

# Automatisation du Test Logiciel

Sébastien Bardin

CEA-LIST, Laboratoire de Sûreté Logicielle

`sebastien.bardin@cea.fr`

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

- Contexte
- Définition du test
- Aspects pratiques
- Le processus du test
- 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)

## 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

- Contexte
- Définition du test
- Aspects pratiques
- Le processus du test
- 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)

## Process (1 seul test)

- 1 choisir un cas de tests (CT) = scénario à exécuter
- 2 estimer le résultat attendu du CT (Oracle)
- 3 déterminer (1) une donnée de test (DT) suivant le CT, et (2) son oracle concret (concrétisation)
- 4 exécuter le programme sur la DT (script de test)
- 5 comparer le résultat obtenu au résultat attendu (verdict : pass/fail)

---

**Script de Test** : code / script qui lance le programme à tester sur le DT choisi, observe les résultats, calcule le verdict

**Suite / Jeu de tests** : ensemble de cas de tests

Spécification : tri de tableaux d'entiers + enlève la redondance

Interface : `int[] my-sort (int[] vec)`

---

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]

# Exemple (2)

## Script de test

```
1 void testSuite () {
2
3     int [] td1 = [5,3,15] ; /* prepare data */
4     int [] oracle1 = [3,5,15] ; /* prepare oracle */
5     int [] res1 = my-sort(td1); /* run CT and */
6                                     /* observe result */
7     if (array-compare(res1,oracle1)) /* assess validity */
8     then print('test1 ok')
9     else {print('test1 erreur')};
10
11
12     int [] td2 = [] ; /* prepare data */
13     int [] oracle2 = [] ; /* prepare oracle */
14     int [] res2 = my-sort(td2);
15     if (array-compare(res2,oracle2)) /* assess validity */
16     then print('test2 ok')
17     else {print('test2 erreur')};
18
19
20     ... /* same for TD3 */
21
22
23 }
```

# 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 test)
- notamment
  - ▶ tests nominaux : cas de fonctionnement les plus fréquents
  - ▶ tests de robustesse : cas limites / délicats

# Deux aspects différents du test

## 1- Contribuer à assurer 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- Validation : Démontrer la qualité à un tiers

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

## Critère de tests

- boîte blanche / boîte noire / probabiliste

## Phase du processus de test

- test unitaire, d'intégration, système, acceptation, regression

**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

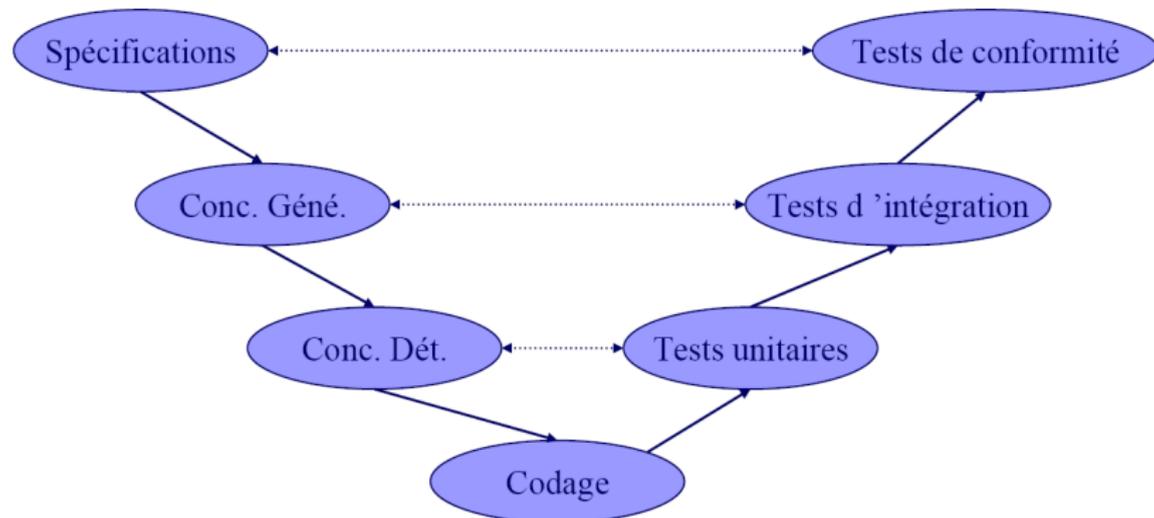
- on teste aussi toutes les caractéristiques émergentes  
sécurité, performances, etc.

**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

## Phase du processus de test (2)



**Boîte Noire** : à partir de spécifications

- dossier de conception
- interfaces des fonctions / modules
- modèle formel ou semi-formel

**Boîte Blanche** : à partir du code

**Probabiliste** : domaines des entrées + arguments statistiques

- Ne nécessite pas de connaître la structure interne du système
- Basé sur la spécification de l'interface du système et de ses fonctionnalités : taille raisonnable
- Permet d'assurer la conformance spéc - code, mais aveugle aux défauts fins de programmation
- Pas trop de problème d'oracle pour le CT, mais problème de la concrétisation
- Approprié pour le test du système mais également pour le test unitaire

- La structure interne du système doit être accessible
- Se base sur le code : très précis, mais plus “gros” que les spécifications
- Conséquences : DT potentiellement plus fines, mais très nombreuses
- Pas de problème de concrétisation, mais problème de l’oracle
- Sensible aux défauts fins de programmation, mais aveugle aux fonctionnalités absentes

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

- Contexte
- Définition du test
- Aspects pratiques
- Le processus du test
- Discussion

La définition de l'oracle est un problème très difficile

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

## Quelques cas pratiques d'oracles parfaits automatisables

- comparer à une référence : logiciel existant, tables de résultats
- résultat simple à vérifier (ex : solution d'une équation)
- disponibilité d'un logiciel similaire : test dos à dos

---

## Des oracles partiels mais automatisés peuvent être utiles

- oracle le plus basique : le programme ne plante pas
- instrumentation du code (assert)
- plus évolué : programmation avec contrats (Eiffel, Jml pour Java)

## Composition du script de test

- **préambule** : amène le programme dans la configuration voulue pour le test (ex : initialisation de BD, suite d'émissions / réceptions de messages, etc.)
- **corps** : appel des "stimuli" testés (ex : fonctions et DT)
- **identification** : opérations d'observations pour faciliter / permettre le travail de l'oracle (ex : log des actions, valeurs de variables globales, etc.)
- **postambule** : retour vers un état initial pour enchaîner les tests

## Le script doit souvent inclure de la glue avec le reste du code

- **bouchon** : simule les fonctions appelées mais pas encore écrites

## Quelques exemples de problèmes

---

Code manquant (test incrémental)

Exécution d'un test très coûteuse en temps

Hardware réel non disponible, ou peu disponible

Présence d'un environnement (réseau, Base de Données, machine, etc.)

- comment le prendre en compte ? (émulation ?)

Réinitialisation possible du système ?

- si non, l'ordre des tests est très important

Forme du script et moyens d'action sur le programme ?

- 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 ?)

Message :

- code “desktop” sans environnement : facile
- code embarqué temps réel peut poser de sérieux problèmes, solutions ad hoc

**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
- détecter les tests dont les scripts ne sont plus (syntaxiquement) 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 ( tests de MyClass dans MyClassTest)

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

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

## Sélection des Tests (2)

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

## Sélection des Tests (2)

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

## Sélection des Tests (2)

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

## Sélection des Tests (2)

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 (parmi 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 de xxxx : 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)

## 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

## Boite noire

- couverture des partitions des domaines d'entrée
- couverture des scénarios d'utilisation

## Boite blanche

- couverture du graphe de flot de contrôle (instructions, branches)
- couverture logique (décision , condition, MCDC)
- couverture des dépendences de données
- mutations

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

Mais toujours mieux que rien ...

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)

- Contexte
- Définition du test
- Aspects pratiques
- Le processus du test
- Discussion

Problème spécifique au test unitaire : code incomplet

- modules appelés pas forcément codés (bouchons, stubs)
- souvent, pas de module appelant (lanceurs, drivers)

Notion de stratégie de test unitaire :

- toujours incrémentale
- bottom-up ou top-down
- et coder les lanceurs / bouchons au besoin

## Avantages / inconvénients du bottom-up

- (++) lanceurs souvent + simples à faire que bouchons
- (++) DT et observations souvent + faciles
- (-) on assemble et teste les parties hautes du programme seulement à la fin (on teste plus les feuilles que les modules de haut niveau)

## Avantages / inconvénients du top-down

- (++) les modules de plus haut niveau sont le plus testés (servent de drivers)
- (++) CT plus simples à trouver
- (-) bouchons difficiles à faire

# Quels critères de qualité ?

critère	TB	B	M	F
use-case	100%	100% (C) 80% (NC)	100% (C) 80% (NC)	80%
interfaces	100%	100%	80%	50%
autres BN	>90%	80%-90%	60%-80%	>50%
couv. code	de 100% I à 100% MC/DC	> 90% I	>80% I	-

TB : très bien, B : bien, M : moyen, F : faible

(C) : code critique, (NC) : non critique

I : instructions

Test planifié à l'avance : définir

- méthodes (?), objectifs, critères de qualité visés, critères d'arrêt
- format des tests (DT + oracle), outils de maintenance et rejeu, stockage
- qui fait les tests ? qui corrige les fautes trouvées ?

---

Pas de tests jetables : tracer les DT aux exigences, m<sup>à</sup>j des tests, rejeu

Prévoir budget suffisant : penser qu'il y aura des erreurs!!!

L'activité de test démarre dès le début du cycle en V

Penser aux tests autres que correction fonctionnelle

Penser que corriger un bug introduit souvent autre(s) bug(s)

Suivi de l'activité de test : (monitoring)

Tout au long du projet, collecter des données sur l'activité de test :

- pour le suivi de ce projet en particulier (ex : #bugs trouvés / semaine)
- pour aider à mieux calibrer l'activité de test dans l'entreprise
  - ▶ stat. efforts / méthodes de test → #bugs rapportés par le client
  - ▶ profil des fautes trouvées (ex : besoins, spec/design, code, régression)
  - ▶ stat. sur le profil de découverte des fautes
  - ▶ ...

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

Nb erreurs trouvées ne croît pas linéairement avec le temps

Typiquement 2 phases

- alternations de : #erreurs ↗ puis palliers (saturation)
- à un moment : stagnation du #erreurs trouvées

---

## Conséquences

- ne pas arrêter les tests en période d'augmentation forte du #erreurs trouvées
- ne pas arrêter les tests dès qu'on arrive au premier pallier

# Critères d'arrêt en pratique

mauvaise idée 1 : tester jusqu'à la fin du budget

- pourquoi : objectif implicite = ne rien faire
- 

mauvaise idée 2 : tester jusqu'à ce que tous les tests passent

- pourquoi : objectif implicite = ne pas trouver de bugs
- 

(fausse) bonne idée : quand #fautes découvertes stagne

- pourquoi : attention aux effets de saturation
- 

idée ok : expliciter un critère de qualité à atteindre

- risque = chercher à atteindre critère plutôt que trouver bugs  
(or, peu de corrélation critère / découverte de bugs)
- distinguer objectifs et moyens

# Critères d'arrêt en pratique (2)

Proposition (réaliste ?) [G. J. Myers]

- constat 1 : le but est de trouver des bugs
- constat 2 : il y a toujours des bugs
- donc : exprimer le critère en fonction du # bugs trouvés

---

Par exemple : programme de 10 000 instructions. On estime 50 erreurs / kLoc soit 500 erreurs. On suppose 200 erreurs de codage et 300 erreurs de conception. On veut détecter 98% des erreurs de code et 95% des erreurs de conception. On doit donc trouver : 196 erreurs de code, 285 erreurs de conception.

Critères d'arrêt choisis

- test unitaire : 130 erreurs trouvées et corrigées (65%)
- test intégration : 240 erreurs (30% de 200 + 60% de 300), **ou 4 mois se sont écoulés**
- test système : 111 erreurs, **ou 3 mois**

## Problèmes pratiques pour l'approche de G. J. Myers

- estimer #bugs à trouver ?
- on trouve vite bcp de bugs : attention, #bugs sous-estimé ??
- on ne trouve pas assez de bugs : bon code ou mauvais tests ? (auditer qualité du test)
- attention à la saturation

(voir [Art of Software Testing] pour plus de détails)

## 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]

- Contexte
- Définition du test
- Aspects pratiques
- Le processus du test
- Discussion

Certains postes sont dédiés à la V & V, et donc surtout au test :

- équipes de test dédiées sur de grands projets
- auditeur pour la certification

Bagage essentiel de l'“honnête programmeur” du 21e siècle

- souvent le programmeur doit réaliser les tests unitaires et mettre en place les tests de régression
- tests unitaires et tests de régressions reconnus comme bonne pratique (cf projets libres)
- test au coeur de certains process récents (*test-driven development*)

## C1 : Caractéristique à tester

- correction fonctionnelle, performances, etc.

## C2 : Phase du process de test

- tests unitaires / d'intégration / système / ...

## C3 : Famille de sélection des cas de tests

- boîte blanche / boîte noire / probabiliste

## C4 : Critère de qualité retenu

- couverture fonctionnelle / structurelle / probabiliste

Terminologie pas bien fixée

Souvent, dans le langage courant :

- “test structurel” : unitaire, correction, sélection BB, qualité = couverture structurelle
- “test fonctionnel” : intégration / système, correction, sélection BN, qualité = couverture fonctionnelle

MAIS rien n'empêche d'avoir d'autres combinaisons (cf Amman - Offutt 2008)

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

## Matûrit  du process de test (Beizer)

- 0 : pas de diff rence entre test et d bogage
  - ▶ activit  “artisanale”, ni identifi e, ni planifi e
- 1 : le test sert   montrer que le logiciel fonctionne
  - ▶ impossible
  - ▶ pas d’objectif clair et quantifiable
- 2 : le test sert   trouver des bugs
  - ▶ opposition avec les d veloppeurs
  - ▶ position schizophr nique pour le d veloppeur - testeur
- 3 : le test sert   diminuer le risque d’utilisation du logiciel
  - ▶ on accepte le fait qu’il y a toujours des bugs
  - ▶ il faut essayer de les pr venir au maximum (designer, d veloppeur) et de les corriger (testeur)
- 4 : le test est une “discipline mentale” qui permet d’am liorer la qualit  du logiciel
  - ▶ le test prend une place centrale dans le process de d veloppement
  - ▶ l’ing nieur test devient le responsable technique du projet

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)