

Automatisation du Test Logiciel

Sébastien Bardin

CEA-LIST, Laboratoire de Sûreté Logicielle

`sebastien.bardin@cea.fr`

`http://sebastien.bardin.free.fr`

Compétences à aquérir

- connaître les principes généraux du test logiciel
 - comprendre l'état de la technique en automatisation du test logiciel
 - avoir un aperçu des tendances et de l'état de l'art académique
-

Pour quelles industries ?

- bagage général de l'honnête informaticien [Agile Programming, etc.]
- systèmes critiques
- systèmes “de qualité” (sécurité, etc.)

- Introduction au test logiciel
- Exécution symbolique pour l'automatisation du test
- Exécution symbolique et critères de test avancés

■ Introduction au test logiciel

- ▶ Contexte
- ▶ Définition du test
- ▶ Éléments de classification
- ▶ Aspects pratiques
- ▶ Point de vue industriel
- ▶ Complément : critères boîte noire
- ▶ Complément : critères boîte blanche
- ▶ Complément : tests de régression
- ▶ Discussion

Coût des bugs

- Coûts économique : 64 milliards \$/an rien qu'aux US (2002)
- Coûts humains, environnementaux, etc.

Nécessité d'assurer la qualité des logiciels

Domaines critiques

- atteindre le (très haut) niveau de qualité imposée par les lois/normes/assurances/... (ex : DO-178B pour aviation)

Autres domaines

- atteindre le rapport qualité/prix jugé optimal (c.f. attentes du client)

Motivations (2)

Validation et Vérification (V & V)

- **Vérification** : est-ce que le logiciel fonctionne correctement ?
 - ▶ *“are we building the product right ?”*
- **Validation** : est-ce que le logiciel fait ce que le client veut ?
 - ▶ *“are we building the right product ?”*

Quelles méthodes ?

- revues
- simulation/ animation
- tests méthode de loin la plus utilisée
- méthodes formelles encore très confidentielles, même en syst. critiques

Coût de la V & V

- 10 milliards \$/an en tests rien qu'aux US
- plus de 50% du développement d'un logiciel critique (parfois > 90%)
- en moyenne 30% du développement d'un logiciel standard

- La vérification est une part cruciale du développement
- Le test est de loin la méthode la plus utilisée
- Les méthodes manuelles de test passent très mal à l'échelle en terme de taille de code / niveau d'exigence
- fort besoin d'automatisation

■ Introduction au test logiciel

- ▶ Contexte
- ▶ Définition du test
- ▶ Éléments de classification
- ▶ Aspects pratiques
- ▶ Point de vue industriel
- ▶ Complément : critères boîte noire
- ▶ Complément : critères boîte blanche
- ▶ Complément : tests de régression
- ▶ Discussion

Le test est une méthode dynamique visant à trouver des bugs

Tester, c'est exécuter le programme dans l'intention d'y trouver des anomalies ou des défauts

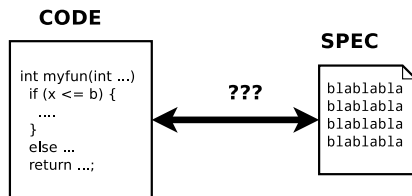
- G. J. Myers (The Art of Software Testing, 1979)

1- Construire la qualité du produit

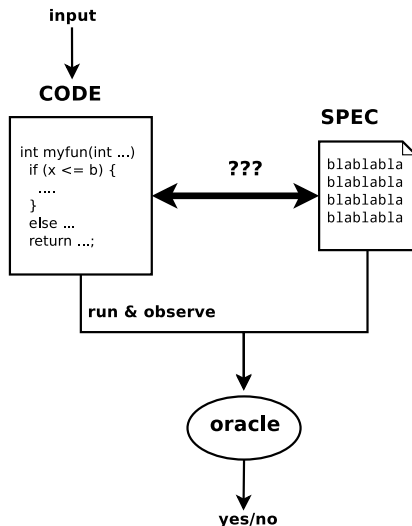
- lors de la phase de conception / codage
- en partie par les développeurs (tests unitaires)
- but = trouver rapidement le plus de bugs possibles (avant la commercialisation)
 - ▶ test réussi = un test qui trouve un bug

2- Démontrer la qualité du produit à un tiers

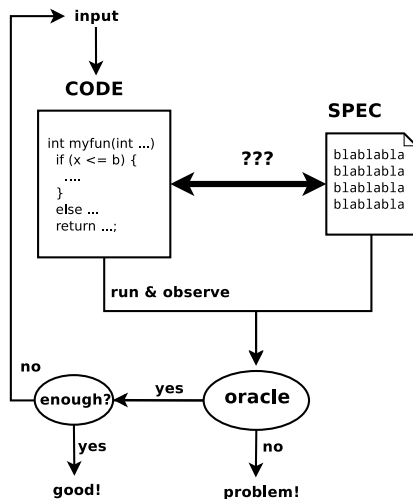
- une fois le produit terminé
- idéalement : par une équipe dédiée
- but = convaincre (organismes de certification, hiérarchie, client)
 - ▶ test réussi = un test qui passe sans problème
 - ▶ + tests jugés représentatifs
(systèmes critiques : audit du jeu de tests)



Process du test (1)



Process du test (1)



- 1 choisir un cas de test (CT) [\approx scénario] à exécuter
- 2 estimer le résultat attendu [oracle]
- 3 déterminer une donnée de test (DT) exerçant le CT, et son oracle concret [concrétisation]
- 4 exécuter le programme sur la DT [script de test]
- 5 comparer le résultat obtenu à l'oracle [verdict : pass/fail]
- 6 a-t-on assez de tests ? si oui stop, sinon goto 1 [notion de critère de couverture]

Suite / Jeu de tests : ensemble de cas de tests

```
int[] my-sort (int[] vec)  
//@ Ensures : tri du tableau d'entrée + élimination de la redondance
```

Quelques cas de tests (CT) et leurs oracles :

| | | |
|-----|-----------------------------------|----------------------|
| CT1 | tableau d'entiers non redondants | le tableau trié |
| CT2 | tableau vide | le tableau vide |
| CT3 | tableau avec 2 entiers redondants | trié sans redondance |

Concrétisation : DT et résultat attendu

| | | |
|-----|-------------------------|----------------------|
| DT1 | vec = [5,3,15] | res = [3,5,15] |
| DT2 | vec = [] | res = [] |
| DT3 | vec = [10,20,30,5,30,0] | res = [0,5,10,20,30] |

Script de test

```
1 void testSuite() {
2
3     int [] td1 = [5,3,15] ; /* prepare data */
4
5     int [] oracle1 = [3,5,15] ; /* prepare oracle */
6
7     int [] res1 = my-sort(td1); /* run CT and */
8                                 /* observe result */
9
10    if (array-compare(res1,oracle1)) /* assess validity */
11    then print('test1 ok')
12    else {print('test1 erreur')};
13
14
15
16    ... /* same for other test data */
17
18
19 }
```


- ✗ choisir les cas de test / données de test à exécuter
- ✗ estimer le résultat attendu [oracle]
- ✓ exécuter le programme sur les données de test
 - . attention : systèmes embarqués / cyber-physiques
 - . attention : niveau unitaire, code incomplet ! [stubs, mocks]
- ✓ comparer le résultat obtenu à l'oracle
 - . attention : systèmes embarqués / cyber-physiques
- ✓ a-t-on assez de tests ? si oui stop, sinon goto 1
 - . attention : calcul couverture ✓ , choix du critère de couverture ●
- ✓ rejouer les tests à chaque changement [test de régression]
 - . attention : rejeu ✓ , maintenance / optimisation ●

pour les ✓ , des solutions standard existent, doivent être appliquées!!

Qu'apporte le test ?

Le test ne peut pas prouver au sens formel la validité d'un programme

Testing can only reveal the presence of errors but never their absence.

- E. W. Dijkstra (Notes on Structured Programming, 1972)

Par contre, le test peut "augmenter notre confiance" dans le bon fonctionnement d'un programme

- correspond au niveau de validation des systèmes non informatiques

Un bon jeu de tests doit donc :

- exercer un maximum de "comportements différents" du programme
[notion de critères de couverture]
- notamment
 - ▶ tests nominaux : cas de fonctionnement les plus fréquents
 - ▶ tests de robustesse : cas limites / délicats





Beware of bugs in the above code ; I have only proved it correct, not tried it.

- Donald Knuth (1977)

It has been an exciting twenty years, which has seen the research focus evolve [...] from a dream of automatic program verification to a reality of computer-aided [design] debugging.

- Thomas A. Henzinger (2001)

Livres

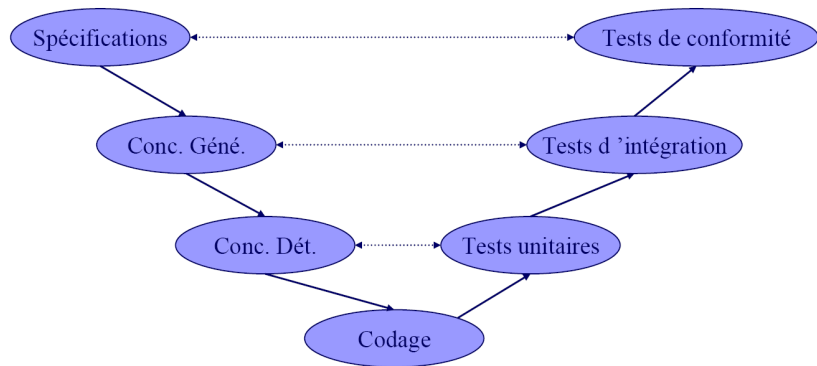
-  Introduction to software testing [Ammann-Offutt 08]
-  Foundations of Software Testing [Mathur 08]
-  Art of Software Testing (2nd édition) [Myers 04]
-  Software Engineering [Sommerville 01]

■ Introduction au test logiciel

- ▶ Contexte
- ▶ Définition du test
- ▶ Éléments de classification
- ▶ Aspects pratiques
- ▶ Point de vue industriel
- ▶ Complément : critères boîte noire
- ▶ Complément : critères boîte blanche
- ▶ Complément : tests de régression
- ▶ Discussion

- correction fonctionnelle
- robustesse
- performances
- sécurité
- ergonomie
- ...

Selon le cycle de développement (1)



aussi : phase d'évolution / maintenance : tests de (non) régression

Selon le cycle de développement (2)

Tests unitaire : tester les différents modules en isolation

- définition non stricte de “module unitaire” (procédures, classes, packages, composants, etc.)
- uniquement test de correction fonctionnelle

Tests d'intégration : tester le bon comportement lors de la composition des modules

- uniquement test de correction fonctionnelle

Tests système / de conformité : valider l'adéquation du code aux spécifications

- on teste aussi toutes les caractéristiques émergentes
sécurité, performances, etc.

Tests de validation / acceptance : valider l'adéquation aux besoins du client

- souvent similaire au test système, mais réaliser / vérifier par le client

Tests de régression : vérifier que les corrections / évolutions du code n'ont pas introduits de bugs

Boîte Noire : à partir de spécifications

- dossier de conception
- interfaces des fonctions / modules
- modèles formels ou semi-formels

Boîte Blanche : à partir du code

- critères basés sur le graphe de flôt de contrôle
- critères basés sur le graphe de flôt de données
- mutations syntaxiques

Boîte Noire : à partir de spécifications

- dossier de conception
- interfaces des fonctions / modules
- modèles formels ou semi-formels

Boîte Blanche : à partir du code

- critères basés sur le graphe de flôt de contrôle
- critères basés sur le graphe de flôt de données
- mutations syntaxiques

Probabiliste : *domaines des entrées + arguments statistiques*

- *random : distribution uniforme [ou pas ...]*
- *statistique : distribution mimant le profil opérationnel attendu*

Selon la source de cas de tests (2)

| | boîte noire | boîte blanche |
|-----------------------|----------------------------------|--|
| nécessite le code ? | non | oui |
| fautes de conformance | oui | erreur fonctionnelle : oui omission de fonctionnalité : non |
| fautes de codage | non | oui |
| critères classiques | partition des entrées limites | instructions, branches MCDC |
| # tests | ok | potentiellement énorme |
| pb spécifique | concrétisation des CT en DT | oracle |
| quel niveau ? | tous | plutôt unitaire (ou critères light : couv. fonctions) |

Les deux familles de tests BB et BN sont complémentaires

- Les approches structurelles trouvent plus facilement les défauts de programmation
- Les approches fonctionnelles trouvent plus facilement les erreurs d'omission et de spécification

Spec : retourne la somme de 2 entiers modulo 20 000

```
fun (x:int, y:int) : int =  
  if (x=500 and y=600) then x-y           (bug 1)  
  else x+y                                (bug 2)
```

- fonctionnel : bug 1 difficile, bug 2 facile
- structurel : bug 1 facile, bug 2 difficile

Les deux familles de tests BB et BN sont complémentaires

- Les approches structurelles trouvent plus facilement les défauts de programmation
- Les approches fonctionnelles trouvent plus facilement les erreurs d'omission et de spécification

Spec : retourne la somme de 2 entiers modulo 20 000

```
fun (x:int, y:int) : int =
```

- fonctionnel : bug 1 difficile, bug 2 facile
- structurel : bug 1 facile, bug 2 difficile

Les deux familles de tests BB et BN sont complémentaires

- Les approches structurelles trouvent plus facilement les défauts de programmation
- Les approches fonctionnelles trouvent plus facilement les erreurs d'omission et de spécification

```
fun (x:int, y:int) : int =  
  if (x=500 and y=600) then x-y           (bug 1)  
  else x+y                                (bug 2)
```

- fonctionnel : bug 1 difficile, bug 2 facile
- structurel : bug 1 facile, bug 2 difficile

Les deux familles de tests BB et BN sont complémentaires

- Les approches structurelles trouvent plus facilement les défauts de programmation
- Les approches fonctionnelles trouvent plus facilement les erreurs d'omission et de spécification

Spec : retourne la somme de 2 entiers modulo 20 000

```
fun (x:int, y:int) : int =  
  if (x=500 and y=600) then x-y           (bug 1)  
  else x+y                                (bug 2)
```

- fonctionnel : bug 1 difficile, bug 2 facile
- structurel : bug 1 facile, bug 2 difficile

Exemples de complémentarités des critères

Bug du Pentium : cause = erreurs sur quelques valeurs (parmis des millions) dans une table de calculs

- impossible à trouver en test boîte noire
- aurait pu être trouvé facilement en test unitaire du module concerné

Bug du “Mars Climate Orbiter” : cause = problème de métriques (mètres vs pouces)

- chaque module fonctionne correctement, test unitaire inutile
- aurait pu être détecté en phase de tests d'intégration, en se basant sur l'interface de communication (boîte noire)

Les données sont choisies dans leur domaine selon une loi statistique

- loi uniforme (test aléatoire)
- loi statistique du profil opérationnel (test statistique)

Pros/Cons du test aléatoire

- sélection aisée des DT en général
- test massif si oracle (partiel) automatisé
- “objectivité” des DT (pas de biais)
- PB : peine à produire des comportements très particuliers (ex : $x=y$ sur 32 bits)

Pros/Cons du test statistique

- permet de déduire une garantie statistique sur le programme
- trouve les défauts les plus probables : défauts mineurs ?
- PB : difficile d’avoir la loi statistique

■ Introduction au test logiciel

- ▶ Contexte
- ▶ Définition du test
- ▶ Éléments de classification
- ▶ Aspects pratiques
 - oracle
 - exécution
 - tests de régression et JUnit
 - dilemmes du test
 - critères de couvertures
- ▶ Point de vue industriel
- ▶ Complément : critères boîte noire
- ▶ Complément : critères boîte blanche
- ▶ Complément : tests de régression
- ▶ Discussion

- ✗ choisir les cas de test / données de test à exécuter
- ✗ estimer le résultat attendu [oracle]
- ✓ exécuter le programme sur les données de test
 - . attention : systèmes embarqués / cyber-physiques
 - . attention : niveau unitaire, code incomplet ! [stubs, mocks]
- ✓ comparer le résultat obtenu à l'oracle
 - . attention : systèmes embarqués / cyber-physiques
- ✓ a-t-on assez de tests ? si oui stop, sinon goto 1
 - . attention : calcul couverture ✓ , choix du critère de couverture ●
- ✓ rejouer les tests à chaque changement [test de régression]
 - . attention : rejeu ✓ , maintenance / optimisation ●

pour les ✓ , des solutions standard existent, doivent être appliquées!!

- oracle
- exécution
- tests de régression et JUnit
- dilemmes
- critères de couverture

La définition de l'oracle est un problème très difficile

- limite fortement certaines méthodes de test (ex : test random)
- impose un trade-off avec la sélection de tests
- point le plus mal maîtrisé pour l'automatisation

Quelques cas pratiques d'oracles parfaits automatisables [à reconnaître !]

- résultat simple à vérifier (ex : solution d'une équation)
- comparer à une référence : tables de résultats, logiciel existant
- disponibilité d'un logiciel similaire : test dos à dos
 - ▶ aussi : version précédente
 - ▶ aussi : version courante, mais options différentes (opt vs no-opt)

Des oracles partiels mais automatisés peuvent être utiles

- oracle basique : pass/crash
- instrumentation du code (assert)
- plus évolué 1 : instrumentation dynamique (Valgrind)
- plus évolué 2 : programmation avec contrats (Eiffel, Jml, etc.)

- oracle
- **exécution**
- tests de régression et JUnit
- dilemmes
- critères de couverture

- . Code manquant (test incrémental)
- . Présence d'un environnement (réseau, Base de Données, etc.)
- . Exécution d'un test très coûteuse en temps
- . Hardware réel non disponible, ou peu disponible
- . Possibilité de réinitialiser le système ?
 - si non, l'ordre des tests devient très important
- . Moyens d'observation et d'action sur le système
 - sources dispo, compilables et instrumentables : cas facile, script = code
 - si non : difficile, "script de test" = succession d'opérations (manuelles ?) sur l'interface disponible (informatique ? électronique ? mécanique ?)

- oracle
- exécution
- tests de régression et JUnit
- dilemmes
- critères de couverture

Tests de régression : à chaque fois que le logiciel est modifié, s'assurer que "ce qui fonctionnait avant fonctionne toujours"

Pourquoi modifier le code déjà testé ?

- correction de défaut
- ajout de fonctionnalités

Quand ?

- en phase de maintenance / évolution
- ou durant le développement

Quels types de tests ?

- tous : unitaires, intégration, système, etc.

Objectif : avoir une méthode automatique pour

- rejouer automatiquement les tests *nécessaires* [perfs !]
- détecter les tests dont les scripts ne sont plus corrects

JUnit pour Java : idée principale = tests écrits en Java

- simplifie l'exécution et le rejeu des tests (juste tout relancer)
- simplifie la détection d'une partie des tests non à jour : tests recompilés en même temps que le programme
- simplifie le stockage et la réutilisation des tests (tag @Test)

JUnit offre :

- des primitives pour créer un test (assertions)
- des primitives pour gérer des suites de tests
- des facilités pour l'exécution des tests
- statistiques sur l'exécution des tests
- interface graphique pour la couverture des tests
- points d'extensions pour des situations spécifiques

Solution très simple et extrêmement efficace

- . création manuelle des tests
- . pas de détection des scripts de test devenus sémantiquement incorrects
- . rejeu total, pas de minimisation des tests à rejouer

1. Pour chaque fichier `Foo.java` créer un fichier `FooTest.java` (dans le même repertoire) qui inclut (au moins) le paquetage `junit.framework.*`
2. Dans `FooTest.java`, pour chaque classe `Foo` de `Foo.java` écrire une classe `FooTest` qui hérite de `TestCase`
3. Dans `FooTest` définir les méthodes suivantes :
 - ▶ le constructeur qui initialise le nom de la suite de tests
 - ▶ `setUp` appelée avant chaque test
 - ▶ `tearDown` appelée après chaque test
 - ▶ une ou plusieurs méthodes dont le nom est prefixé par `test` qui implementent les tests unitaires
 - ▶ `suite` qui appelle les tests unitaires
 - ▶ `main` qui appelle l'exécution de la suite

Dans les méthodes de test unitaire, les méthodes testées sont appelées et leur résultat est testé à l'aide d'**assertions** :

- `assertEquals(a,b)`
teste si a est égal à b (a et b sont soit des valeurs primitives, soit des objets possédant une méthode equals)
- `assertTrue(a)` et `assertFalse(a)`
testent si a est vrai resp. faux, avec a une expression booléenne
- `assertSame(a,b)` et `assertNotSame(a,b)`
testent si a et b réfèrent au même objet ou non.
- `assertNull(a)` et `assertNotNull(a)`
testent si a est null ou non, avec a un objet
- `fail(message)`
si le test doit echouer (levée d'exception)

Exemple : Conversion binaire/entier

```
// File Binaire.java
public class Binaire {
    private String tab;
    public Binaire() {tab = new String(); }
    public Binaire(String b, boolean be) {
        tab = new String(b); if (be) revert(); }

    private void revert() {
        byte[] btab = tab.getBytes();
        for (int i = 0; i < (btab.length >> 1); i++) {
            byte tmp = btab[i]; btab[i] = btab[btab.length - i];
            btab[btab.length - i] = tmp;        }
        tab = new String(btab); }

    public int getInt() {
        int nombre = 0;
        /* little endian */
        for (int i = tab.length()-1; i >= 0; i--) {
            nombre = (nombre << 1) + (tab.charAt(i) - '0'); }
        return nombre; }
}
```

Exemple : Test Conversion binaire/entier (2)

```
// Fichier BinaireTest.java
import junit.framework.*;

public class BinaireTest extends TestCase {
    private Binaire bin; // variable pour les tests

    public BinaireTest(String name) {super(name); }

    protected void setUp() throws Exception {
        bin = new Binaire(); }

    protected void tearDown() throws Exception {
        bin = null; }

    public void testBinaire0() {
        assertEquals(bin.getInt(),0); }

    public void testBinaire1() {
        bin = new Binaire("01",false);
        assertEquals(bin.getInt(),2); }
```


Utilisation plus souple via les tags

- `@Test`, `@Before`, `@After`, etc.

Plus puissant

- `@Test(expected = Exception.class)`
- `@Test(timeout=100)`
- tests paramétrés par des collections de données
- ...

- Ecrire les test en même temps que le code.
- Exécuter ses tests aussi souvent que possible, idéalement après chaque changement de code.
- Ecrire un test pour tout bogue signalé (même s'il est corrigé).
- Ne pas mettre plusieurs assert dans un même test : JUnit s'arrete à la première erreur.
- Attention, les méthodes privées ne peuvent pas être testées !

- oracle
- exécution
- tests de régression et JUnit
- quelques dilemmes du test
- critères de couverture

Si la campagne de tests trouve peu d'erreurs

- choix 1 (optimiste) : le programme est très bon
- choix 2 (pessimiste) : les tests sont mauvais

Si la campagne de tests trouve beaucoup d'erreurs

- choix 1 (optimiste) : la majeure partie des erreurs est découverte
- choix 2 (pessimiste) : le programme est de très mauvaise qualité, encore plus de bugs sont à trouver

solution : qualité du jeu de tests [critères de couverture]

- oracle
- exécution
- tests de régression et JUnit
- quelques dilemmes du test
- critères de couverture

Problèmes de la sélection de tests :

- efficacité du test dépend crucialement de la qualité des CT/DT
- ne pas “râter” un comportement fautif
- MAIS les CT/DT sont coûteux (design, exécution, stockage, etc.)

Deux enjeux :

- DT suffisamment variées pour espérer trouver des erreurs
- maîtriser la taille : éviter les DT redondantes ou non pertinentes

Sujet central du test

Tente de répondre à la question : “qu’est-ce qu’un bon jeu de test ?”

Plusieurs utilisations des critères :

- guide pour choisir les CT/DT les plus pertinents
- évaluer la qualité d’un jeu de test
- donner un critère objectif pour arrêter la phase de test

Quelques qualités attendues d’un critère de test :

- bonne corrélation au pouvoir de détection des fautes
- concis
- automatisable

Sélection des CT/DT pertinents : très difficile

- expériences industrielles de synthèse automatique

Script de test : de facile à difficile, mais toujours très ad hoc

Verdict et oracle : très difficile

- certains cas particuliers s'y prêtent bien
- des oracles partiels automatisés peuvent être utiles

Régression : bien automatisé (ex : JUnit pour Java)

■ Introduction au test logiciel

- ▶ Contexte
- ▶ Définition du test
- ▶ Éléments de classification
- ▶ Aspects pratiques
- ▶ Point de vue industriel
- ▶ Complément : critères boîte noire
- ▶ Complément : critères boîte blanche
- ▶ Complément : tests de régression
- ▶ Discussion

1- Construire la qualité du produit

- lors de la phase de conception / codage
- en partie par les développeurs (tests unitaires)
- but = trouver rapidement le plus de bugs possibles (avant la commercialisation)
 - ▶ test réussi = un test qui trouve un bug

2- Démontrer la qualité du produit à un tiers

- une fois le produit terminé
- idéalement : par une équipe dédiée
- but = convaincre (organismes de certification, hiérarchie, client)
 - ▶ test réussi = un test qui passe sans problème
 - ▶ + tests jugés représentatifs
(systèmes critiques : audit du jeu de tests)

Systèmes critiques

- domaines : aéronautique [DO-178B], énergie, ferroviaire, etc.
- démontrer la qualité : tests exigés pour la certification
- niveau unitaire : critères structurels (boite blanche) +/- exigeants [jusque MCDC]
- attention : couverture calculée sur le code, mais tests justifiés vis à vis des spécifications

Méthodes agiles et test-driven development

- domaines : très large !
- principes : test-driven development (test first, scenarios), automatisation des tests de régression (xUnit)
- construire la qualité : niveau unitaire, par le développeur
- démontrer la qualité : scénarios de test définis par le client

Sécurité

- domaines : OS, navigateurs, hyperviseurs / VM / sandbox, briques de chiffrement, etc.
- utilisation massive du *fuzzing* en test de vulnérabilité
 . test boîte noire, essentiellement random, oracle pass/crash

Utilisation courante

- xUnit et assimilés [tests de régression] [plein d'outils associés, ex : mocks]
- outils de calcul de couverture (ex : Emma, Cobertura)
- analyse dynamique (ex : Valgrind) [oracle automatisé un peu meilleur que pass/crash]

À surveiller

- outils à la quickcheck [génération (simple) de cas de tests]
- spécifications exécutables (jml, CodeContract, e-ACSL) [oracle automatisé + puissant]

Outils avancés de génération de test

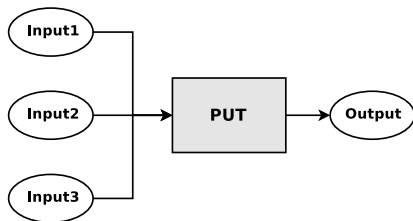
- fuzzing (ex : Radamsa de Google)
(oracle = pass / crash)
- Pex : but = aide au développeur (Parametrized unit testing), bientôt intégré dans Visual Studio
(oracle = assert, CodeContract)
- SAGE : but = trouver des bugs de sécurité, vise les lecteurs media, en production en interne
(oracle = pass / crash)
- CSmith : but = test de compilateurs
(oracle = back2back testing)

■ Introduction au test logiciel

- ▶ Contexte
- ▶ Définition du test
- ▶ Éléments de classification
- ▶ Aspects pratiques
- ▶ Point de vue industriel
- ▶ Complément : critères boîte noire
- ▶ Complément : critères boîte blanche
- ▶ Complément : tests de régression
- ▶ Discussion

- . Couverture des partitions des domaines d'entrée
- . Test aux limites
- . Approche pair-wise pour limiter la combinatoire

Remarque : si on dispose d'un modèle formel, on peut appliquer les critères de couverture boîte blanche au niveau du modèle



`outType function-under-test(inType x, inType y);`

Constat : test exhaustif souvent impraticable

- espace des entrées non borné / paramétré / infini (BD, pointeurs, espace physique, etc.)
- simplement deux entiers 32 bits : 2^{64} possibilités

Principe :

- diviser le domaine des entrées en un nombre fini de classes tel que le programme réagisse pareil (en principe) pour toutes valeurs d'une classe
- conséquence : il ne faut tester qu'une valeur par classe !
- \Rightarrow permet de se ramener à un petit nombre de CTs

Exemple : valeur absolue : `abs : int \mapsto int`

- 2^{32} entrées
- MAIS seulement 3 classes naturelles : $< 0, = 0, > 0$
- on teste avec un DT par classe, exemple : -5, 0, 19

Procédure :

1. Identifier les classes d'équivalence des entrées
 - ▶ Sur la base des conditions sur les entrées/sorties
 - ▶ En prenant des classes d'entrées valides et invalides
2. Définir des CT couvrant chaque classe

Comment faire les partitions ?

Définir un certain nombre de caractéristiques C_i représentatives des entrées du programme

Pour chaque caractéristique C_i , définir des blocs $b_{i,j} \subseteq C_i$

- (couverture) $\cup_j b_{i,j} = C_i$ [< 0, = 0, > 0 ok]
- (séparation) idéalement $b_{i,j'} \cap b_{i,j} = \emptyset$ [< 0, = 0, > 0 ok]

Pourquoi plusieurs caractéristiques ?

- plusieurs variables : `foo(int a, bool b)` :
 $C_1 = \{< 0, = 0, > 0\}$ et $C_2 = \{\top, \perp\}$
- caractéristiques orthogonales : `foo(list<int> l)` :
 $C_1 = \{sorted(l), \neg sorted(l)\}$ et $C_2 = \{size(l) > 10, size(l) \leq 10, \}$

Les partitions obtenues sont le produit cartésien des $b_{i,j}$

- attention à l'explosion !
- possibilité de combiner avec le test combinatoire

Deux grands types de partition

interface-based

- basée uniquement sur les types des données d'entrée
- facile à automatiser ! (cf. exos)

functionality-based

- prend en compte les relations entre variables d'entrées
- plus pertinent
- peu automatisable

Exemple : `searchList` : `list<int> × int ↦ bool`

interface-based : $\{empty(l), \neg empty(l)\} \times \{< 0, = 0, > 0\}$

functionality-based : $\{empty(l), e \in l, \neg empty(l) \wedge e \notin l\}$

À propos des entrées invalides du programme

Conseil 1 : attention à en faire !

Conseil 2 : attention à ne pas trop en faire !!

Pour une fonction de calcul de valeur absolue :

- si le programme a une interface textuelle : légitime de tester les cas où l'entrée n'est pas un entier, il n'y a pas d'entrée, il y a plusieurs entiers, etc.
 - si on a à faire à un module de calcul avec une interface "propre" (un unique argument entier) : on ne teste pas les valeurs invalides sur le moteur de calcul (la phase de compilation nous assure de la correction), mais sur le front-end (GUI, texte)
-

Conseil 3 : Ne pas mélanger les valeurs invalides !

Le test des valeurs limites est une tactique pour améliorer l'efficacité des DT produites par d'autres familles.

- s'intègre très naturellement au test partitionnel

Idée : les erreurs se nichent dans les cas limites, donc tester aussi les valeurs aux limites des domaines ou des classes d'équivalence.

- test partitionnel en plus agressif
- plus de blocs, donc plus de DT donc plus cher

Stratégie de test :

- Tester les bornes des classes d'équivalence, et juste à côté des bornes
- Tester les bornes des entrées et des sorties

Exemples :

- soit N le plus petit/grand entier admissible : tester $N - 1$, N , $N + 1$
- ensemble vide, ensemble à un élément
- fichier vide, fichier de taille maximale, fichier juste trop gros
- string avec une requête sql intégrée
- ...

Exemple : valeur absolue : $\text{abs} : \text{int} \mapsto \text{int}$

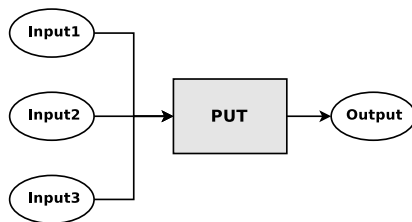
- 2^{32} entrées
- seulement 3 classes naturelles : $< 0, = 0, > 0$
- on ajoute la limite suivante : -2^{31} [question : pourquoi?]

Si on a plusieurs entrées :

dans le cas où il y a trop de $b_{i,j}$, le nombre de partitions $\prod b_{i,j}$ explose et la technique devient inutilisable

Comment faire : l'approche combinatoire peut être appliquée aux $b_{i,j}$

- on ne cherche plus à couvrir tout $\prod b_{i,j}$
- mais par exemple toutes les paires $(b_{i,j}, b_{i',j'})$
- on retrouve l'approche pair-wise [cf juste après]



`outType function-under-test(inType x, inType y);`

Constat : test exhaustif souvent impraticable

- espace des entrées non borné / paramétré / infini (BD, pointeurs, espace physique, etc.)
- simplement deux entiers 32 bits : 2^{64} possibilités

Test combinatoire = test exhaustif sur une sous-partie (bien identifiée) des combinaisons possibles des valeurs d'entrée

Approche pairwise : sélectionner les DT pour couvrir toutes les paires de valeurs

- observation 1 : # paires bcp plus petit que # combinaisons
- observation 2 : un seul test couvre plusieurs paires
- # DT diminue fortement par rapport à test exhaustif

Remarque : on peut étendre à t -uplet, t fixé

- plus de tests, meilleur qualité
- ne semble guère intéressant en pratique

Hypothèse sous-jacente :

- majorité des fautes détectées par des combinaisons de 2 valeurs de variables
 - ▶ semble ok en pratique

Utile quand : beaucoup d'entrées, chacune ayant un domaine restreint

- typiquement : GUI (menus déroulants), interface "ligne de commande" avec de nombreux paramètres, tests de configuration (cf exos)
- très utile aussi en addition au test partitionnel (cf. ci-après)

Test Combinatoire (3)

Exemple : 3 variables booléennes A, B ,C

Nombre de combinaisons de valeurs / tests : $2^3 = 8$

Nombre de paires de valeurs (= nb paires de variables \times 4) : 12

- (A=1,B=1), (A=1,B=0), (A=1,C=1), (A=1,C=0)
- (A=0,B=1), (A=0,B=0), (A=0,C=1), (A=0,C=0)
- (B=1,C=1), (B=1,C=0)
- (B=0,C=1), (B=0,C=0)

IMPORTANT : le DT (A=1,B=1,C=1) couvre 3 paires, mais 1 seule combinaison

- (A=1,B=1), (A=1,C=1), (B=1,C=1)

Ici 6 tests pour tout couvrir :

- (0,0,1), (0,1,0), (1,0,1), (1,1,0) couvrent presque tout, sauf (*,0,0) et (*,1,1)
- on ajoute (1,0,0) et (1,1,1)

Sur de plus gros exemples avec N variables à M valeurs :

- nb combinaisons : M^N
- nb paires de valeurs : $\approx M^2 \times N(N - 1)/2$
- un test couvre au plus $N(N - 1)/2$ paires de valeurs

On peut espérer tout couvrir en M^2 tests plutôt que M^N

- indépendant de N
- plus sensible à la taille des domaines qu'au nombre de variables

Attention : trouver un ensemble de tests de cardinal minimal pour couvrir t-wise est NP-complet

- se contenter de méthodes approchées

Pour aller plus loin 1 : algorithmes usuels [Aditya Mathur, chap. 4]

- covering arrays
- pour $M = 2$: procédure dédiée efficace (polynomiale)

Pour aller plus loin 2 : les DT générées par l'algorithme précédent ne sont pas équilibrées : certaines valeurs sont exercées bien plus que d'autres

- algorithmes à base de carrés latins orthogonaux pour assurer aussi l'équilibrage

Pour aller plus loin 3 : on peut vouloir intégrer certaines contraintes sur les entrées, typiquement exprimer que certaines paires de valeurs sont impossibles

■ Introduction au test logiciel

- ▶ Contexte
- ▶ Définition du test
- ▶ Éléments de classification
- ▶ Aspects pratiques
- ▶ Point de vue industriel
- ▶ Complément : critères boîte noire
- ▶ Complément : critères boîte blanche
- ▶ Complément : tests de régression
- ▶ Discussion

Le graphe de flot de contrôle d'un programme est défini par :

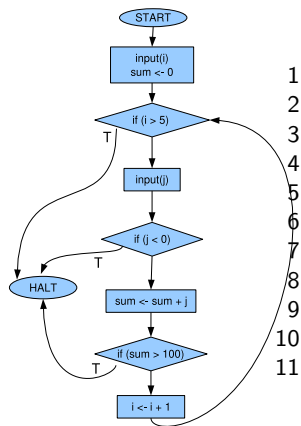
- un noeud pour chaque instruction, plus un noeud final de sortie
- pour chaque instruction du programme, le CFG comporte un arc reliant le noeud de l'instruction au noeud de l'instruction suivante (ou au noeud final si pas de suivant), l'arc pouvant être étiqueté par l'instruction en question

Quelques définitions sur les instructions conditionnelles :

`if (a<3 && b<4) then ... else ...`

- un `if` est une instruction conditionnelle / branchante
- `(a<3 && b<4)` est la condition
- les deux décisions possibles sont (*condition, true*) et (*condition, false*) (chaque transition)
- les conditions simples sont `a<3` et `b<4`

Graphe de flot de contrôle (2)



```
1  START
2  input(i)
3  sum := 0
4  loop : if (i > 5) goto end
5  input(j)
6  if (j < 0) goto end
7  sum := sum + j
8  if (sum > 100) goto end
9  i := i + 1
10 goto loop
11 end : HALT
```

Quelques critères de couverture sur flot de contrôle

- Tous les noeuds (I) : le plus faible.
- Tous les arcs / décisions (D) : test de chaque décision
- Toutes les conditions (C) : peut ne pas couvrir toutes les décisions
- Toutes les conditions/décisions (DC)
- Toutes les combinaisons de conditions (MC) : explosion combinatoire !
- Tous les chemins : le plus fort, impossible à réaliser s'il y a des boucles

Remarque : il existe d'autres critères boîte blanche

- basés sur la couverture du flot de données
- basés sur les mutations syntaxiques du code

Utilisé en avionique (DO-178B). But :

- puissance entre DC et MC
- ET garde un nombre raisonnable de tests

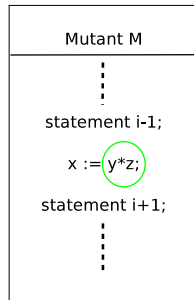
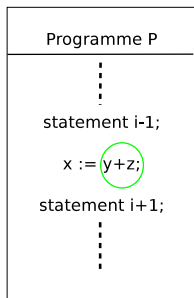
Définition

- critère DC
- ET les tests doivent montrer que chaque condition atomique peut influencer la décision :
par exemple, pour une condition $C = a \wedge b$, les deux DT $a = 1, b = 1$ et $a = 1, b = 0$ prouvent que b seul peut influencer la décision globale C

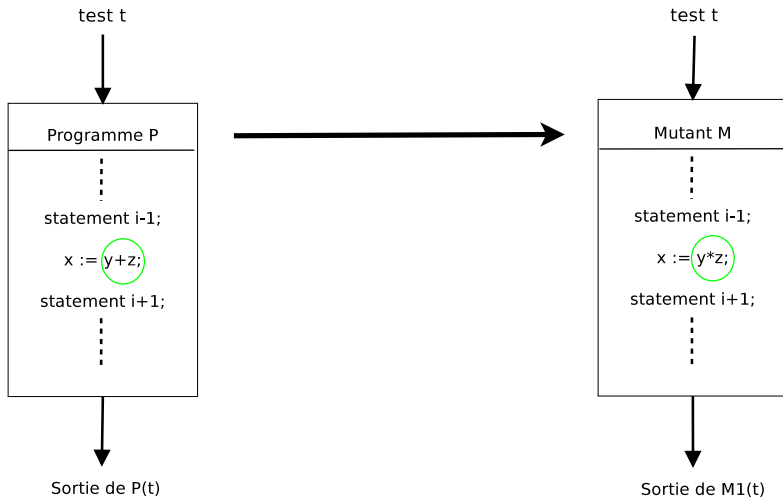
Principe (mutations fortes)

- modifier le programme P en P' en injectant une modification syntaxique correcte (P' compile toujours)
- idéalement, le comportement de P' est différent de celui de P
- sélectionner une DT qui met en évidence cette différence de comportement (= tuer le mutant P')

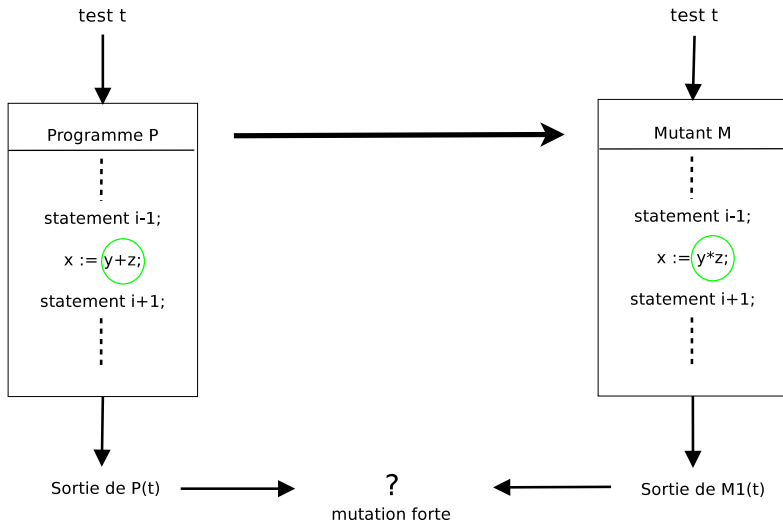
Test par Mutations (fortes)



Test par Mutations (fortes)



Test par Mutations (fortes)



- mutation de P : modification syntaxique de P
- mutant de P : P' obtenu par mutation de P
- TS tue P : $\exists t \in TS$ tq $P(t) \neq P'(t)$
- score de mutation de TS
(# mutants tués)/(# mutants)

Les 3 conditions suivantes doivent être remplies

Accessibilité : $P(t)$ atteint la mutation

Infection : $P(t) \neq P(t')$ juste après la mutation

Propagation : $P(t) \neq P(t')$ à la fin du programme

Pour l'instruction suivante :

```
if a > 8 then x := y+1
```

on peut considérer les mutants suivants :

- if a < 8 then x := y+1
- if a ≥ 8 then x := y+1
- if a > 8 then x := y-1
- if a > 8 then x := y

Pour un programme donné, on considèrera un très grande nombre de mutants

Opérateurs classiques de mutations [Offutt-Ammann]

- **bomb** : ajouter `assert(false)` après une instruction
- modifier une expression arithmétique e en $|e|$ (ABS)
- modifier un opérateur relationnel arith par un autre (ROR)
- modifier un opérateur arith par un autre (AOR)
- modifier un opérateur booléen par un autre (COR)
- modifier une expression bool/arith en ajoutant $-$ ou \neg (UOI)
- modifier un nom de variable par un autre
- modifier un nom de variable par une constante du bon type
- modifier une constante par une autre constante du bon type
- ...

Critère le plus fort, peut émuler les autres !

Très bonne corrélation expérimentale avec le “pouvoir de détection de bugs” d’une suite de tests

Créer les mutants : très très grand nombre, souvent $|M| \approx |P|$

Compiler les mutants : $|M|$ compilations d'un prog. de taille $|P|$

- une compilation par mutant
- (rappel) : couv. D : 1 compilation

Calculer le score de mutation : $(|M| + 1) \times |T|$ exécutions

- exec. chaque test contre chaque mutant (+ prog. initial)
- (rappel) : couv. D : $|T|$ exécutions

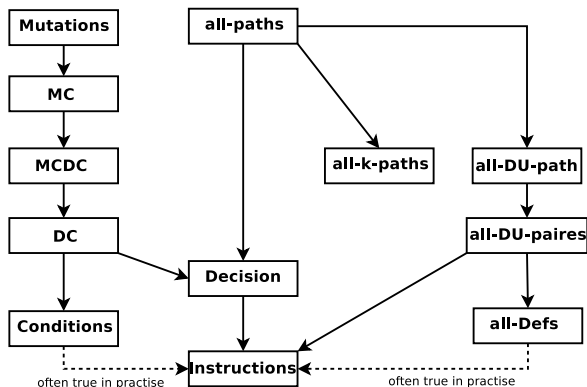
Notion de hiérarchie entre ces différents critères de couverture

Le critère CT1 est plus fort que le critère CT2 (CT1 *subsumes* CT2, noté $CT1 \succeq CT2$) si pour tout programme P et toute suite de tests TS pour P, si TS couvre CT1 (sur P) alors TS couvre CT2 (sur P).

Exercice : supposons que TS2 couvre CT2 et trouve un bug sur P, et TS1 couvre un critère CT1 tq $CT1 \succeq CT2$.

Question : TS1 trouve-t-il forcément le même bug que TS2 ?

Critères boîte blanche (6) - Hiérarchie des critères



Ne sont pas reliés à la qualité finale du logiciel (MTBF, PDF, ...)

- sauf test statistique

Ne sont pas non plus vraiment reliés au # bugs /kloc

- exception : mcdc et contrôle-commande
- exception : mutations

■ Introduction au test logiciel

- ▶ Contexte
- ▶ Définition du test
- ▶ Éléments de classification
- ▶ Aspects pratiques
- ▶ Point de vue industriel
- ▶ Complément : critères boîte noire
- ▶ Complément : critères boîte blanche
- ▶ Complément test de régression
- ▶ Discussion

Compromis entre tout rejouer (sûr mais trop cher) et ne pas rejouer assez.

- certains tests ne passent pas par les modifications : les ignorer

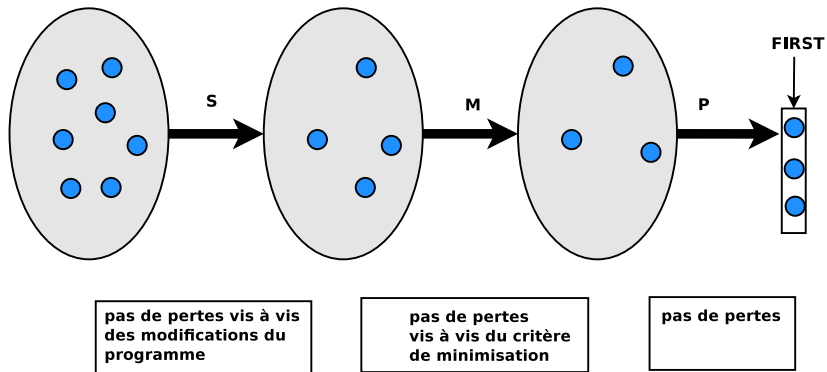
Problème additionnel : temps total pour le rejeu limité

- on arrête après N tests
- avec cette limite, le rejeu total est risqué
- faire tests pertinents d'abord

Trois phases distinctes dans la sélection :

- **S**électionner les tests pertinents (aucune perte)
- **M**inimiser les tests pertinents (perte possible)
- **P**rioritiser les tests restants (aucune perte)

Tests de régression : problème SMP (2)



■ Introduction au test logiciel

- ▶ Contexte
- ▶ Définition du test
- ▶ Éléments de classification
- ▶ Aspects pratiques
- ▶ Point de vue industriel
- ▶ Complément : critères boîte noire
- ▶ Complément : critères boîte blanche
- ▶ Complément test de régression
- ▶ Discussion

Test = activité difficile et coûteuses

Difficile

- trouver les défauts = pas naturel (surtout pour le programmeur)
- qualité du test dépend de la pertinence des cas de tests

Coûteux : entre 30 % et 50 % du développement

Besoin de l'automatiser/assister au maximum

Gains attendus d'une meilleur architecture de tests

- amélioration de la qualité du logiciel
- et/ou réduction des coûts (développement - maintenance) et du time-to-market

Si la campagne de tests trouve peu d'erreurs

- choix 1 (optimiste) : le programme est très bon
- choix 2 (pessimiste) : les tests sont mauvais

Si la campagne de tests trouve beaucoup d'erreurs

- choix 1 (optimiste) : la majeure partie des erreurs est découverte
- choix 2 (pessimiste) : le programme est de très mauvaise qualité, encore plus de bugs sont à trouver

Aller doucement, du plus simple au plus compliqué

1. test-driven development : test-first, xUnit, intégration continue, objectifs de couverture
2. automatisation simple de la génération de tests (random, fuzzing, etc.)
3. langage d'annotation
4. automatisation plus poussée
 - ▶ smart fuzzing, parametrized unit testing
 - ▶ model-based testing

Testing can only reveal the presence of errors but never their absence

- E. W. Dijkstra (Notes on Structured Programming, 1972)

Oui, mais ...

- Correspond au niveau de fiabilité exigé du reste du système
- Correspond aux besoins réels de beaucoup d'industriels
- Peut attaquer des programmes + complexes

Testing can only reveal the presence of errors but never their absence

- E. W. Dijkstra (Notes on Structured Programming, 1972)

Oui, mais ...

- Correspond au niveau de fiabilité exigé du reste du système
- Correspond aux besoins réels de beaucoup d'industriels
- Peut attaquer des programmes + complexes

Testing can only reveal the presence of errors but never their absence

- E. W. Dijkstra (Notes on Structured Programming, 1972)

Oui, mais ...

- Correspond au niveau de fiabilité exigé du reste du système
- Correspond aux besoins réels de beaucoup d'industriels
- Peut attaquer des programmes + complexes

Testing can only reveal the presence of errors but never their absence

- E. W. Dijkstra (Notes on Structured Programming, 1972)

Oui, mais ...

- Correspond au niveau de fiabilité exigé du reste du système
- Correspond aux besoins réels de beaucoup d'industriels
- Peut attaquer des programmes + complexes

- déjà utilisé : ne modifie ni les process ni les équipes
- retour sur investissement proportionnel à l'effort
- simple : pas d'annotations complexes, de faux négatifs, ...
- robuste aux bugs de l'analyseur / hypothèses d'utilisation
- trouve des erreurs non spécifiées

Testing can only reveal the presence of errors but never their absence

- E. W. Dijkstra (Notes on Structured Programming, 1972)

Oui, mais ...

- Correspond au niveau de fiabilité exigé du reste du système
- Correspond aux besoins réels de beaucoup d'industriels
- Peut attaquer des programmes + complexes

Testing can only reveal the presence of errors but never their absence

- E. W. Dijkstra (Notes on Structured Programming, 1972)

Oui, mais ...

- Correspond au niveau de fiabilité exigé du reste du système
- Correspond aux besoins réels de beaucoup d'industriels
- Peut attaquer des programmes + complexes

Offre des solutions (partielles) pour

- bibliothèques sous forme binaire (COTS)
- codes mélangeant différents langages (assembleur, SQL, ...)
- code incomplet

Beware of bugs in the above code ; I have only proved it correct, not tried it.

- Donald Knuth (1977)

It has been an exciting twenty years, which has seen the research focus evolve [...] from a dream of automatic program verification to a reality of computer-aided [design] debugging.

- Thomas A. Henzinger (2001)

Opposition historique forte

Complémentaire au niveau fonctionnel

- propriété prouvée = pas besoin de tests
- propriété non prouvée = peut être fausse ? (test !)
- certaines classes de propriétés sont pour le moment non modélisables [perfs, ergonomie, etc.]
- facilité de mise en oeuvre : $AS \text{ unsound} \leq \text{test} \leq AS \text{ sound}$

De plus en plus similaire en terme de technologie

- preuve d'invariance vs preuve d'accessibilité
- mêmes outils : logique, sémantique, analyse de programme, etc.
- mais approximations différentes (sur- vs sous-), importance de la synthèse
- de plus en plus de techniques "hybrides" (bounded model checking, context-bounded analysis, etc.)
- remarque : langage de spécification utile pour vérification et test [next big step in industry ?]