# IFT3913
# Qualité du logiciel et métriques

## Chapitre 8
## Test du logiciel

## Plan du cours

- Introduction
- Théorie de la mesure
- Qualité du logiciel
- Mesure de la qualité du logiciel
- Études empiriques
- Mesure du produit logiciel
- Collecte et analyse des métriques
- Test du logiciel

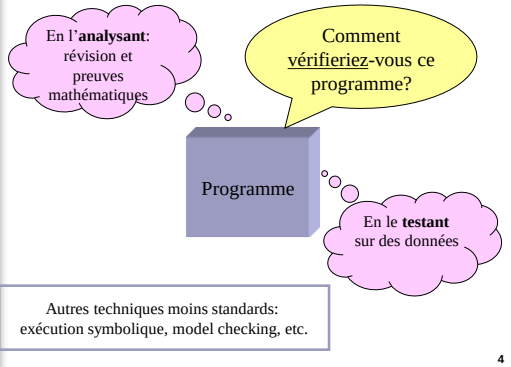


## I. Introduction

- Vérification: Ensemble des activités entreprises pour s'assurer que le logiciel répond à ses objectifs.
- Vérification des qualités d'un logiciel
  - Dans ce chapitre, on s'intéressera principalement à vérifier la « correction » des fonctionnalités d'un logiciel i.e**. vérifier si l'implémentation répond à la spécification des besoins fonctionnels** du système.
  - On pourrait également vérifier les autres qualités du logiciels exprimées par les **besoins non fonctionnels** de la spécification (performance, portabilité, etc.)



## I. Introduction






## I. Introduction

- Validation versus vérification
  - Validation
    - Avez-vous construit le bon produit ?
    - Avez-vous répondu aux besoins du client ?
  - Vérification
    - Avez-vous construit le produit correctement?
    - Avez-vous répondu aux exigences de la spécification ?



## I. Introduction

- Mise en garde
  - Limiter la vérification à l'essai de quelques cas d'exemple:
    - parfois suffisant pour de petits programmes non critiques…
    - malheureux ou grave pour les programmes importants et critiques!

*La vérification d'un logiciel est un problème **indécidable** dans le cas général.*

*Il nécessite une approche rigoureuse, basée sur des principes, des techniques et … de l'intuition!!!*

## II. Approches fondamentales de la vérification

- Approches dynamique et statique
  - **Test:** Approche par expérimentation. On observe le comportement dynamique du logiciel pour voir s'il se comporte tel qu'espéré.
    ***Vérification dynamique***
  - **Analyse:** Approche consistant à déduire, par analyse (preuve formelle), le comportement d'un logiciel à partir de son code et des documents de conception.
    ***Vérification statique***
- Autres approches
  - Exécution symbolique, model checking, etc.

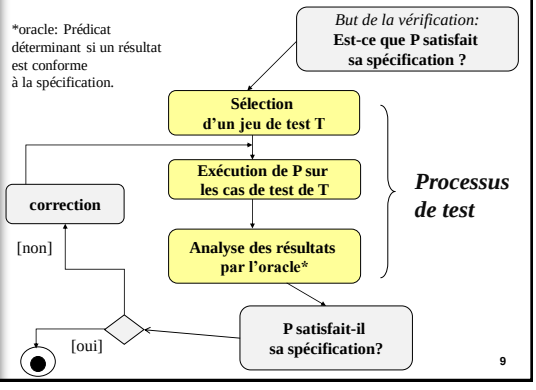
7

## III. Test : Généralités

- Types de test
  - Processus manuel ou automatique
  - Expérimental
- Objectif du test
  - S'assurer qu'un système **vérifie les propriétés** exigées par sa spécification
  - **Détecter des différences** entre les résultats engendrés par le système et ceux qui sont attendus par sa spécification

8

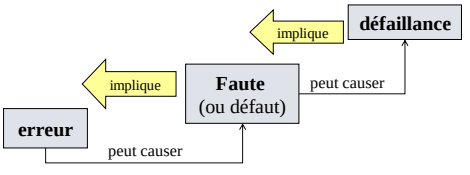
## III. Test : Généralités




9

## III. Test : Généralités

- Défaillance, faute et erreur
  - Défaillance : situation symptomatique ou **manifestation externe** d'une erreur
  - Faute : **état interne incorrect** dans lequel un programme se trouve




10

## III. Test : Généralités

- Exercice
  - Le programme ci-dessous contient une erreur. Trouver un exemple d'exécution où l'erreur engendre
    - ni faute, ni défaillance.
    - une faute mais aucune défaillance.
    - une faute et une défaillance.

```
Function toto(x: in boolean, t: in integer): boolean;
  Var r : integer; rep: boolean;
  If x=true then r := t;
  Else r := t+5;     /* erreur: on voulait t +10 */
  End if;
  If r<20 then rep := true
  Else rep:= false;
  End if;
  Return rep;
End toto
```

11

## III. Test : Généralités

- Peut-on tester un logiciel dans toutes les conditions possibles de fonctionnement ?
  - Rép.: En général, non.
  - **Donc: Il faut choisir des cas de test pertinents !**
- Pourtant, les ingénieurs arrivent, eux, à extrapoler et à conclure, à partir de quelques tests, que leur produit fonctionne bien dans tous les cas! Pourquoi ne pouvons-nous faire de même ?

*Exemple* : on teste un pont pour vérifier sa capacité à supporter des poids lourds. Si le test est concluant pour des camions de 200 tonnes, il est concluant pour tous des camions de poids inférieurs.
*Principe de continuité!*

*Même principe en génie logiciel ???*

12

## III. Test : Généralités

- Principe de continuité
  - une petite différence dans les conditions de fonctionnement est sans conséquence importante sur le comportement du système
- Dans les disciplines d'ingénierie traditionnelle, le principe de continuité s'applique et permet l'extrapolation des résultats
- En génie logiciel
  - **Le principe de continuité ne s'applique pas** la plupart du temps
  - La nature du produit (le logiciel) ne s'y prête pas. Difficile de généraliser en utilisant l'hypothèse du principe de continuité

  *Exemple* : Si le programme fonctionne bien sur les entiers 5, 4 et 3… on ne peut rien conclure sur 2 sans vérifier…

  - Une ligne de code retirée dans un programme peut tout chambouler

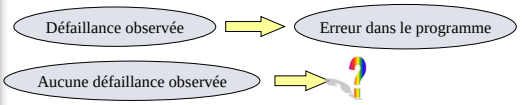


## III. Test : Généralités

- Le test : un problème semi-décidable

  ***Le test logiciel peut être utilisé pour montrer la présence d'erreurs, mais jamais pour prouver leur absence***

  ***[Dijkstra, 1972]***
  - Si le logiciel se comporte mal sur un ou plusieurs cas de test: on est sûr qu'il y a une erreur
  - Si le logiciel se comporte bien sur un ensemble fini de cas de test: on ne peut rien conclure




Le test permet tout de même d'**augmenter notre confiance** en la qualité du logiciel (même si ce n'est pas une certitude absolue)



## III. Test : Généralités

- Un procédé aléatoire ?
  - Il est important d'utiliser **des techniques de test systématiques et non aléatoire**. Pourquoi ?

    *Exemple* : on génère au hasard des cas de test pour le programme ci-dessous
    - Read(x); Read(y);
    - If x= y then z:=2; /* erreur: on voulait z:=22 */
    - Else z:=0
    - End if
  - **Peu de chance de tomber au hasard sur un cas qui vérifie la condition** x=y et qui permette de révéler une défaillance (et donc l'erreur)

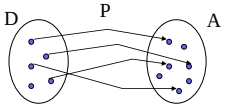


## III. Test : Généralités

- Caractéristiques
  - Permet de **détecter la présence d'erreurs** dans un logiciel (par l'observation des défaillances qu'elles induisent)
  - Doit produire les **mêmes résultats lorsque que répété avec les mêmes données et dans le même environnement** (attention aux variables et paramètres non initialisés: leur valeur initiale (non définie) peut varier d'une exécution à l'autre)
  - Doit être **exact et précis**. Une spécification formelle peut contribuer à la réalisation de tests précis et peut servir à la formulation des oracles. Comment ?
  - Peut parfois contribuer à **identifier la localisation** précise des erreurs dans le logiciel



## IV. Fondements théoriques du test




- Soit P un programme.
- Soit D et A, respectivement les ensembles d'entrée et de sortie de P tels que
  - D *(Départs)* contient toutes les valeurs qui peuvent être soumises en entrée à P
  - Tout résultat obtenu suite à l'exécution de P (s'il y en a un) se trouve dans A *(Arrivés)*
- P peut être considéré comme une fonction partielle de D dans A.

  N.b. Si P est un programme non déterministe, P pourrait ne pas être une fonction



## IV. Fondements théoriques du test

- Définitions
  - Soit **OR** (*Output Requirements ou ORacle*), l'ensemble des conditions que les résultats de P doivent satisfaire (conformément à sa spécification)
  - Soit $d \in D$. P est dit correct pour d,

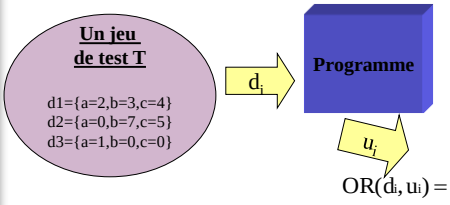
    si P(d) satisfait OR.

    $P(d) = u \rightarrow OR(d,u),\ \text{où } u \in A$
  - **P est correct, s'il est correct pour tout d ϵ D.**

    $\forall d \in D\ \ (P(d) = u \rightarrow OR(d,u)), \text{où } u \in A$

## IV. Fondements théoriques du test

- Jeu de test
  - Un cas de test est un élément d ∈ D
  - Un jeu de test T est un ensemble fini de cas de test i.e. un sous-ensemble fini de D






## IV. Fondements théoriques du test

- Jeu de test idéal
  - Un jeu de test T est dit idéal, si **chaque fois que P est incorrect** (i.e. défaillant), il existe un cas d ∈ T tel que P est incorrect pour d
  - Rarement possible
- Jeu de test pertinent
  - Qui a **de forte chance de révéler** la présence d'une erreur dans un programme
  - Comment sélectionner des jeux de test pertinents ?
  - Compromis entre le coût d'exécution et le degré de confiance



## V. Techniques de test

- Selon les phases du cycle de vie
  - **Test unitaire** : Test d'un programme ou d'un module isolé dans le but de s'assurer qu'il ne comporte pas d'erreur d'analyse ou de programmation
  - **Test d'intégration** : Une progression ordonnée de tests dans laquelle des éléments logiciels et matériels sont assemblés et testés jusqu'à ce que l'ensemble du système soit testé



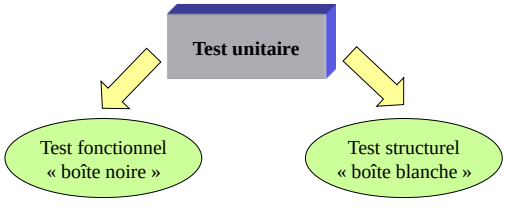
## V. Techniques de test

- Selon les phases du cycle de vie
  - **Test système**
    - **Test de réception** : Essai d'un matériel, d'un logiciel ou d'un ensemble matériel-logiciel après installation, effectué par l'acquéreur dans ses locaux avec la participation du fournisseur, afin de **vérifier que les dispositions contractuelles ont été respectées**.
    - **Test de régression**: Test visant à **détecter les erreurs introduites après chaque changement** apporté dans les programmes d'un système



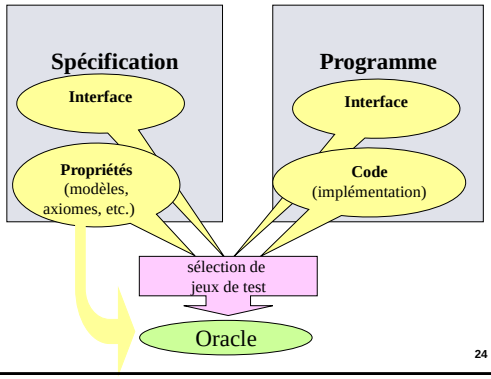
## V. Techniques de test

- Le test à « petite échelle »
  - On s'intéresse à la sélection de jeux de test pour le test à « petite échelle ».






## VI. Sélection de jeux de test



# VI. Sélection de jeux de test

- Méthodes de sélection de jeux de test unitaire

| Spécification → / Programme ↓ | Interface de la spécification | Propriétés de la spécification |
|---|---|---|
| Interface du programme | Technique de test aléatoire (non recommandé) | **Technique de test fonctionnel (boîte noire)** |
| Code du programme | **Technique de test structurel (boîte blanche)** | |



# VII. Test boîte noire (fonctionnel)

- Principes de la boîte noire:
  - On teste le logiciel sans prendre connaissance des détails de sa conception et de son implémentation
  - On établit des jeux de test et **on évalue les résultats sur la base de la spécification uniquement**
  - On teste ce que le programme est supposé faire
  - Permet éventuellement de détecter les erreurs commises et les omissions.



# VII. Test boîte noire (fonctionnel)

- Il existe deux approches principales pour la sélection de jeux de test de boîte noire

A. Partition du domaine des entrées en classes d'équivalence
B. Analyse des valeurs frontières



# VII. Test boîte noire (fonctionnel)

A. Partition du domaine des entrées en classes d'équivalence
   1. On détermine le domaine D des valeurs d'entrée à partir des interfaces (spécification & programme).
   2. On partitionne ce domaine D en classes d'équivalence $D_i$ de façon à ce que …
      - **le programme se comporte à peu près de la même manière sur toute les valeurs d'entrée appartement à une même classe**
   3. On construit un jeu de test en sélectionnant un représentant par classe d'équivalence

$$\text{Soit } D = \bigcup_{i=1..n} D_i \text{ où } \forall i \neq j . D_i \cap D_j = \{\ \}$$

$$\{d_1 \in D_1, ..., d_n \in D_n\} \quad \text{est un jeu de test}$$



# VII. Test boîte noire (fonctionnel)

A. Partition du domaine des entrées en classes d'équivalence
   - Exemples de partitionnement (Lorsque le domaine d'entrée est un intervalle de valeurs)
     - Par exemple: Soit D = Entiers et P défini sur [0,5000]
     - On définit 3 classes d'équivalence:
       - Une classe d'équivalence pour les valeurs appartenant à l'intervalle de valeurs valides. $D_1 = \{0 \le d \le 5000\}$
       - Une classe d'équivalence pour les valeurs d'entrées invalides inférieures à l'intervalle. $D_2 = \{d < 0\}$
       - Une classe d'équivalence pour les valeurs d'entrées invalides supérieures à l'intervalle. $D_3 = \{d > 5000\}$
     - Un jeu de test valide serait: T={10,-1,6008}



# VII. Test boîte noire (fonctionnel)

A. Partition du domaine des entrées en classes d'équivalence
   - Exemples de partitionnement (Lorsque le domaine d'entrée est un ensemble fini de valeurs discrètes)
     - Par exemple: D = Couleurs = {bleu, blanc, rouge, vert, noir, orange} et P défini sur {blanc, rouge, vert}
     - On définit deux classes d'équivalence:
       - Une classe d'équivalence pour les valeurs d'entrée valides. D1= {blanc, rouge, vert}
       - Une autre classe d'équivalence pour les valeurs d'entrées invalides devrait aussi être définie. D1= {bleu, noir, orange}
     - Un jeu de test valide serait T={blanc, orange}

## VII. Test boîte noire (fonctionnel)

B. Analyse des valeurs frontières
- Erreurs souvent aux frontières de comportement
  - indice de tableau tout juste trop grand ou trop petit
  - boucles avec une itération en trop ou en manque
  - comparaisons et expressions conditionnelles
  - …
- Analyse précise aux bornes des classes
  - plusieurs représentants par classe d'équivalence
    - une ou plusieurs valeurs aux bornes
    - une valeur « médiane »



## VII. Test boîte noire (fonctionnel)

B. Analyse des valeurs frontières
1. On procède comme dans le cas de la partition du domaine des entrées en classes d'équivalence (étape 1 et 2)
2. On choisit les cas de test aux frontières des différentes classes d'équivalence

Par exemple: Soit D = Entiers et P défini sur [0,5000]

$$D_1 = \{0 \le d \le 5000\}$$
$$D_2 = \{d < 0\}$$
$$D_3 = \{d > 5000\}$$

Un jeu de test valide serait: T={-1,0,5000,5001}

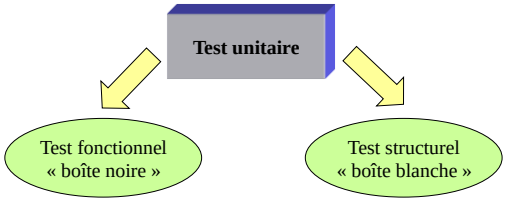


## VII. Test boîte noire (fonctionnel)

- Avantages
  - Le jeu de test sélectionné peut **garantir une bonne couverture du domaine des entrées** du programme
  - Des **oublis par rapport à la spécification de l'application peuvent être détectés**
  - Lors de modifications du programme ne remettant pas en cause la spécification, il est possible de **réutiliser le jeu de tests** précédent pour valider la nouvelle version
- Inconvénient
  - Ne donne **pas d'information à propos de la localisation** des erreurs



## V. Techniques de test






## VIII. Test boîte blanche (structurel)

- Principes de la boîte blanche
  - On teste le programme en tenant compte de sa structure interne
  - On établit des jeux de test en fonction de la conception détaillée du programme.
  - **On teste ce que le programme fait**
  - Permet éventuellement de détecter les erreurs commises… mais pas les omissions!



## VIII. Test boîte blanche (structurel)

- Pourquoi se baser sur la structure du programme pour le tester ?
  - Les erreurs ont tendance à se concentrer dans des chemins, instructions, conditions qui sont **hors de l'«exécution normale»**
  - On a **tendance à croire qu'un chemin particulier a peu de chances d'être exécuté** alors qu'il l'est très souvent
  - Les **erreurs typographiques sont réparties au hasard**

# VIII. Test boîte blanche (structurel)

- Critères de sélection de jeux de test
  A. Critère de couverture des instructions
  B. Critère de couverture des arcs du graphe de flot de contrôle
  C. Critère de couverture des chemins indépendants du graphe de flot de contrôle
  D. Critère de couverture des conditions
  E. Critère de couverture des i-chemins



# VIII. Test boîte blanche (structurel)

A. Critère de couverture des instructions

« Sélectionner un jeu de test T tel que, lorsqu'on exécute P sur les d ∈ T, **chaque instruction de P est exécutée** au moins une fois. »

Exemple: Algorithme d'Euclide

```
begin
  read(x); read(y)
  while x <> y do
        If x>y then x:= x-y;
        else y:= y-x;
        end if
  end while
  pgcd := x;
end
```

Trouver un jeu de test qui permet de couvrir toutes les instructions du programme.

•Pour constituer les jeux de test, on va tenter de grouper dans des classes Di les éléments du domaine d'entrée D qui activent les mêmes instructions dans P:
•D1={(x,y) | x=y}
•D2={(x,y) | x>y};
•D3={(x,y) | x<y};

•Jeu de test: {(4,2), (2,4), (3,3)}

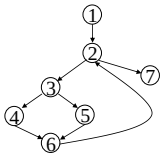


# VIII. Test boîte blanche (structurel)

B. Critère de couverture des arcs du graphe de flot de contrôle

– Au lieu de s'intéresser aux instructions, on s'intéresse ici aux branchements de contrôle conditionnels dans un programme.
– Un programme bien structuré peut être représenté par un *graphe de flot de contrôle (GFC)*
– Chap. IV – Mesure du logiciel, pour les détails sur les GFC

•Les sommets représentent les instructions
•Les arcs représentent le flot de contrôle entre les instruction.






# VIII. Test boîte blanche (structurel)

B. Critère de couverture des arcs du graphe de flot de contrôle

« Sélectionner un jeu de test T tel que, lorsqu'on exécute P sur les d ∈ T, **chaque arc du graphe de flot de contrôle de P est traversé** au moins une fois. »

– Pour chaque instruction conditionnelle (if, while) on teste le cas où la condition est vraie et celui où elle est fausse..
– Critère de sélection plus fort que celui de la couverture des instructions.



# VIII. Test boîte blanche (structurel)

C. Critère de couverture des chemins indépendants

« Sélectionner un jeu de test T tel que, lorsqu'on exécute P sur les d ∈ T, tous les 1-chemins du graphe de flot de P sont parcourus au moins une fois. »

– Chemin = Séquence de nœuds et d'arcs dans le graphe de flot de contrôle, initiée depuis le nœud de départ jusqu'à un nœud terminal. (Il peut y avoir plusieurs nœuds terminaux dans un programme.)
  - 1-chemin : Chemin parcourant les boucles 0 ou 1 fois.
  - Chemin indépendant : (1-)chemin du graphe de flot de contrôle qui parcourt au moins un nouvel arc par rapport aux autres chemins définis dans une base B (i.e. ce chemin introduit au moins une nouvelle instruction non parcourue).



# VIII. Test boîte blanche (structurel)

C. Critère de couverture des chemins indépendants

– Méthode de sélection des jeux de test
  1. Construire le graphe de flot de contrôle de P
  2. Déterminer la complexité cyclomatique *V(G)* du GFC
     – Constitue une borne supérieure sur le nombre de chemins nécessaires pour couvrir tous les chemins indépendants du graphe de flot d'un programme.
  3. Définir un ensemble de base B de chemins indépendants dans le graphe.
  4. Construire un jeu de test qui permettra l'exécution de chaque chemin de l'ensemble B.

# VIII. Test boîte blanche (structurel)

C. Critère de couverture des chemins indépendants

– Méthode de sélection des jeux de test

3. Définir un ensemble de base B de chemins indépendants dans le graphe
   – *Chemin indépendant* : chemin du graphe de flot de contrôle qui parcourt **au moins un nouvel arc** par rapport aux autres chemins définis (introduit au moins une nouvelle instruction non parcourue).
   – Une *base* comportant V(G) chemins nous assure de couvrir tous chemins indépendants du graphe de flot G.
   – *Mais on ne couvre pas nécessairement tous les 1-chemins du graphe…*



# VIII. Test boîte blanche (structurel)

C. Critère de couverture des chemins indépendants

– Chemins linéairement indépendants

- La représentation vectorielle d'un chemin est un vecteur qui compte le nombre d'occurrence de chaque arc.

  Ex. : Chemin1 = (1, 0, 0, 0, 0, 0, 0, 1)

- Un ensemble de chemins est linéairement indépendant si aucun ne peut être représenté par une combinaison linéaire des autres (en représentation vectorielle).

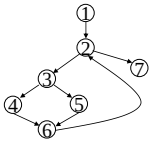


# VIII. Test boîte blanche (structurel)

C. Critère de couverture des chemins indépendants

– Chemins linéairement indépendants

Algorithme d' Euclide




Par exemple:

- 1-2-7
- 1-2-3-4-6-2-7 (nouveaux arcs: 2-3, 3-4, 4-6, 6-2)
- 1-2-3-5-6-2-7 (nouveaux arcs: 3-5, 5-6)



# VIII. Test boîte blanche (structurel)

C. Critère de couverture des chemins indépendants

– Méthode de sélection des jeux de test

4. Sélection des jeux de test

- Pour chaque chemin indépendant de la base, on doit trouver un jeu de test qui permette de le traverser (en itérant possiblement sur certains segments du chemin).
- Cette sélection peut être ardue dans le cas de gros programmes!
- En effet, ceci équivaut à
  – Résoudre un système de contraintes composé des nœuds prédicats qui se trouvent sur le chemin à parcourir.
  – Attention: Tous les chemins ne sont pas nécessairement satisfiables!!! (Problème indécidable)



# VIII. Test boîte blanche (structurel)

C. Critère de couverture des chemins indépendants

– Méthode de sélection des jeux de test

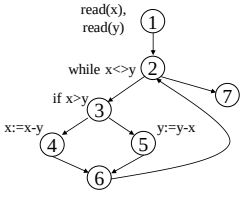
4. Sélection des jeux de test

Partitionnement

**Chemin 1-2-7**
D1= {(x,y) | x=y}
Cas de test: <x=3,y=3>

**Chemin 1-2-3-4-6-2-7**
D2 = {(x,y)| x>y, x-y=y}
Cas de test: <x=8,y=4>

**Chemin 1-2-3-5-6-2-7**
D3 = {(x,y)| x<y, x=y-x}
Cas de test: <x=3,y=6>




Jeu de test: {<x=3 y=3>, <x=8, y=4>, <x=3, y=6>}



# VIII. Test boîte blanche (structurel)

C. Critère de couverture des chemins indépendants

– Méthode de sélection des jeux de test

4. Sélection des jeux de test - Limites

```
found:= false; counter:=1;
While (not found) and counter< numberItems do    /* erreur <= */
  If table(counter) = desiredElem then
       found := true;
  endif
  counter:= counter+1;
End while;
If found then
  write(« Elément existe. »);
Else
  write(« Élément n'existe pas. »);
Endif;
```

En effet, qu'arrive-t-il si on cherche « a » dans [« a »] ?

**Limite…**

Ex: Soit le jeu de test suivant qui parcourt tous les chemins indépendants du graphe (il y en a 4, mais l'un d'eux est impossible à parcourir) :

- table vide
- table avec 1 élément ne contenant pas celui désiré.
- table avec 3 éléments dont le premier est celui cherché.

Malheureusement, on n'a pas découvert l'erreur !

# VIII. Test boîte blanche (structurel)

D. Critère de couverture des conditions (et des arcs)

« Sélectionner un jeu de test T tel que, lorsqu'on exécute P sur chacun des d ∈ T, chaque arc du graphe de flot de contrôle de P est traversé au moins une fois et toutes les valeurs possibles des constituantes des conditions complexes sont calculées au moins une fois. »



# VIII. Test boîte blanche (structurel)

D. Critère de couverture des conditions (et des arcs)

– Décomposer pour tenter repérer les erreurs plus subtiles dans l'évaluation des conditions composées et tenir compte de toutes les valeurs possibles de leurs parties composantes…

– Conditions composées

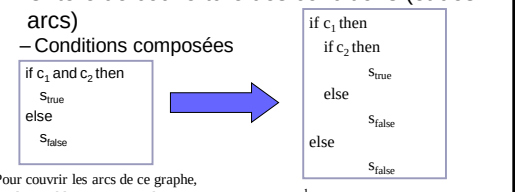
- Condition composée = conjonction ou disjonction de deux ou plusieurs clauses élémentaires (i.e. formules atomiques ou leur négation).
  - Ex. Voici une condition composée:
  - x>4 and y>8 and z<>0
  - où il y a trois clauses élémentaires.



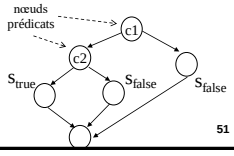
# VIII. Test boîte blanche (structurel)

D. Critère de couverture des conditions (et des arcs)

– Conditions composées




Pour couvrir les arcs de ce graphe, on devra obligatoirement choisir un jeu de test où **toutes les valeurs possibles de c1 et c2** sont expérimentées au moins une fois.
Attention: toutes les **combinaisons** de valeurs de c1 et c2 ne sont toutefois pas expérimentées.

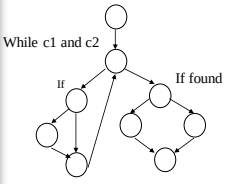


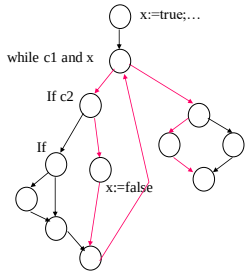



# VIII. Test boîte blanche (structurel)

D. Critère de couverture des conditions (et des arcs)

– Conditions composées






# VIII. Test boîte blanche (structurel)

- Malgré tout il demeure des erreurs non détectées par les jeux de test satisfaisant les critères portant sur les conditions et sur les chemins indépendants…

```
If x<>0 then y:=5;
Else z:=z-x;
Endif;
If z>1 then z:=z/x;   /*risque de division par 0 */
Else z:=0;
Endif;
```

Jeu de test qui couvre tous les arcs et conditions:
{<x=0, z=1>,
<x=1, z=3>}

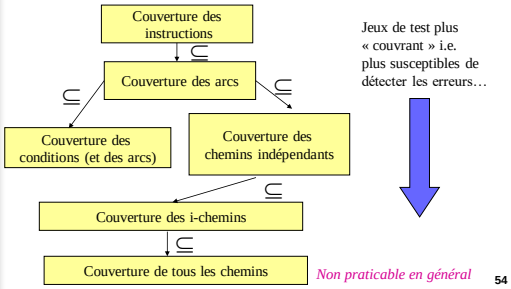
– Pour sélectionner le bon jeu de test, il faudrait employer un critère de couverture de tous les i-chemins d'exécution possibles (pas seulement les 1-chemins indépendants mais tous les i-chemins!).

– Mais ce critère est difficilement applicable en pratique à cause de la complexité de la sélection des jeux de test correspondants!…



# VIII. Test boîte blanche (structurel)

- Degré de couverture des jeux de test générés par les différents critères




Jeux de test plus « couvrant » i.e. plus susceptibles de détecter les erreurs…

*Non praticable en général*

## VIII. Test boîte blanche (structurel)

- Couverture des i-chemins – Sélection de jeux de test pour parcourir les boucles…
  - Cas des boucles simples (bornées par n itérations autorisées)
    - Trouver un jeu de test permettant de couvrir chacun des cas suivants
      - on saute la boucle
      - une itération de la boucle
      - deux itérations
      - m itérations (m<n)
      - n-1, n et n+1 itérations…



## VIII. Test boîte blanche (structurel)

- Couverture des i-chemins – Sélection de jeux de test pour parcourir les boucles…
  - Cas des boucles imbriquées
    1. Commencer par la boucle la plus intérieure, les indices des autres boucles étant à leur valeur minimale.
    2. Appliquer les tests de boucle simple à la boucle la plus interne.
    3. Passer à la suivante, les boucles externes maintenues à leur valeur minimale, les internes à valeur « typique ».
    4. Continuer vers la boucle la plus externe.



## VIII. Test boîte blanche (structurel)

- Couverture des i-chemins – Sélection de jeux de test pour parcourir les boucles…
  - Boucles concaténées
    - Si réellement indépendantes: traiter chacune comme une boucle simple.
    - Si dépendantes (index de la 1re boucle = valeur initiale de l'index de la 2e boucle) : approche des boucles imbriquées.
  - Boucles non structurées
    - Si possible, recoder ou refaire la conception.

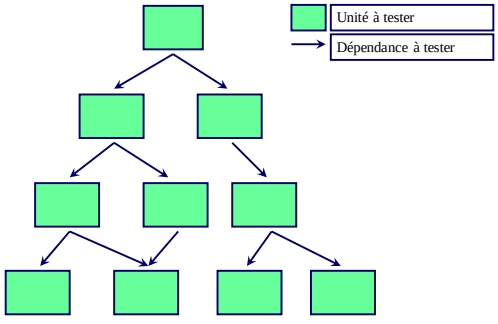


## IX. Test d'intégration

- Objectif
  - Vérifier l'interaction entre unités (méthode, classe ou package).
- Difficultés principales de l'intégration
  - Interfaces floues (ex. ordre des paramètres de même type).
  - Implantation non conforme à la spécification (ex. dépendances entre unités non spécifiées).
  - Réutilisation d'unités (ex. risque d'utilisation hors domaine).



## Architecture de dépendances

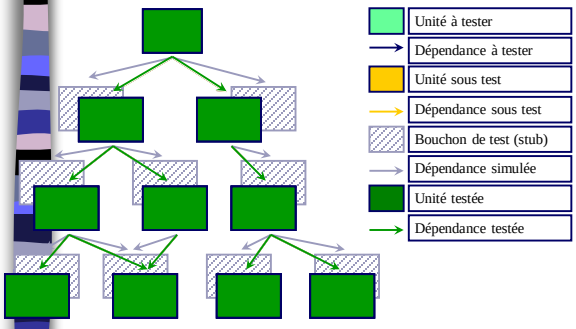





## Intégration – Approches classiques

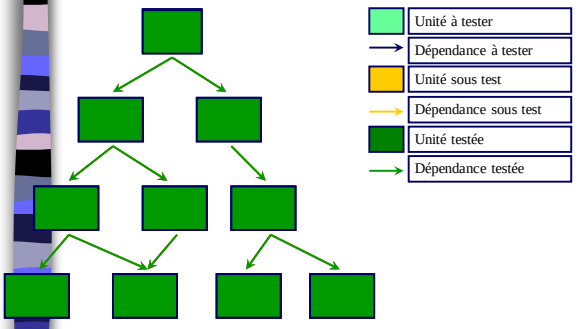
- Architecture arborescente
- Approches
  - De haut en bas (top-down)
  - De bas en haut (bottom-up)

## Approche classique : De haut en bas




61

## Approche classique : De bas en haut




62