UNIVERSITÉ PARIS SUD

Comprendre le monde, construire l'avenir®

Test de logiciels

Différence entre Validation et Vérification

- Validation :
  - Le système répondra-t-il aux besoins du client ?
  - Offrira-t-il le niveau attendu de performances ?
  - Sera-t-il pratique à utiliser ?

Est-ce le bon système qu'on construit ?

- Vérification: Le système développé satisfait-il les spécifications ?

Ce système est-il correctement construit ?

Est-il « correct » ?

Pourquoi vérifier et valider?

Pour assurer la qualité

- Validité : réponse aux besoins des utilisateurs
- Facilité d'utilisation : prise en main et robustesse
- Performance : temps de réponse, débit, fluidité...
- Fiabilité : tolérance aux pannes
- Sécurité : intégrité des données et protection des accès
- Maintenabilité : facilité à corriger ou transformer le logiciel
- Portabilité : changement d'environnement matériel ou logiciel

Evolution du test

Aujourd'hui, le test de logiciels :

- est la méthode la plus utilisée pour assurer la qualité des logiciels
- fait l'objet d'une pratique trop souvent artisanale

Demain, le test de logiciels devrait être :

- une activité rigoureuse
- fondé sur des modèles et des théories
- de plus en plus automatique

Le coût du test

- coût du test ? *35 à 50 % de l'effort global ?*
- Tous les "gros" logiciels ont des erreurs résiduelles…
- Coût d'une erreur ?
  - Coût de correction et de diffusion de "mises à jour"…
  - Atteinte à l'image de marque ; perte de confiance
  - Retards pour la mise sur le marché
  - Dédommagements
    indemnités pour les victimes (en plus du coût « moral »)
    valeurs de remplacement (100M € pour certains satellites !)

pourtant on ne peut pas tester indéfiniment, ni tout prouver !

Qu’est ce que tester ?

- Ce que le test n’est pas !
  - prouver le programme : ce n’est pas le but !
  - le mettre au point : le test se fait après !
  - montrer que le programme ne “bombe” pas : résultat correct ?
  - montrer que ça marche : ce n’est pas le développeur qui teste !
  - En cas d’erreur: trouver la cause, la corriger …
- Le test c’est:
  - lire, faire lire, soumettre le texte du programme à un outil d’analyse = TEST “STATIQUE” OU ANALYSE
  - exécuter le programme sur un ensemble de données “significatives” = TEST DYNAMIQUE

… dans le but d’y trouver des erreurs !

Tester c’est :

- Selon la norme IEEE (Standard Glossary of Software Engineering Terminology)

  « Le test est l’exécution ou l’évaluation d’un système ou d’un composant par des moyens automatiques ou manuels, pour vérifier qu’il répond à ses spécifications ou identifier les différences entre les résultats attendus et les résultats obtenus. »

  ⇨ Validation dynamique (exécution du système)

  ⇨ Comparaison entre système et spécification

Test statique / Test dynamique

- **statique**: sans exécuter le programme sur des données réelles
  - analyser le produit à partir de ses composants
  - travailler symboliquement sur une classe d'exécution
    = représenter un nombre potentiellement infini d'exécutions
- **dynamique**: exécuter le programme (ou un composant)
  - expérimenter avec le comportement
  - une suite d'expérimentation "isolées", significatives
    = une suite finie d'exécutions
- Il est parfois plus facile d'analyser que de tester:
  allocation mémoire, gestion de fichiers, parallélisme…

☹ *vérification du résultat, reproductibilité des défaillances, ...*
☹ *cela demande souvent un outillage (environnement) coûteux*

Bug?

- Erreur (programmation, conception) :
  - Comportement du programmeur ou du concepteur conduisant à un défaut
- Défaut (interne) :
  - Elément ou absence d'élément dans le logiciel entraînant une anomalie
- Anomalie (fonctionnement):
  - Différence entre le comportement attendu et le comportement observé

Chaîne causale : erreur -> défaut -> anomalie

Un bon test est un test qui permet de découvrir un défaut en déclenchant une anomalie

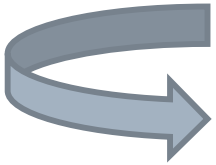
Classification sommaire des anomalies

- Mineures : pas d'incidence fâcheuse sur le fonctionnement du système, ni sur son environnement. La correction des défauts correspondants ne nécessite pas une reprise importante du processus de développement.
- Bloquantes (partiellement) : incidence sensible sur le fonctionnement du système ou sa disponibilité, modéré sur son environnement. La correction peut nécessiter une reprise importante du processus de développement.
- Majeures (ou anti-sécuritaires) : incidence destructive sur le système et son environnement (cause de décès, de blessés) ou entraînant des coûts extrêmement importants.

Limites du test

- Explosion combinatoire : Nombre d'exécutions possibles d'un programme potentiellement infini

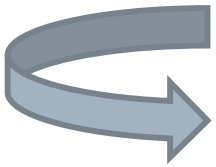
Mais test = processus fini



Nécessité d'approcher l'infini (l'extrêmement grand) par le fini (heuristique)

Quelques définitions

- Objectif de test : comportement du système à tester
- Données de test : données à fournir en entrée au système de manière à déclencher un objectif de test
- Résultats d'un test : conséquences ou sorties de l'exécution d'un test (affichage à l'écran, modification des variables, envoi de messages...)

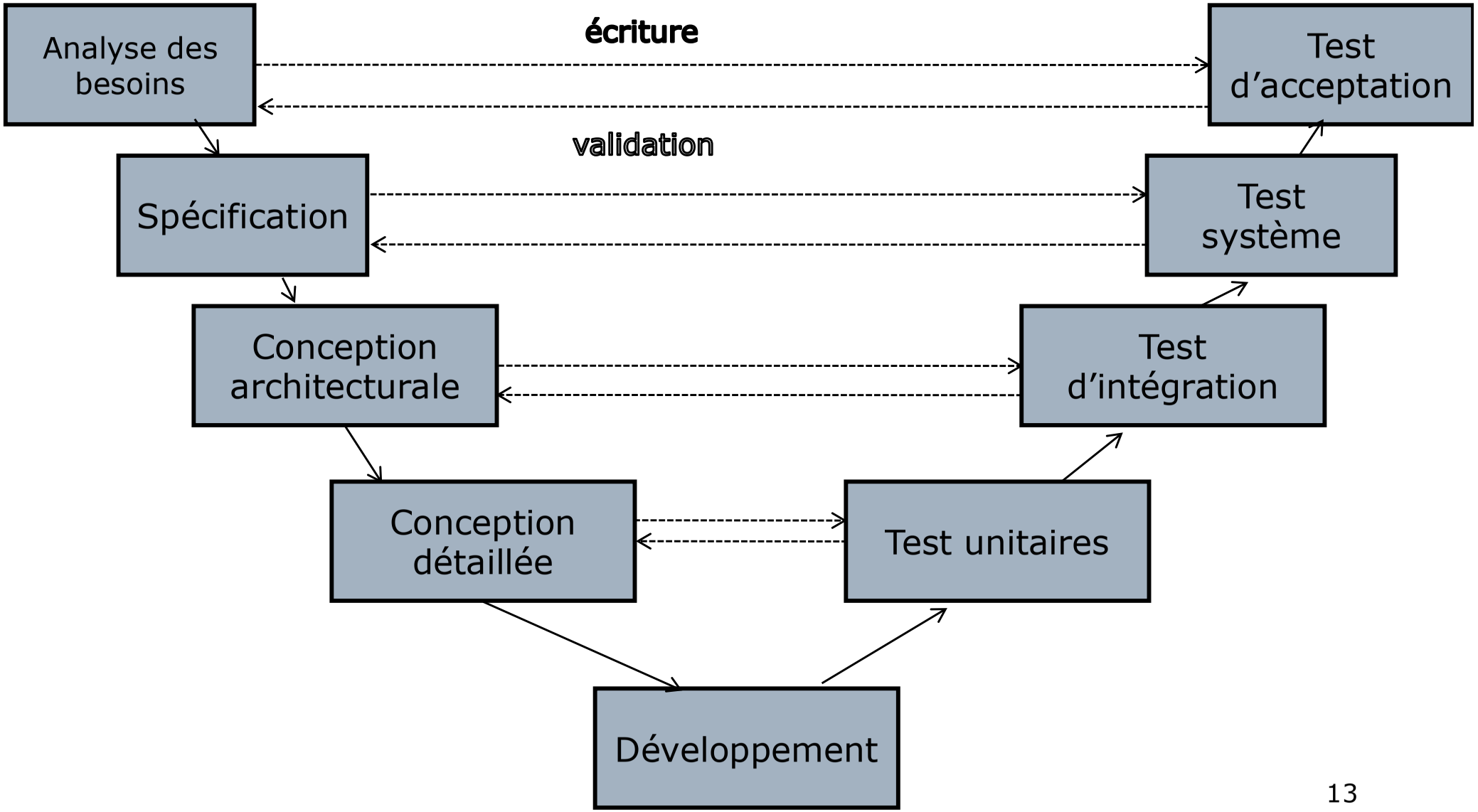


Cas de test : données d'entrée et résultats attendus associés à un objectif de test

---

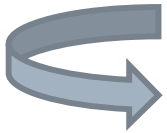
Glossary of Testing Terms, ISTQB (International Software Testing Qualifications Board)

Cycle de vie du logiciel



Test Unitaire

- Test des unités de programme de façon isolée, indépendamment les unes des autres, c'est-à-dire sans appel à une fonction d'un autre module, à une base de données...



méthodes, classes, modules, composants

- *Ex : GPS*

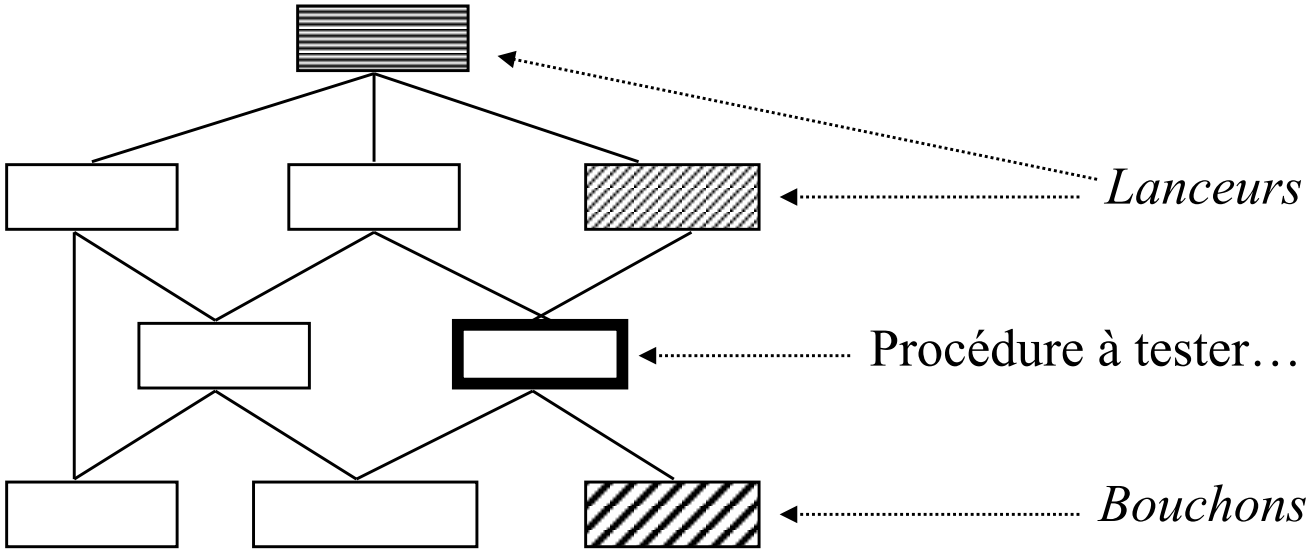
  Algorithme de calcul d'itinéraire sur des exemples de graphes construits à la main

Possibilité d'utiliser des outils de mesure de la qualité des tests en terme de couverture de code

Méthode

Test incrémental : méthode, classe, package, composant

Lanceurs et Bouchons



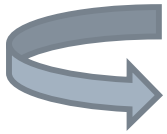
« Comment tester un module qui utilise des modules inachevés ? »

- du code à développer et qui sera jeté !
- ce code est-il correct ? => il faut aussi le tester
- besoins matériels/logiciels similaires: bancs de test…

Problème similaire avec des ensembles de classes inter-dépendantes !

Test d’intégration

- Test de la composition des modules via leur interface



communications entre modules, appels de procédures...

- Tests générés à partir de l’architecture du système et de la spécification des interfaces des composants

- *Ex : GPS*
  - Lecture des données depuis la base de données
  - Communications avec l'IHM

Test système

- Test de la conformité du produit fini par rapport au cahier des charges, effectué en boîte noire au travers de son interface
- Tests ciblés sur les exigences décrites dans la spécification
- Construction possible des tests à partir des cas et des scénarios d'utilisation

- *Ex : GPS*
  - Utilisation du logiciel sur des scénarios réalistes et complets

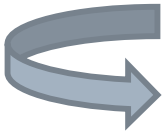
Intérêt de la planification des tests système en phase d'analyse

- Validation du produit fini par rapport aux spécifications initiales
- Permet de s'assurer qu'on n'a pas dévié des spécifications

Test de non régression

- But : Assurer que les corrections et les évolutions du code n'ont pas introduit de nouveaux défauts

- Méthode : À chaque ajout ou modification de fonctionnalité, rejouer les tests pour cette fonctionnalité, puis pour celles qui en dépendent, puis les tests des niveaux supérieurs



Lourd mais indispensable

Automatisable en grande partie

Problématique du test dynamique

- Impossible d'exécuter un programme sur toutes ses entrées possibles
  - Nécessité de sélectionner les tests possibles
- Objectif : détecter un maximum de défauts avec un minimum de tests
- Ensemble de tests idéal
  - Ensemble qu'on peut exécuter en un temps fini et raisonnable
  - Ensemble représentatif qui est susceptible de trouver un grand nombre de fautes

Méthodes de sélection de tests

- Couverture des exigences : toute exigence fonctionnelle et non fonctionnelle est couverte par au moins un test

  Critère de mesure de la satisfaction du cahier des charges
- Couverture des données : réduction de la combinatoire des valeurs possibles des données
  - Grands domaines de valeurs pour chaque donnée : choix de données représentatives d'un comportement standard (*analyse partitionnelle*), d'un comportement limite (*test aux limites*)
  - Petits domaines de valeurs pour grand nombre de données : couverture des valeurs individuelles (*all-singles*), des couples de valeurs (*test pairwise*)

Méthodes de sélection de tests

- Couverture structurelle du modèle : critères structurels permettant de couvrir un modèle du comportement du système, un modèle du code du programme (*test boîte blanche*)...
  - Couverture des nœuds, des transitions, des branchements, des boucles, des chemins…

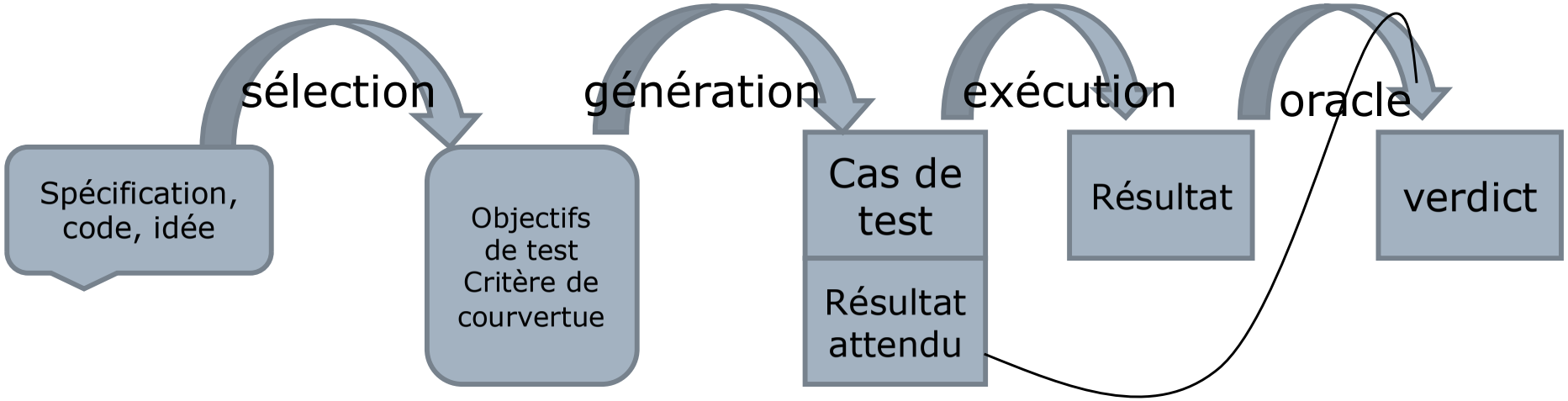
- Génération guidée par les fautes : tests conçus pour cibler des modèles de fautes connues
  - Tests générés pour détecter les effets de petites modifications syntaxiques dans le code (*test de mutation*)
  - Tests générés pour détecter des failles de sécurité connues (injection de code, buffer overflow...)

Méthodes de sélection de tests

- Génération statistique :
  - Génération aléatoire des valeurs des données (suivant une loi de distribution ou suivant un profil d'utilisation)
  - Génération aléatoire d'un chemin dans un modèle d'utilisation du système, ou dans un modèle du comportement du système
- Spécification explicite des tests : compléter la génération automatique avec des objectifs de test construits à la main, mais dont les tests peuvent être générés automatiquement (expression régulière représentant une suite d'actions par exemple)

Processus de test

1. Choisir les comportements à tester (objectifs de test)
2. Choisir des données de test permettant de déclencher ces comportements + décrire le résultat attendu pour ces données
3. Exécuter les cas de test sur le système + collecter les résultats
4. Comparer les résultats obtenus aux résultats attendus pour établir un verdict



Exécution d'un test

Scénario de test :

- Préambule : Suite d'actions amenant le programme dans l'état nécessaire pour exécuter le cas de test
- Corps : Exécution des fonctions du cas de test
- Identification (facultatif) : Opérations d'observation rendant l'oracle possible
- Postambule : Suite d'actions permettant de revenir à un état initial

préambule corps identification postambule

Décision du succès ou de l'échec d'un test

- Comparaison entre le résultat obtenu et le résultat attendue

  Problème relativement complexe : problème de l'oracle

  ➡
  - Types de données sans prédicat d'égalité
  - Système non déterministe : sortie possible mais pas celle attendue
  - Heuristique : approximation du résultat optimal attendu

- Solution : description du résultat attendu, selon les cas :
  - La valeur attendue
  - Énumération des valeurs possibles
  - Ensemble de conditions

Mise en garde

- Les tests doivent être exécutés dans des conditions et sur des données connues et contrôlées
- Le résultat d'un test doit être observable
- Il doit exister un moyen sûr de comparer le résultat observé au résultat attendu afin de déterminer le succès ou l'échec du test
- Les cas de test ne doivent pas être redondants : plusieurs cas ne doivent pas cibler la même faute
- Un cas de test correspond à un comportement cible unique : pas de choix pendant l'exécution du test

Exemple sur le tri d'une liste

Objectif de test	Données de test	Résultat attendu	Résultat du test
Liste vide	[]	[]	
Liste à 1 élément	[1]	[1]	
Liste >= 2 éléments déjà triés	[1;2;9;13]	[1;2;9;13]	
Liste >= 2 éléments non triés	[7;10;3;8;5]	[3;5;8;7;10]	

Exemple sur le tri d'une liste

Objectif de test	Données de test	Résultat attendu	Résultat du test
Liste vide	[]	[]	[…]
Liste à 1 élément	[1]	[1]	[…]
Liste >= 2 éléments déjà triés	[1;2;9;13]	[1;2;9;13]	[…]
Liste >= 2 éléments non triés	[7;10;3;8;5]	[3;5;8;7;10]	[…]

Problème de l’oracle

- *Ex : Trouver le minimum d'une liste d'entiers*

Entrée : [4; 2; 3; 6] Sortie attendue : 2
Oracle : Égalité entre entiers OK

- *Ex : Calculer l'itinéraire le plus rapide entre deux villes*

Entrée : Paris – Lyon Sortie attendue : ...A6...
Oracle : Égalité des chemins ? Non

Problème de l’oracle

- *Ex : Trouver le minimum d'une liste d'entiers*

Entrée : [4; 2; 3; 6] Sortie attendue : 2
Oracle : Égalité entre entiers OK

- *Ex : Calculer l'itinéraire le plus rapide entre deux villes*

Entrée : Paris – Lyon Sortie attendue : ...A6...
Oracle : Trajet de 4h17 (quel que soit l'itinéraire choisi) OK

Test statique par inspection

- Se déroule avant toute exécution…
- Réunion en petit comité (3/4) avec modérateur et travail préparatoire (lecture de documents relatifs au module)
- Pas d'analyse/correction des erreurs mises-à-jour
- Compte-rendu des erreurs trouvées pour correction ultérieure
- Méthode:
  - l'auteur expose sa solution
  - check-list des erreurs les plus courantes
- Efficacité:
  - 40% à 70%+ des erreurs de logique et de programmation ?
  - 150/200 instructions à l'heure ?

Un exemple de « Check-list »

Initialisation des variables ?

Constantes nommées ou littérales ?

Validité des prédicats dans les conditionnelles (=, <, ≤, ...)

Prédicats pour la terminaison des boucles (<, ≤, ...)

Bornes inférieures et supérieures des tableaux (0, 1, taille, taille-1)

Délimiteurs des chaînes de caractères (ex: \0 en fin de chaîne de en C)

Correction des allocations dynamiques (allocation/désallocation)

Passage de paramètres dans les sous-programmes (selon les langages)

Gestion des liens dans les listes chaînées

Parenthésage des instructions dans les boucles/conditionnelles

Parenthésage des expressions booléennes

Traitement des exceptions (récupération, propagation)

...

Test statique par analyse

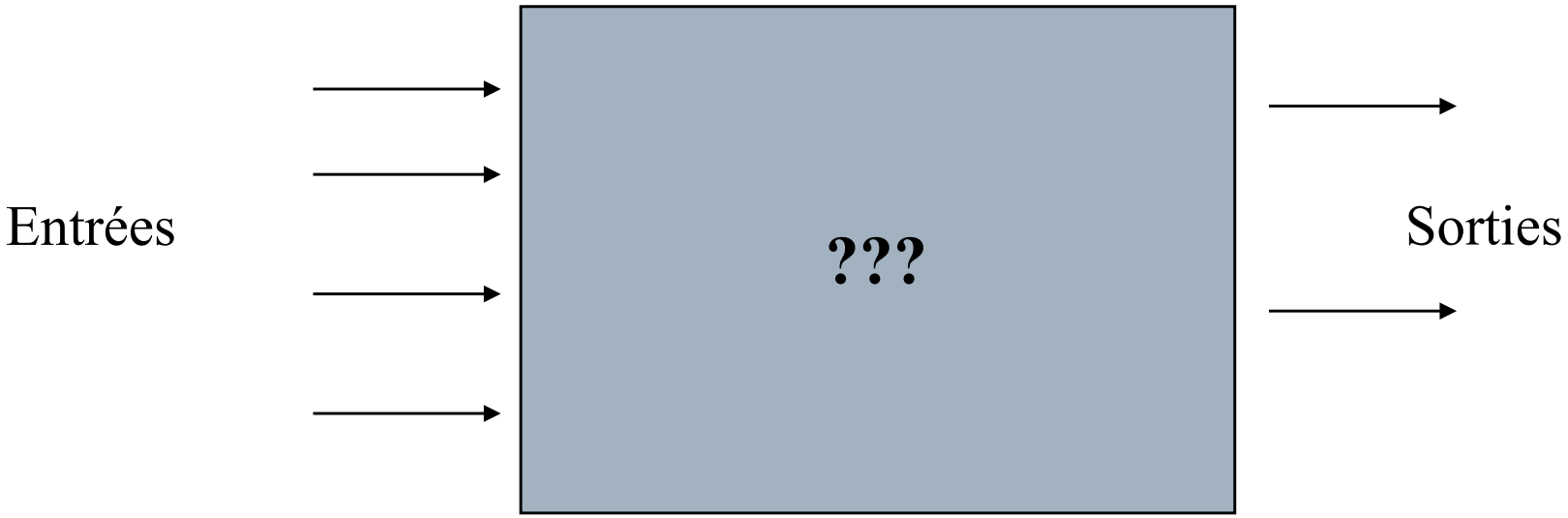
Outils d'analyse de programmes…

Exemple: « Logiscope » (Verilog)

- Analyse du graphe de flot de contrôle, de flot de données
- Détection d'anomalies
- Graphe d'appels entre modules
- Mesures de complexité (Halstead)
- Chemins d'appels entre procédures, entre prédicats, avec conditions de parcours à satisfaire
- Analyse par "évaluation partielle": approximation des calculs
- Instrumentation du code

Test dynamique fonctionnel

- On se base sur les spécifications, pas sur le programme : le programme est une boite noire !



Ce que le programme devrait faire…

Test Boîte noire

Mise en pratique

La spécification:

> *Le programme prend en entrée trois entiers interprétés comme étant les longueurs des côtés d'un triangle. Le programme retourne la propriété du triangle correspondant : scalène, isocèle ou équilatéral.*

Donner un jeu de test pour ce programme… ***;-(***

Les tests - Les triangles

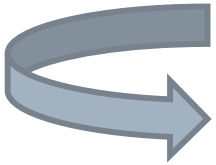
Cas valides	Données	Résultat attendu
Triangle scalène valide	(3,4,5)	scalène
Triangle isocèle valide + permutations	(6,6,7)	Isocèle
Triangle equilatéral valide	(5,5,5)	Équilatéral

Cas invalides	Données	Résultat attendu
Pas un triangle + permutations x+y <z	(1,2,4)	Triangle invalide
Pas un triangle + permutations x+y=z	(1,2,3)	Triangle invalide
Une valeur à nulle	(0,5,4)	Triangle invalide
Toutes les valeurs à 0	(0,0,0)	Triangle invalide
Une valeur négative	(2,-1,6)	Triangle invalide
Une valeur non entière	('a',3,2)	Triangle invalide
Mauvais nombre d'arguments	(3,5)	Triangle invalide

Les tests - Les triangles

- 16 cas correspondant aux défauts constatés dans des implantations de cette spécification

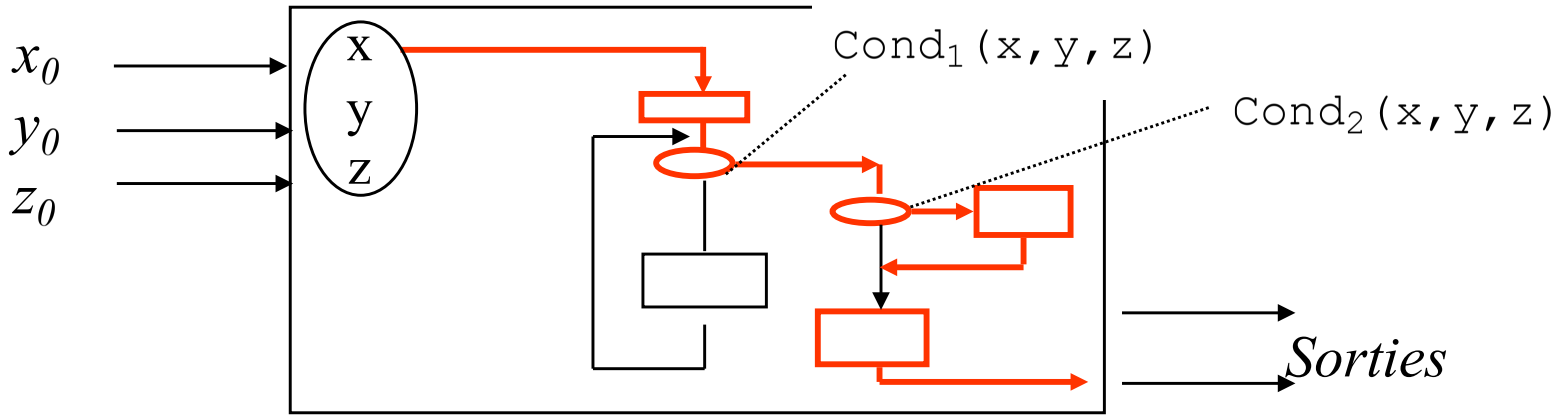
- Moyenne des résultats obtenus par un ensemble de développeurs expérimentés : 55%



La construction de tests est une activité difficile, encore plus sur de grandes applications

Test boîte blanche - Test Dynamique Structurel

- on sélectionne des chemins d'exécution "pertinents"
- la spécification sert à vérifier les résultats obtenus



Ce que le programme fait et comment …

Un chemin = une expression logique sur x_0, y_0, z_0 = un test (ou plusieurs)

$\leftarrow Cond_1(x_0, y_0, z_0) \wedge Cond_2(x_0, y_0, z_0)$

Passer d'un chemin « cible » aux valeurs d'entrée pour le parcourir ?

On s'intéresse soit aux arcs (flot de contrôle), soit aux sommets (flot de données)

Un programme pour les triangles

```
procedure triangle(j,k,l : positive) is
 eg: natural := 0;
begin
if  j + k <= l or k + l  <= j or l + j <= k then
    put("impossible");
else if  j = k  then   eg := eg + 1;  end if;
     if  j = l  then   eg := eg + 1;  end if;
     if  l = k  then   eg := eg + 1;  end if;
      if  eg = 0  then  put("quelconque");
     elsif   eg = 1  then put("isocele");
     else    put("equilateral");
     end if;
end if;
end triangle;
```

Quels sont les tests adaptés à ce programme ?

- essayer un certain nombre de “chemins” d’exécution (lesquels ? tous ?)
- trouver les valeurs d’entrée pour parcourir ces chemins
- comparer les résultats obtenus avec ceux attendus (par la spécification)

Test fonctionnel ou test structurel ?

Les deux se complètent:

- Le test structurel ne suffit pas :
  - si vous avez oublié un cas, c'est la spécification qui a le plus de chance de vous l'indiquer !
- le test fonctionnel ne suffit pas: pour une spécification donnée, il y a plusieurs implémentation possibles, plus ou moins complexes:
  - *recherche dans une table triée : linéaire ? dichotomique ?*
  - *(x, n) → x^n : par multiplication successives ? Par carrés ?*

  Chaque réalisation demande des tests spécifiques !

Programme 1 :

```
S:=1; P:=N;
while P >= 1 loop S:= S*X; P:= P-1; end loop;
```

Programme 2 :

```
S:=1; P:= N;
while P >= 1 loop
    if P mod 2 /= 0 then P := P -1; S := S*X; end if;
    S:= S*S; P := P div 2;
end loop;
```

Ces deux programmes calculent la même chose mais…

- L'un est plus efficace que l'autre, mais plus dur à tester.
- Les valeurs de test pertinentes pour l'un ne le sont pas forcément pour l'autre !

Tests liés aux critères de qualité

- Tests fonctionnels (conformité) : validité
- Tests d'interface utilisaeur : facilité d'utilisation
- Tests d'intégrité des données : sécurité
- Tests de sécurité et de contrôle d'accès : sécurité
- Tests de performance : efficacité
- Tests de charge : robustesse
- Tests d'installation : portabilité

- Critères de réussite : conditions de qualité à atteindre pour arrêter les tests (couverture des cas du(uiliadtion, mesure de performance…)