
  - Observable (ex: le programme dévie de sa spécifications)

3

Erreurs, Fautes & Défaillances



4

Développement de logiciels fiables

- Prévention des fautes
  - Méthodes formelles (IFT3911 – J. Vachon)
  - Réutilisation de code
  - Méthode de développement rigoureuses
- Prévision des fautes
  - Métriques de qualité (IFT3913 – H. Sarahoui)
  - Méthodes statistiques
- Identification et correction des fautes
  - Vérification
  - Validation
- Tolérance aux fautes

5

Tests

- **Test**: exécution d'un programme dans le but de découvrir des erreurs [Myers 1979], non pas « … dans le but de démontrer l'absence d'erreurs »
  - Démontrer l'absence est très difficile ou impossible même pour de petits programmes
  - « Si on tente de démontrer l'absence d'erreurs, on n'en découvre que très peu. Si on tente de démontrer la présence d'erreurs, on en découvre la grande majorité. » [Myers, 1979]
  - Similairement, « … dans le but de démontrer que le programme fait ce qui est demandé » n'est pas suffisant : un programme peut contenir une erreur s'il fait autre chose en plus de ce qui est demandé, par exemple.

6

L'importance des tests

- On peut généralement faire l'hypothèse qu'un programme contient des erreurs
- Les tests peuvent être utilisés durant toutes les étapes du développement
  - **Avant** de débuter l'implémentation (TDD)
  - **Durant** le développement
  - **Après** que le logiciel soit complété
- Les tests représentent une partie importante du développement
  - Jusqu'à 33% du budget
  - Jusqu'à 27% du temps de développement

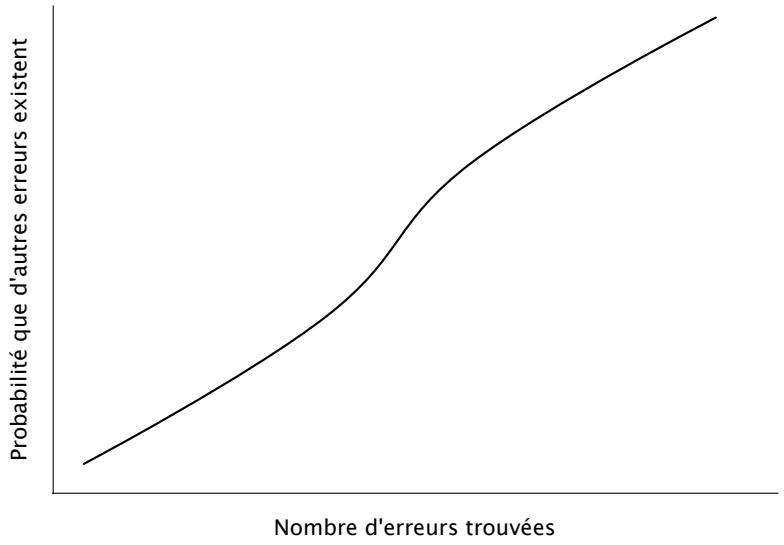
7

L'art du test

- Bien tester est difficile
  - Tester est un processus destructif
  - Tester une activité très intellectuelle et requiert beaucoup de créativité
- Faire **échouer** le programme est une **réussite** lorsqu'on développe des tests
  - Éventuellement, les tests permettent d'atteindre un niveau de qualité acceptable

8

Une erreur n'est (presque) jamais seule



9

Principes de base

- Ne jamais planifier un effort de test sous l'hypothèse tacite qu'aucune erreur ne sera trouvée.
- Un élément nécessaire d'un test est une définition de la sortie ou du résultat
- Un programmeur doit éviter de tester son propre programme
- Les tests doivent être écrits pour les conditions d'entrée qui sont invalides et inattendues, ainsi que pour celles qui sont valides et attendues.
- L'examen d'un programme pour déterminer s'il ne fait pas ce qu'il est censé faire n'est que la moitié de la bataille, l'autre moitié, consiste à déterminer si le programme fait ce qu'il n'est pas censé faire.
- Éviter les tests jetables à moins que le programme soit lui-même vraiment un programme jetable.

10

L'activité de test

- 2 catégories d'activités :
  - Tests humains : basés sur la lecture du code par un être humain
    - Ex: revues, inspections
  - Tests informatisés : basés sur l'exécution d'un programme par une machine
    - Différents niveaux de granularité
    - Différents niveaux d'automatisation

11

Inspections et revues

12

Inspections & revues

- Techniques de test par des humains
  - Une équipe lit ou inspecte le code visuellement
- L'objectif d'une inspection ou revue est l'identification d'erreurs de programmation ou de logique
  - Pas de solutions identifiées en général (test et non débogage)
- Avantages :
  - Des développeurs autres que l'auteur du code sont impliqués
  - Peut être appliqué avant que le code ne soit fonctionnel
  - Permet de réduire le coût de débogage puisque l'emplacement des erreurs est précisément identifié
- Efficacité :
  - Permet de trouver de 30% à 70% des erreurs
  - Est plus efficace que des tests automatisés à détecter certains types d'erreurs

13

Inspections

- Une inspection de code est un ensemble de procédures et de techniques de détection d'erreurs pour la lecture de code en groupe.
- Une inspection est généralement effectuée en groupe de 4 personnes, et dure entre entre 1 et 2 heures
  - Un modérateur
  - Un programmeur (auteur du code inspecté)
  - Deux autres développeurs

14

Inspections - Déroulement

- Les participants reçoivent le matériel à l'avance (code, spécification) et se familiarisent avec celui-ci
- Durant l'inspection :
  - L'auteur du code fait la narration de la logique du code, ligne par ligne.
  - Les autres participants posent des questions, et l'équipe tente d'y répondre pour découvrir les erreurs.
  - Le programme est analysé à l'aide d'une liste d'erreurs fréquentes
    - Contient des erreurs qui sont indépendantes du langage utilisé ainsi que des erreurs spécifiques au langage
- À la fin de la session, le programmeur reçoit une liste d'erreurs identifiés. Si plusieurs erreurs sont trouvées, une autre inspection peut être demandée.
  - Cette liste peut servir à améliorer le processus pour les inspections futures

15

Exemple – List d'erreurs fréquentes

- Erreurs de référence de données
  - Variable non-initialisée
  - Index invalide pour les accès aux tableaux
    - Ne respecte pas les limites, valeur autre qu'un entier
  - Mémoire utilisée n'a pas été préalablement allouée
- Erreurs de déclaration de données
  - Variables non-déclarées
  - Les champs d'une classe n'ont pas les bonnes propriétés (type, visibilité)
  - Variable ont des noms trop similaires (ex: `volt`, `volts`)
- Erreurs de calcul
  - Calculs impliquant des types mixtes ou incohérents
  - Débordement des valeurs (*overflow*)
  - Division possible par zéro
  - Problème de précédence des opérateurs

16

Exemple – List d'erreurs fréquentes

- Erreurs de comparaisons
  - Comparaisons entre types mixtes et/ou incohérents
  - Mauvaise comparaison employée (> vs <, >= vs >, == vs equals(), etc.)
  - Dépendance sur le mécanisme évaluation :

```
// x=1, y=0, z=4
if ((y==0 || (x/y)>z) {…}
```

- Erreurs de flot de contrôle
  - Boucle infinie
  - Boucle non-exécutée à cause des conditions initiales
  - Nombre d'itérations qui dépasse ou est inférieur à la limite par 1
  - Décisions non-exhaustives (ex: cas manquants pour `switch`)

17

Exemple – List d'erreurs fréquentes

- Erreurs d'interface
  - Nombre de paramètres incorrect
  - Incohérence des unités entre arguments et paramètres déclarés
- Erreurs d'entrée / sortie
  - Fichiers non-ouverts avant l'écriture
  - Fichiers non-fermés après l'écriture
  - Exceptions dues aux opérations d'entrée / sortie ne sont pas traitées correctement
  - Fautes d'orthographe / de grammaire dans le texte généré ou affiché
- Autres erreurs
  - Variables non-utilisées
  - Validité des arguments non-testée
  - Fonction absente du programme
  - Présence d'alertes lors de la compilation

18

Inspection - Avantages

- Permet d'identifier rapidement les erreurs dans les parties du système qui sont les plus à risque
- Permet de recevoir des commentaires sur le style de programmation ou le choix d'algorithmes
- Permet d'apprendre des erreurs des autres programmeurs
- Permet de concentrer l'effort de test automatisé sur les régions problématiques du programme

19

Revues structurées

- Similaire aux inspections
- Rôles :
  - Modérateur
  - Secrétaire
  - Programmeur (auteur du code en revue)
  - Testeur
- Déroulement :
  - Les participants se préparent avant la revue, comme pour une inspection
  - Le testeur apporte une courte liste de tests qui sont « exécutés » mentalement
    - Ces tests permettent de lancer la discussion
  - Le programmeur reçoit une liste d'erreurs (compilée par le secrétaire), comme pour l'inspection

20

Test automatisés

Niveaux de tests informatisés

- **Tests unitaires**: vérifient chaque classe ou module individuellement
- **Tests d'intégration**: vérifient les interactions entre plusieurs classes ou modules
  - Souvent utilisés lors de l'ajout d'un nouveau module à un groupe de modules (testés) existants
- **Tests de régression**: vérifient qu'un changement (ex: correction) n'a pas introduit de nouvelles erreurs
  - En théorie, tous les tests de régression devraient être exécutés après chaque changement
  - En pratique, exécuter tous les tests est trop coûteux
  - Certains outils permettent de choisir un sous-ensemble de tests à exécuter

Niveaux de tests informatisés

- **Tests de système**: vérifient que le système en entier est conforme aux spécifications
- **Tests de validation**: s'assurent que le système réponde aux besoins du client
- **Tests de marges** (*stress tests*): contrôlent la performance et la fiabilité d'un système dans des conditions similaires aux conditions normales d'opération
  - Ex: Augmenter graduellement le nombre de transactions d'un système distribué

23

Tests fonctionnels

- Aussi appelés tests « boîte noire » (*black box tests*)
- Identifient les fautes en se basant uniquement sur les entrées et les sorties
- Ne tiennent pas compte du comportement interne du programme
  - Ex: chemin exécuté pour le calcul, cas spéciaux
- La spécification et/ou les connaissances du domaine sont utilisées pour déterminer les cas à tester

24

Tests fonctionnels

- Avantages
  - Supportent plusieurs types de tests
  - Requièrent moins de ressources
- Inconvénients
  - Ne permettent pas d'identifier les cas où plusieurs erreurs s'annulent pour produire le bon résultat
  - Ne permettent pas d'évaluer la couverture du code
  - Ne permettent pas d'évaluer la qualité du code

25

Tests structurels

- Aussi appelés tests « boîte blanche » (*white box tests*) ou « boîte de verre » (*glass box tests*)
- Exploitent la connaissance de la structure du code
- Requièrent parfois des analyses pour s'assurer que le code a été couvert adéquatement
  - Par ligne de code
  - Par instruction
  - Par branche
  - …

26

Tests structurels

- Avantages
  - Permettent de vérifier l'implémentation d'un algorithme de façon précise
  - Permettent d'évaluer la quantité de code non-testé
  - Permettent d'évaluer la qualité du code et son niveau d'adhérence aux standards
- Inconvénient
  - Utilisent plus de ressources

27

Tests exhaustifs

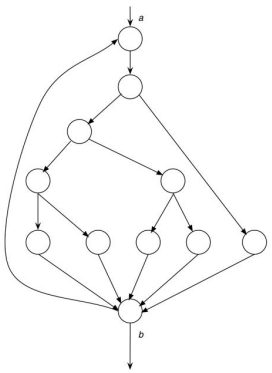
- Est-il possible de tester un programme **complèment**?
  - c'est-à-dire, trouver toutes les erreurs présentes
- Tests fonctionnels :
  - Pour tester le programme complètement, il faudrait tester **toutes** les entrées possibles
  - Une infinité de cas en général

28

Tests exhaustifs

- Tests structurels
  - Programme de ~ 50-100 lignes
  - 1 boucles (1-20 itérations), 4 conditions imbriquées à l'intérieur de la boucle

```
do
   if (…) then
        if (…) then
           if (…) then
           else …
        else …
           if (…) then
           else …
   else
while (i < 20)
```



~ 10^{14} chemins d'exécution possibles ($5^1 + 5^2 + … + 5^{20}$)
3174 ans pour tester tous les chemins à raison d'un chemin par milliseconde!

29

Tests exhaustifs

- En général, tester un programme de façon exhaustive est impossible
- Il faut choisir un sous-ensemble des tests qui maximise la probabilité de détecter les erreurs
- Il est possible d'utiliser des tests aléatoires, mais leur efficacité est faible pour tester le comportement attendu
  - « Fuzz testing » est une technique répandue qui consiste à tester les cas inattendus ou anormaux à l'aide de valeurs d'entrée générées aléatoirement, et permet d'augmenter la robustesse d'un programme
- Une meilleure approche : déterminer un ensemble de tests fonctionnels qui seront complémentés de tests structurels

30

Approches

Tests fonctionnels

- Partitionnement & classes d'équivalences
- Analyse des valeurs de marge

Tests structurels

- Couverture des instructions
- Couverture des décisions
- Couverture des conditions
- Couverture des conditions et décisions
- Couverture des conditions multiples

31

Partitionnement & classes d'équivalence

- Un bon test fonctionnel devrait posséder deux propriétés :
  - Il réduit le nombre de tests additionnels nécessaires pour atteindre un jeu de tests « raisonnable »

    => Un test devrait couvrir les plus d'aspects possible du programme de façon à minimiser le nombre de tests requis.
  - Il couvre un ensemble d'autres tests possibles.
    - Il indique la présence ou l'absence d'erreurs pour ses valeurs d'entrée spécifiques.

    => Les tests peuvent être partionnés en **classes d'équivalence**.
- Les tests qui appartiennent à la même class d'équivalence devraient identifier les mêmes erreurs
  - En pratique, il n'y a pas de certitude sur les classes d'équivalence, seulement des probabilités

32

Approche de partionnement

- Chaque condition imposée sur les entrées est partionnée en deux groupes :
  - Classes d'équivalence valides
  - Classes d'équivalence invalides
- Plage de valeurs (ex: 1..100)
  - Classes d'équivalence valides : 1..100
  - Classes d'équivalence invalides: < 1, > 100
- Nombre de valeurs (ex: 3 ou 4 arguments spécifiés)
  - Classes d'équivalence valides : {3,4} arguments spécifiés
  - Classes d'équivalence invalides : < 3 arguments, > 4 arguments spécifiés

33

Approche de partionnement

- Ensemble de valeurs prédéfinies (ex: « auto », « maison », « bateau »)
  - Classes d'équivalence valides : chaque valeur est sa propre classe d'équivalence
  - Classes d'équivalence invalides : une pour chaque valeur (ex: « moto » pour « auto »)
- « Doit être » (ex: le 1er caractère doit être une lettre)
  - Classes d'équivalence valides : le 1er caractère est une lettre
  - Classes d'équivalence invalides : le 1er caractère n'est pas une lettre
- NB : les classes d'équivalences peuvent être partionnées encore plus si il existe un doute que le programme ne traite pas tous les éléments d'une même classe de la même façon.

34

Identification des tests

- À partir des classes d'équivalences, un jeu de tests peut être défini comme suit :
  1. Numéroter toutes les classes d'équivalence
  2. Jusqu'à ce que toutes les classes d'équivalence **valides** soient couvertes par les tests :
     - Écrire un nouveau test qui couvre autant de classes d'équivalences non-couvertes que possible
  3. Jusqu'à ce que toutes les classes d'équivalences **invalides** soient couvertes par les tests :
     - Écrire un nouveau test qui couvre **une seule** des classes d'équivalences non-couvertes

- NB: L'étape 3 n'utilise qu'un test par classe car une valeur erronée en masque souvent une autre

35

Analyse des valeurs de marge

- En pratique, il est souvent bénéfique de tester les conditions de marge
- Cette approche est similaire au partionnement en classes d'équivalence, mais avec deux différences :
  - Les éléments aux limites d'une classe d'équivalence sont utilisés pour définir les tests, et non pas un seul représentant arbitraire pour la classe entière
  - Les résultats (sortie) sont aussi pris en compte

36

Procédure

1. Plage de valeurs
   - Écrire un test valide pour la valeur de début et celle de fin, et des tests invalides pour les conditions juste en dehors de la plage valide
   - ex: -1..1 : tester -1, 1, -1.001, 1.001
2. Nombre de valeurs
   - Écrire un test pour le nombre minimum, le nombre maximum, un de moins que le nombre minimum et un de plus que la nombre maximum
   - ex: 1..255 enregistrements: tester pour 0, 1, 255, 256

37

Procédure

3. Utiliser les règles 1 & 2 pour chaque condition de sortie
   - ex: le programme calcule le montant d'une déduction entre 0$ et $1500. Écrire des tests qui causent ces valeurs, et, si possible, des tests qui causent une valeur négative ou une valeur supérieure à $1500
   - ex: si un programme génère une liste d'au plus 4 éléments, écrire des tests qui génèrent 0, 1, 2, 3 et 4 éléments, et si possible un test qui cause la génération de plus de 4 éléments.
   - NB : Pour les tests qui ne peuvent pas être écrits, il est tout de même utile de considérer la possibilité
4. Si la sortie du programme est une liste ordonnée, se concentrer sur les premiers et derniers éléments
5. Chercher d'autres conditions de marge spécifique au programme

38

Couverture

- La **couverture** est une mesure d'exhaustivité des tests effectués
  - Idéalement, la couverture devrait être exprimée en fonction de tous les chemins d'exécution possibles
  - En pratique, des approximations sont utilisées
- Un ensemble de tests devraient toujours viser la couverture complète d'un programme
  - En pratique, il peut être impossible de couvrir tout le programme d'après une certaine approximation
  - Maximiser la couverture maximise augmente la probabilité de détecter les erreurs

39

Couverture des instructions

- Nécessite que chaque instruction du programme soit exécutée au moins une fois
- Un critère de couverture assez faible :

```
public int foo(int x, int y) {
    if (x != 0)
        y=x+1;
    return y;
}
```

  - Si un test appelle `foo(1,1)`, toutes les instructions sont exécutées, mais le cas où `y` est retournée sans modification n'est pas testé.
  - La condition `(x!=0)` peut être erronée sans que les tests le démontrent.

40

Couverture des décisions / branches

- Nécessite que chaque décision prenne chaque valeur possible au moins un fois (en général vrai ou faux)
  - Les conditions incluent les boucles (`while`, `for`, `do-while`, etc.), les instructions conditionnelles (`if-else`, `switch`)
  - Pour la plupart des programmes, la couverture des branches implique la couverture des instructions
    - Exception : les programmes sans décisions

```
public int foo(int x, int y) {
    if (x != 0)
        y=x+1;
    return y;
}
```

41

Couverture des conditions

- Nécessite que chaque **condition** qui fait partie d'une décision prenne chaque valeur possible au moins un fois (en général vrai ou faux)

```
public int foo(int x, int y) {
    if (x != 0 && y != 0)
        y=x+1;
    return y;
}
```

  - Il faudrait écrire au moins 2 tests, par exemple :
    - `foo(0, 0)`
    - `foo(1, 1)`

42

Couverture des conditions

- La couverture de conditions n'implique pas la couverture de branches :

```
public int foo(int x, int y) {
    if (x != 0 && y != 0)
        y = x+1;
    return y;
}
```

  - Considérez les tests suivants :
    - `foo(0, 1)`
    - `foo(1, 0)`
  - Chaque condition prend les deux valeurs possibles, mais l'instruction '`y = x+1`' n'est jamais exécutée

43

Couverture des branches/conditions

- Requiert que chaque décision prenne toutes les valeurs possibles, et que chaque condition d'une décision prenne toutes les valeurs possibles
  - Combinaisons des deux approches précédentes
- Problème : certaines valeurs peuvent en masquer d'autres

```
public int foo(int x, int y, int z) {
    if (x != 0 && (y != 0 || z < 0))
        y=x+1;
    return y;
}
```

  - Le test `foo(0,1,1)` aurait pu exposer le fait que la condition `z<0` est erronée, mais comme la condition `x!=0` masque le reste de l'expression, l'erreur n'est pas détectée.

44

Couverture de conditions multiples

- Requiert que toutes les combinaisons de conditions dans toutes les décision soient couvertes au moins une fois :
  - Implique toutes les approches précédentes

```
public int foo(int x, int y) {
    if (x != 0 && y != 0)
        y=x+1;
    return y;
}
```

  - Il faudrait écrire au moins 4 tests, par exemple :
    - foo(0,0)
    - foo(1,0)
    - foo(0,1)
    - foo(1,1)

45

Test unitaires

46

Tests unitaires

- Un test unitaire vise à tester une unité individuelle d'un programme
  - En général, une fonction, une méthode, une classe ou un module
- But : comparer le résultat d'une opération à sa spécification
  - Il faut tenter de démontrer que l'unité contredit sa spécification
- Motivation :
  - Permet d'identifier les erreurs plus facilement lorsqu'une partie restreinte du code est testée
    - La source de l'erreur doit se trouver dans l'unité testée
  - Permet de tester plusieurs unités en parallèle
  - Permet de tester une unité lorsque le système est encore incomplète

47

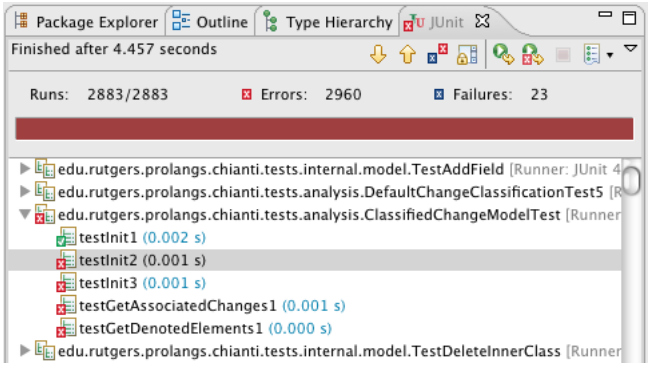
Écrire des tests unitaires

- Les tests unitaires sont souvent structurels
  - Comme les tests unitaires sont exécutés sur une portion très restreinte du code, les tests structurels sont beaucoup moins coûteux à effectuer qu'aux autres niveaux de tests
  - Les tests d'ordre supérieurs tentent souvent d'identifier d'autres types d'erreurs que les erreurs de programmation et de logique détectées par les tests structurels
- Les techniques de génération de tests peuvent être utilisées pour déterminer un ensemble de tests unitaires
- En TDD, les tests fonctionnels sont initialement générés avant de commencer l'implémentation
  - Les tests structurels sont ajoutés au fur et à mesure que l'implémentation progresse

48

Outils

- Les tests unitaires sont souvent automatisés
  - Un outil se charge d'exécuter tous les tests en séquence, et présente les résultats de façon visuelle



49

JUnit - Tests unitaires pour Java

- Permet d'exécuter un ou plusieurs **cas de test** (*test cases*) automatiquement
  - Le jugement d'un utilisateur n'est pas requis
- Organise les cas de tests par **suites**
  - Suite = regroupement de tests
- Les tests sont en général indépendants
  - L'état doit être recréé pour chaque test
  - Une **fixture** permet de définir des objets, mais les objets sont recréés **pour chaque test**!
- Utilise les annotations (@annotation) de Java pour spécifier le rôle des divers éléments

50

JUnit – Exemple

```
public class TestSuite {
    @Test
    public void testSomething() {
        // test
    }
}

org.junit.runner.JUnitCore.runClasses(TestSuite.class)
```

(ou utiliser l'intégration avec Eclipse, par exemple)

51

JUnit – Tester une valeur de retour

```
public class IntTests {
    @Test
    public void testParse() {
        int v = Integer.parseInt("-FF", 16);
        Assertions.assertEquals(-255, v);
    }
}
```

52

JUnit – Tester l'état interne

```
public class CollectionTests {
    @Test
    public void testCollectionAdd() {
        Collection collection = new ArrayList();
        assertEquals(0, collection.size());
        collection.add("itemA");
        assertEquals(1, collection.size());
        collection.add("itemB");
        assertEquals(2, collection.size());
    }
```

53

JUnit – Tester une exception

```
public class IntTests {
    @Test(expected=NumberFormatException.class)
    public void testParseException() {
        int v = Integer.parseInt("abc");
    }

    @Test
    public void testParseExceptionAlternate() {
        try {
            int v = Integer.parseInt("abc");
            Assert.fail("No exception thrown");
        } catch (NumberFormatException e) {
            // pass
        }
    }
}
```

54

Tester en présence de dépendances

- La plupart des modules requièrent d'autres modules pour leur bon fonctionnement
- Idéalement, chaque module devrait être testé en isolation
  - Débogage plus facile
  - Tests plus robustes en présence de changements dans le système
  - Permet une meilleure organisation des tests
- Comment isoler le comportement d'un module particulier?
  - Par la création d'un « faux module » synthétique qui joue le rôle de la dépendance pour les tests effectués
  - Par exemple, le faux module peut retourner des valeurs prédéterminées lorsqu'invoqué
  - (plus de détails à venir dans la prochaine section)

55

Tests d'intégration

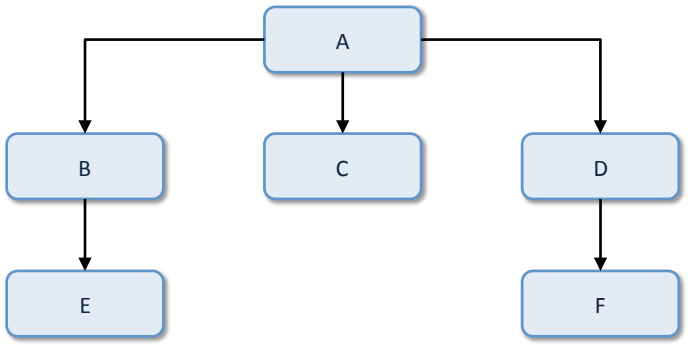
56

Stratégies de test

- **Non-incrémental (« Big bang »)**
  - chaque module est testé indépendamment, puis les modules sont assemblés pour former le système complet.
- **Incrémental**:
  - le prochain module est intégré aux modules testés existants avant d'être lui-même testé

57

Exemple - Test non-incrémental



- Comment gérer les dépendances entre les tests?

58

Pilotes & modules souches

- Chaque module à tester nécessite
  - un pilote (*driver*), qui effectue les tests
  - Les modules A, B et D nécessite des modules souches (*stubs*) qui jouent le rôle de leurs dépendances lors des tests
- Les pilotes et les modules souches sont de « faux modules » qui permettent de tester des systèmes incomplets
  - Pilotes
    - Lancent l'exécution des tests pour un module particulier
    - Une alternative consiste à utiliser un outil de test (ex: JUnit)
  - Modules souches
    - Jouent le rôle d'un autre module durant l'exécution des tests
    - Maintiennent un minimum d'état pour simuler le comportement réel d'un module dans le cadre d'un test précis
    - Sont généralement coûteux à développer

59

Exemple – Module souche

```
public interface MailService {
    public void send (Message msg);
}

public class MailServiceStub implements MailService {
    private List<Message> messages
            = new ArrayList<Message>();
    public void send (Message msg) {
        messages.add(msg);
    }
    public int numberSent() {
        return messages.size();
    }
}
```

Source: Martin Fowler

60

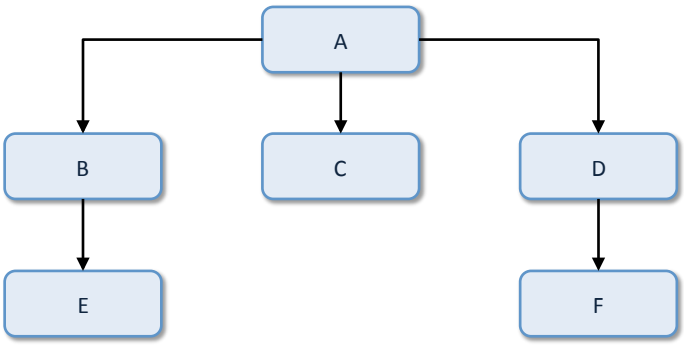
Exemple – Module souche

```
class OrderStateTester...
    public void testOrderSendsMailIfUnfilled() {
        Order order = new Order(TALISKER, 51);
        MailServiceStub mailer = new MailServiceStub();
        order.setMailer(mailer);
        order.fill(warehouse);
        assertEquals(1, mailer.numberSent());
    }
```

Source: Martin Fowler

61

Exemple – Test incrémental

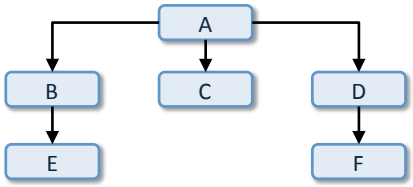


- Comment choisir l'ordre d'exécution des tests?

62

Ordonnancement des tests

- Une options possible :
  - Tester E, C, F en parallèle
  - Tester B avec E, D avec F
  - Tester A avec B, C et D

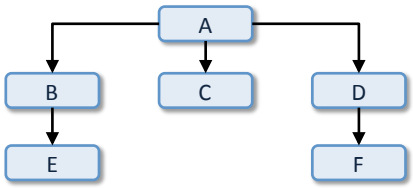


- Remarques :
  - La création de modules souches n'est plus nécessaire
  - Les pilotes doivent être créés pour chaque module testé

63

Ordonnancement des tests

- Une alternative :
  - Tester A
  - Tester B, C et D avec A
  - Tester E avec B, F avec D



- Remarques :
  - La création de pilotes n'est plus nécessaire
  - Les modules souches doivent être créés pour chaque module testé

64

Observations

- Le test incrémental
  - est moins coûteux que le test non-incrémental
    - Nécessite des pilotes ou des modules souches, mais pas les deux à la fois
  - permet d'identifier les erreurs de communication entre modules plus rapidement
  - facilite de débogage
  - teste les modules de façon plus approfondie
    - Un vrai module est utilisé à la place de pilote ou de souche, et donc reçoit plus d'attention lors de tests
  - Réduit la possibilité d'exécuter des tests en parallèle

65

Test incrémental - Direction

- Haut en bas
  - Permet de démontrer les fonctions du système plus rapidement, et donc de faciliter la validation
  - Permet de détecter plus efficacement les erreurs structurelles dans un système
  - Lorsque les fonctions d'entrée / sortie sont ajoutées
    - de vraies données peuvent être utilisées comme tests
    - l'observation des résultats de tests être grandement simplifiée
  - Rend l'écriture de tests très difficile dans certains cas
- Bas en haut
  - Retarde la vérification et la validation du système en entier
    - Le système n'existe que lorsque le dernier module est ajouté
  - Implémentation et observation des tests beaucoup plus facile

66

Doublures de tests

- Objets synthétiques qui permettent l'exécution de tests
  - **Dummy objects**: objets qui sont passés mais jamais utilisés (ex: paramètres)
  - **Fake objects**: objets qui possèdent une implémentation qui est impropre à un environnement de production
    - Ex: base de données en mémoire
  - **Stub objects**: objets qui fournissent des réponses préprogrammées pour des cas précis
    - Peuvent maintenir un état interne simplifié
  - **Mock objects**: objets qui renferment une spécification des appels qui devront être reçus
    - Permettent la vérification du comportement

Mock objects - Exemple

```
public class OrderEasyTester extends TestCase {
  private static String TALISKER = "Talisker";

  private MockControl warehouseControl;
  private Warehouse warehouseMock;

  public void setUp() {
    warehouseControl = MockControl.createControl(Warehouse.class);
    warehouseMock = (Warehouse) warehouseControl.getMock();
  }
```

Source: Martin Fowler

Mock objects - Exemple

```
public void testFillingRemovesInventoryIfInStock() {
  //setup - data
  Order order = new Order(TALISKER, 50);

  //setup - expectations
  warehouseMock.hasInventory(TALISKER, 50);
  warehouseControl.setReturnValue(true);
  warehouseMock.remove(TALISKER, 50);
  warehouseControl.replay();

  //exercise
  order.fill(warehouseMock);

  //verify
  warehouseControl.verify();
  assertTrue(order.isFilled());
}
```

Source: Martin Fowler 69

Mock objects - Exemple

```
public void testFillingDoesNotRemoveIfNotEnoughInStock() {
  Order order = new Order(TALISKER, 51);

  warehouseMock.hasInventory(TALISKER, 51);
  warehouseControl.setReturnValue(false);
  warehouseControl.replay();

  order.fill((Warehouse) warehouseMock);

  assertFalse(order.isFilled());
  warehouseControl.verify();
 }
}
```

Source: Martin Fowler 70

Mock objects – Exemple (autre syntaxe)

```
class OrderInteractionTester...
  public void testOrderSendsMailIfUnfilled() {
    Order order = new Order(TALISKER, 51);
    Mock warehouse = mock(Warehouse.class);
    Mock mailer = mock(MailService.class);
    order.setMailer((MailService) mailer.proxy());

    mailer.expects(once()).method("send");
    warehouse.expects(once()).method("hasInventory")
      .withAnyArguments()
      .will(returnValue(false));

    order.fill((Warehouse) warehouse.proxy());
  }
}
```

Source: Martin Fowler

71

Tests d'ordre supérieur

72

Tests d'ordre supérieur

- Même après avoir testé tous les modules d'un système, celui-ci peut tout de même contenir des erreurs :
  - Lorsque le système se comporte d'une façon contraire aux attentes raisonnables d'un utilisateur, il contient encore des erreurs
- Il donc nécessaire d'effectuer des tests additionnels à un niveau de granularité plus élevé
  - Il ne faut pas répéter les tests des niveaux précédents, mais bien tenter d'identifier d'autres types d'erreurs que celles visées par les tests unitaires et d'intégration
- Ces tests d'ordre supérieur considèrent le système en entier
- Ces tests ont comme objectif :
  - La vérification
  - La validation

73

Vérification & validation

- **Vérification:** Le logiciel est-il développé correctement? (*Are we building the product right?*)
  - Processus d'évaluation d'un système ou composant afin de déterminer si les conditions imposées au début d'une phase de développement sont satisfaites
- **Validation:** Est-ce que le bon logiciel est développé? (*Are we building the right product?*)
  - Processus d'évaluation d'un système ou composant afin de déterminer s'il correspond aux besoins du client

74

Tests de système

- Tests de fonctionnalité
  - Utilisent la spécification des besoins fonctionnels (ex: cas d'utilisation) pour tester les fonctionnalités du système
  - Généralement des tests fonctionnels, puisque les tests structurels sont fréquemment exécutés à un niveau de granularité plus bas
  - Les techniques présentées précédemment (ex: partitionnement) peuvent être utilisées
- Tests de volume
  - Permettent de tester si un programme est capable de traiter un grande volume de données
    - Ex: un très gros programme pour un compilateur, un circuit contenant des milliers de composantes pour un simulateur, etc.

75

Tests de système

- Tests de marge (*stress tests*)
  - Permettent de tester si un programme peut traiter un certain volume de données sur une **période de temps** donnée
    - Ex: 100 transactions par minute, 1000 utilisateurs simultanément, 25 processus concurrents, 500 mb de données par heure, etc.
  - Ne conviennent pas à tous les types de programmes
    - Ex: compilateur
  - Devraient tester non seulement le niveau de stress attendu mais aussi un stress plus élevé:
    - Ex: 50 appels concurrents attendus, tests avec 50 appels puis > 50 appels
  - Permettent d'éviter des situations de *Denial of Service (DoS)*

76

Tests de système

- Tests de facilité d'utilisation
  - Permettent de tester les facteurs humains (ex: l'interface graphique)
  - Quelques questions auxquelles les tests d'utilisation tentent de répondre :
    - Le système est-t-il facile à utiliser (navigation dans les menus, interface claire, etc.)?
    - L'interface est-elle uniforme et cohérente?
    - Les messages d'erreur sont-ils facile à comprendre et informatifs? (ex: « une erreur inconnue s'est produite », « erreur: code 0x43d5 », etc.)
    - Le système donne-t-il une indication immédiate après une entrée de l'utilisateur?
    - Le système présente-t-il une liste d'options rarement utilisées / excessives?

77

Tests de système

- Tests de sécurité
  - Permettent de déterminer si le système contient des failles de sécurité exploitables
    - Ex: SQL injection, XSS, buffer overflows, etc
- Tests de performance
  - Permet de déterminer si le programme rencontre ses objectifs de performance
    - Ex: un cycle de traitement ne doit pas dépasser 50ms
- Tests de stockage
  - Permet de déterminer si le programme rencontre ses objectifs d'utilisation de la mémoire
    - Ex: quantité de mémoire vive utilisée, taille des fichiers temporaires, etc.

78

Tests de système

- Tests de configuration
  - Permettent de déterminer si le système fonctionne correctement avec différentes configurations matérielles et/où logicielles
    - Ex: différentes architectures (x86, ARM, PowerPC), différentes plateformes (linux, Mac OS, Microsoft Windows), différentes quantités de mémoire, etc.
- Tests de compatibilité / conversion
  - Permettent de déterminer si un système remplace adéquatement un système existant
    - Ex: migration d'un système de base de données

79

Tests de système

- Tests d'installation
  - Appropriés pour les systèmes qui ont un processus d'installation complexe
  - Contrairement aux autres types de tests, ne tentent pas d'identifier des erreurs dans le système développé mais plutôt dans son processus d'installation
  - Exemples d'activités :
    - S'assurer que l'usager a choisi un ensemble d'options compatibles
    - S'assurer que tous les fichiers ont été créés correctement
    - S'assurer que toutes les parties du système sont présentes
- Tests de fiabilité
  - Permettent de déterminer si niveau de fiabilité d'un système est adéquat
    - Ex: 99.999% de fonctionnement
  - Peut nécessiter des méthodes formelles, statistiques ou autres techniques avancées

80

Tests de système

- Tests de récupération (*recovery*)
  - Permettent de déterminer si le système peut récupérer après une défaillance
  - Des fautes peuvent être manuellement injectées dans le système pour déterminer comment il pourra récupérer
    - Ex: fautes de lecture/écriture de données, pointeur invalide, bruit sur une ligne de communication, etc.
  - Les spécifications spécifient souvent les limites du temps de récupération (*mean time to recovery*, ou *MTTR*)
- Tests d'entretien / maintenance
  - Permettent de déterminer si le programme peut être facilement maintenu en bon état de fonctionnement
    - Ex: image de la mémoire (*memory dump*), procédures de maintenance, temps moyen de débogage, etc.

81

Tests de système

- Tests de documentation
  - Permettent de déterminer la clarté et l'exactitude de la documentation
  - La documentation peut être sujette à une inspection
  - La documentation peut être utilisée pour spécifier les autres tests (permet de vérifier si le comportement actuel du système correspond à la documentation)
- Tests de procédure
  - Permettent de vérifier les procédures non-automatisées
    - Ex: procédure suivie par un administrateur pour sauvegarder ou restaurer la base de données

82

Tests d'acceptation

- Objectif : comparer le programme aux besoins réels des utilisateurs (validation)
- Contrairement aux autres types de tests, les tests d'acceptation sont effectués par le client
  - Il est utile de spécifier un ensemble de cas à tester en fonction du contrat initial
  - Le client devrait, comme pour les autres types de tests, tenter de démontrer que le programme ne rencontre pas les besoins

83

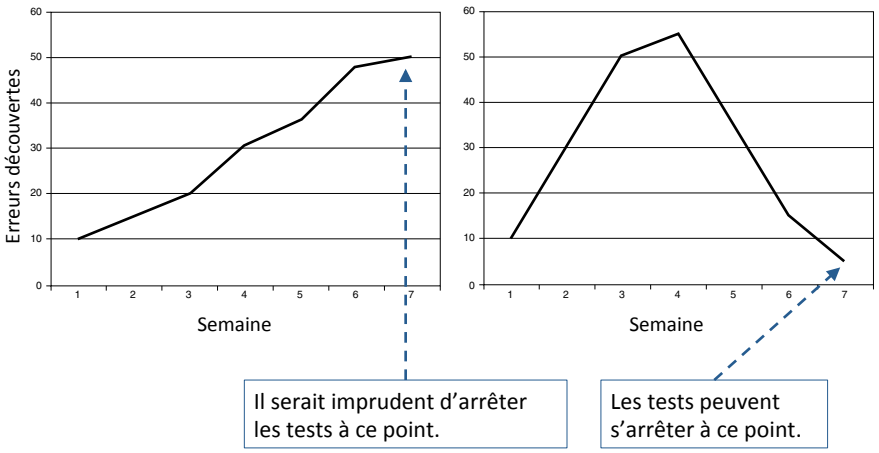
Quand arrêter de tester?

- Arrêter lorsque le temps réservé aux tests est écoulé
  - Ce critère peut être satisfait même si aucun test n'est effectué!
- Arrêter lorsque les tests ne révèlent plus d'erreurs
  - Aussi indépendant de la qualité des tests
- Arrêter lorsqu'une méthodologie a été suivie pour la création des tests et que les tests ne révèlent plus d'erreurs
  - Assure un certain niveau de qualité des tests
  - Ne convient pas aux tests pour lesquels une méthodologie n'est pas disponible (ex: tests de système)
  - Est subjectif et difficile à appliquer en pratique

84

Quand arrêter de tester?

- Arrêter lorsque la fréquence de découverte de nouvelles erreurs est en baisse



Source: Myers et. al., "The Art of Software Testing", Wiley 2004.

85

Quand arrêter de tester?

- Arrêter lorsqu'un nombre d'erreurs ont été trouvées
  - Ex: le programme sera testé jusqu'à ce que 90% des erreurs ait été trouvées, ou 3 mois, selon le dernier critère atteint
  - Nécessite un estimé du nombre d'erreurs dans le système
  - Quoi faire si l'estimé surestime le nombre d'erreurs?
    - Il se peut que le critère voulu ne soit jamais atteint
    - Il est difficile de déterminer si les erreurs n'existent pas dans le code, ou si les tests ne les ont simplement pas encore découverts
    - Une agence de test externe (indépendante) peut aider à faire un choix éclairé

86

Estimation du nombre d'erreurs

- Par comparaison avec des projets antérieurs
- Par étude des premières phases de tests
  - Un modèle statistique peut prédire le nombre d'erreurs en fonction de données initiales du projet
- Par utilisation de moyennes empiriques
  - Ex: 4 à 8 erreurs en moyenne par 100 lignes de code
- Par introduction d'erreurs simulées
  - Des erreurs sont introduites dans le programme, et les tests sont effectués pour savoir combien de ces erreurs sont identifiées
- Par comparaison de deux équipes indépendantes
  - Deux équipes indépendantes tentent d'identifier les erreurs en parallèle
  - Les erreurs identifiées par les deux équipes sont utilisées pour estimer le nombre total d'erreurs

87

Planification de tests

Un rapport de test devrait contenir les sections suivantes :

1. **Objectifs** : décrit les objectifs pour chaque phase de test
2. **Critères d'arrêt** : définit quand chaque phase de test sera considérée comme complétée
3. **Échéancier** : décrit quand les tests de chaque phase seront conçus, écrits et exécutés
4. **Responsabilités** : pour chaque phase, définit qui sera responsable de concevoir, écrire et exécuter les tests, ainsi que qui sera responsable de corriger les erreurs découvertes.
5. **Standards et organisation des tests** : les grands projets nécessitent souvent une manière systématique d'organiser et d'enregistrer les tests
6. **Outils** : décrit quels outils seront utilisés pour les tests, qui sera responsable de les développer ou les acquérir, et comment ils seront utilisés

88

Planification de tests

7. **Temps d'utilisation des machines** : donne un estimé du temps d'utilisation des machines nécessaire pour chaque phase de test
   - Ex: compilation, machines utilisées pour les tests d'installation, serveurs qui servent au tests d'une application web, etc.
8. **Configuration matérielle** : si une configuration spéciale est nécessaire
9. **Intégration** : décrit l'ordre selon lequel le programme sera assemblé progressivement à partir de ses modules.
10. **Procédures de suivi** : permet de définir comment le suivi des tests sera effectué (ex: modules à haut risque, progrès par rapport à l'échéancier)
11. **Procédures de débogage** : permet de définir comment le suivi des corrections sera effectué
12. **Tests de régression** : définit les tests qui seront effectués après un changement au système de façon à s'assurer que des régressions n'ont pas été introduites

89

Application web typique

Architecture à 3 niveaux (*3-tier*)

90

Tests d'applications web

- Les applications web présentent certains défis :
  - **Utilisateurs** : ont des niveaux de compétence variés, des environnements différents (fureteurs, systèmes d'exploitation, connexion internet), et provenant de pays différents
  - **Environnement complexe** : qui doit être répliqué pour effectuer les tests afin de pouvoir identifier un maximum d'erreurs
  - **Sécurité** : les applications web sont accessibles en ligne et donc vulnérable aux attaques
- Tester le couches séparément permet d'identifier plus d'erreurs et de déterminer leur cause plus facilement

91

Tester la couche de présentation

- La couche de présentation doit maintenir une excellent qualité
  - Les utilisateurs ne paient en général pas pour les services, leur loyauté peut donc facilement basculer si le système ne rencontre pas leurs attentes
- Niveaux de tests :
  - Contenu : esthétique, polices de caractères, couleurs, orthographe et grammaire, facilité d'utilisation, etc.
  - Architecture du site : absence de liens brisés, navigabilité, etc.
  - Environnement : différents fureteurs, différentes versions d'un même fureteur, différents systèmes d'exploitation, etc.
  - Scripts : une attention particulière doit être portée aux pages qui utilisent des scripts du côté client (ex: Javascript)

92

Tester la couche de logique d'application

- Cette couche ressemble aux applications plus conventionnelles. La majorité de techniques présentées sont donc utilisables.
- Quelques considérations additionnelles :
  - **Validité des entrées** : des tests fonctionnels doivent être développés pour vérifier que les entrées invalides sont traitées correctement
    - Les utilisateurs entre souvent des données incorrectes ou dans un format différent de celui qui est attendu (ex: numéro de téléphone)
    - Des attaquants peuvent entrer des valeurs incorrectes pour tenter de compromettre le système (ex: SQL injection)
  - **Performance** : il faut s'assurer, à l'aide de tests de marge, que l'application peut atteindre de débit de traitement nécessaire
  - **Transactions** : il faut s'assurer que les transactions se terminent correctement, et que les transactions annulées sont repliées correctement

93

Tester la couche de données

- Temps de réponse : la base de données doit répondre rapidement aux requêtes pour éviter d'introduire un goulot d'étranglement dans le système.
- Intégrité des données : les données doivent être enregistrées et chargées correctement.
  - Type et taille des champs de la base de données
  - Caractères multilingues
- Tolérance aux fautes et récupération : lorsqu'une défaillance de la base de données se produit, il faut en général pouvoir continuer à offrir des services
  - On peut passer à une nouvelle base de données
  - On doit mettre en place et tester des procédures de sauvegarde et de récupération des données

94