Test de logiciel

1 – Aspects « théoriques »

2 – Aspects pratiques

Introduction

- Il est nécessaire d'assurer la fiabilité des logiciels
  - Domaines critiques : atteindre une très haute qualité imposée par les lois/normes/assurances/
  - Autres domaines : atteindre le rapport qualité/prix jugé optimal (attentes du client)
- Assurer la fiabilité du logiciel est une part cruciale du développement
  - Plus de 50% du développement d'un logiciel critique
  - Plus de 30% du développement d'un logiciel standard
- Plusieurs études ont été conduites et indiquent que
  - Entre 30 et 85 erreurs sont introduites par portion de 1000 lignes de code produites (logiciel standard).
- Ces chiffres peuvent fortement augmenter si les méthodes permettant d'assurer la fiabilité sont mal gérées au sein du projet
  - Voir la part importante de l'activité de tests dans les différentes méthodes de conception de logiciels

Introduction

- Dans le cycle de vie du logiciel, on assure la fiabilité / qualité du logiciel par l'activité de Vérification et de Validation
  - Vérification
    - le développement est-il correct par rapport à la spécification initiale ?
    - Est-ce que le logiciel fonctionne correctement?
  - Validation
    - a-t-on décrit le « bon » système ?
    - est-ce que le logiciel fait ce que le client veut ?
- Les tests ont un rôle prépondérant

Outils utilisés

- *Eclipse*
  - Environnement de développement intégré
  - www.eclipse.org
- *JUnit*
  - Canevas pour les test unitaires
  - www.junit.org
- *Emma*
  - Outil d'analyse de couverture structurel
  - emma.sourceforge.net
  - *EclEmma, plugin eclipse (help→Eclipse Marketplace)*
- *Mockito*
  - Outil pour la gestion de mock/doublures/simulacres
  - http://code.google.com/p/mockito/
  - Récupérer la dernière release mockito-all-1.9.0-rc1.jar

Qu'est-ce que tester ?

- Tester c'est réaliser l'exécution du programme pour y trouver des défauts et non pas pour démontrer que le programme ne contient plus d' erreur
  - Le but d'un testeur est de trouver des défauts à un logiciel
  - Si ce n'est pas son but, il ne trouve pas
    → aspect psychologique du test
  - C'est pourquoi il est bon que
    - Le testeur ne soit pas le développeur
      - souvent difficile
      - Valable que pour certains types de tests
    - Les tests soient écrits avant le développement
      - cf. le développement dirigé par les tests (test-driven development) proné par l' eXtrem Programming

Traits caractéristiques d' un test

- Définition IEEE

> 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
> *(Standard Glossary of Software Engineering Terminology)*

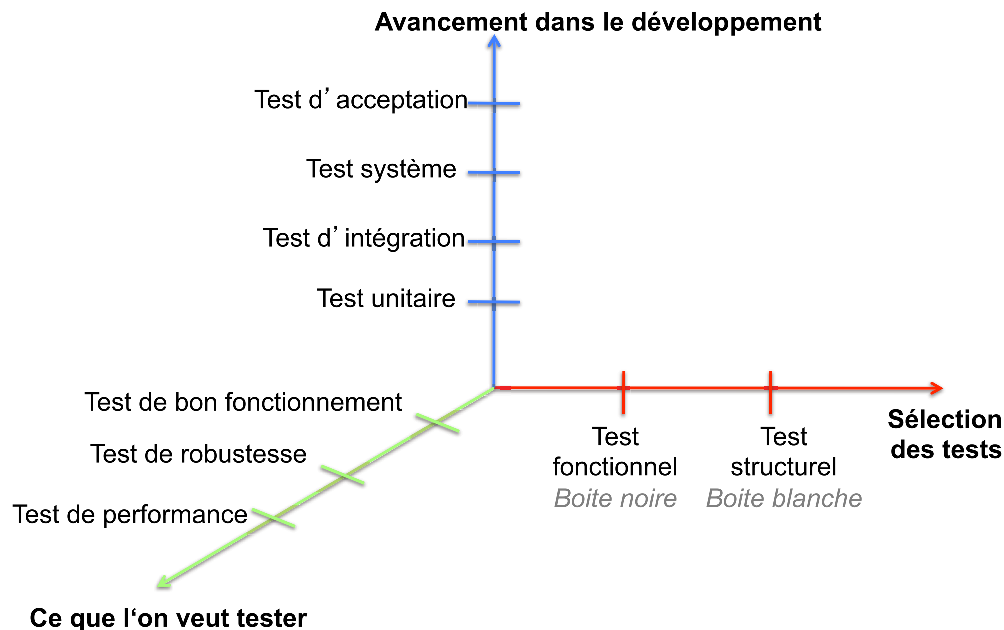
- Le test est une méthode **dynamique** visant à trouver des bugs

> 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)*

- Le test est une méthode de validation **partielle** des logiciels

> Tester permet seulement de révéler la présence d'erreurs mais jamais leur absence.
> *E. W. Dijkstra (Notes on Structured Programming, 1972)*

Quels types de tests ?



Le test dans le cycle de vie du logiciel

- Résumé

Test unitaire	Tester les modules, classes, opérations, au fur et à mesure de leur développement	Fait par le développeur du module (avant ou après écriture)
Test d' intégration	Tester l' assemblage des modules, des composants et leurs interfaces	Fait par le développeur du module ou un autre développeur
Test de système	Tester les fonctionnalités du système dans son ensemble	Développeurs
Test d' acceptation	Confirmer que le système est prêt à être livré et installé	Client

Types de test – ce que l'on veut tester!

- **Tests nominal ou test de bon fonctionnement**
  - Les cas de test correspondent à des données d'entrée valide.
    Test-to-pass
- **Tests de robustesse**
  - Les cas de test correspondent à des données d'entrée invalides
    Test-to-fail
- Règle usuelle : Les tests nominaux sont passés avant les tests de robustesse
- **Tests de performance**
  - *Load testing* (test avec montée en charge)
  - *Stress testing* (soumis à des demandes de ressources anormales)

Sélection de tests

Deux (trois?) grandes familles de sélection de tests

1. Test fonctionnel ou boîte noire
   - test choisi à partir des spécifications (use-case, user story, etc.)
   - évaluation de l'extérieur (sans regarder le code), uniquement en fonction des entrées et des sorties
   - sur le logiciel ou un de ses composants

   (+) taille des spécifications, facilité oracle

   (-) spécifications parfois peu précises, concrétisation des DT
2. Test structurel ou boîte blanche
   - test choisi à partir du code source (portion de codes, blocs, branches)

   (+) description précise, facilité script de test

   (-) oracle difficile, manque d'abstraction

Sélection de tests

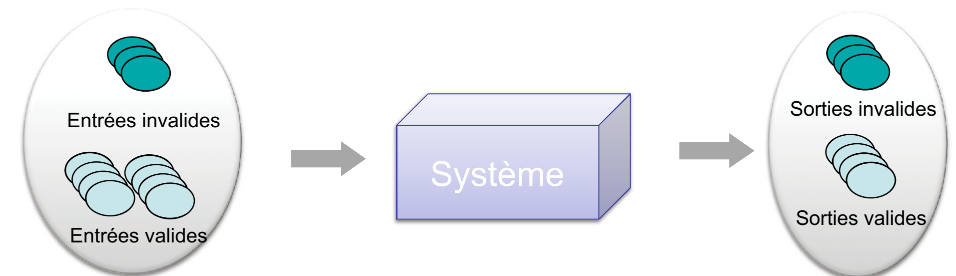
- Les tests boîte noire et boîte blanche sont complémentaires
  - Les approches structurelles trouvent plus facilement les défauts de programmation
  - Les approches fonctionnelles trouvent plus facilement les erreurs d'omission et de spécification
- Exemple: fonction retournant la somme de 2 entiers 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)
```

  - Avec l'approche fonctionnelle (boîte noire),
    - le bug 1 est difficile à détecter, le bug 2 est facile à détecter
  - Avec l'approche structurelle (boîte blanche),
    - le bug 1 est facile à détecter, le bug 2 est difficile à détecter

BN – Classes d'équivalence (1/3)

- Principe : diviser le domaine des entrées en un nombre fini de classes tel que le programme réagisse de la même façon pour toutes les valeurs d'une classe
  - conséquence : il ne faut tester qu'une valeur par classe !
  - évite des tests redondants (si classes bien identifiées)



- 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 un ou quelques CTs pour chaque classe

BN – Classes d' équivalence – Exemple 1

- Supposons que la donnée à l'entrée est un entier naturel qui doit contenir cinq chiffres, donc un nombre compris entre 10,000 et 99,999.

- Le partitionnement en classes d'équivalence identifierait alors les trois classes suivantes (trois conditions possibles pour l'entrée):
  a. N < 10000
  b. 10000 <= N <= 99999
  c. N >= 100000

- On va alors choisir des cas de test représentatif de chaque classe, par exemple, au milieu et aux frontières (cas limites) de chacune des classes:
  a. 0, 5000, 9999
  b. 10000, 50000, 99999
  c. 100000, 100001, 200000

BN – Classes d' équivalence – Exemple 2

Retour à l' exemple célèbre (G.J. Myers, *The Art of Software Testing*, 1979)

> Un programme prend 3 entiers en entrée, qui sont interprétés comme représentant les longueurs des côtés d'un triangle. Le programme doit retourner un résultat précisant s'il s'agit d'un triangle scalène, isocèle ou équilatéral.

Quels cas test pour ce programme avec des classes d' équivalence ?

- Pas un triangle
  1 côté à une longueur supérieure à la somme des 2 autres
- Isocèle
  2 côtés égaux uniquement
- Équilatéral
  3 côtés égaux
- Scalène
  Les autres

BN – Classes d' équivalence – Exemple 3

- Soit un programme calculant la valeur absolue d'un entier relatif saisi sous forme d'une chaîne de caractères au clavier, *quelles classes d'équivalences pour le tester ?*

BN – Classes d' équivalence – Exemple 3

- Soit un programme calculant la valeur absolue d'un entier relatif saisi sous forme d'une chaîne de caractères au clavier, *quelles classes d'équivalences pour le tester ?*

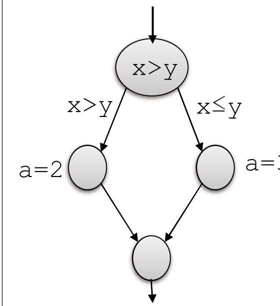
- **Classe 1**
  Entrée invalide, chaîne vide
- **Classe 2**
  Entrée invalide, chaîne avec plusieurs mots
  « 123 983 321 »
- **Classe 3**
  Entrée invalide, un seul mot mais pas un entier relatif
  « 123.az+23 »
- **Classe 4**
  Entrée valide, un mot représentant un entier relatif positif ou nul
  « 673.24»
- **Classe 5**
  Entrée valide, un mot représentant un entier relatif négatif
  « -2.4 »

Méthodes de sélection de tests

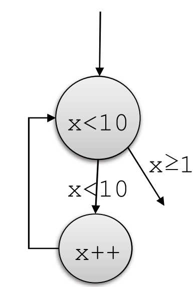
- Méthodes Boîte Blanche
  - couverture structurelle
  - *Mutation*

    sélection des CT par rapport à leur effet au changement sur système

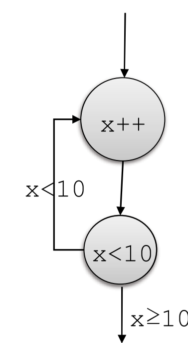
BB – Les graphes de flot pour diverses structures de contrôle



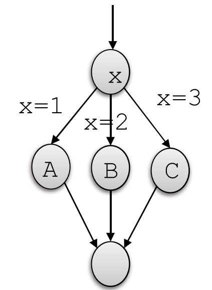

```
if (x>y)
  a = 2;
else
  a = 3;
```




```
while(x<10)
     x++;
```



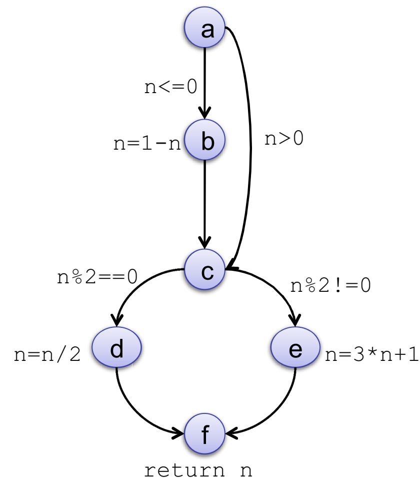

```
do
   x++;
while (x<10)
```




```
switch(x) {
case 1:
      A;break;
case 2:
      B;break;
case 3:
      C;break;
}
```

BB – Exercice 1

```
int f(int n){
 if (n<=0)
   n = 1-n;
 if (n%2==0)
   n = n/2;
 else
   n = 3*n+1;
 return n;
}
```

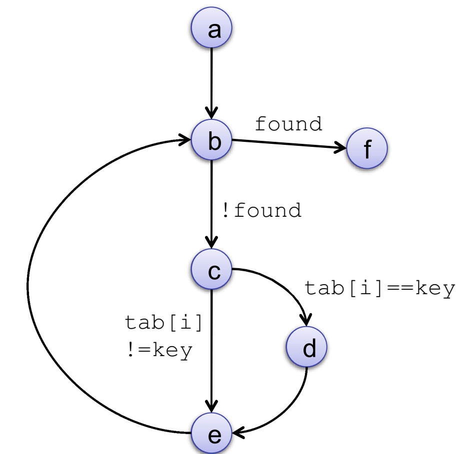


1. Etablir le graphe de flot de contrôle de ce programme
2. Fournir l'expression des chemins

a (1 + b) c (e + d) f

BB – Exercice 2

```
int f(int* tab, int key){
  [                        a ]
 while(!found){
  if(tab[i]==key){
     [                     d ]
  }
  [                        e ]
 }
 [                         f ]
}
```



1. Etablir le graphe de flot de contrôle de ce programme
2. Fournir l'expression des chemins

ab (c (1 + d) eb)* f

BB – Couverture du CFG

Pour trouver quels chemins parcourir pour couvrir le plus de comportements du système, voici quelques critères de couverture sur un flot de contrôle

- Tous les nœuds (I) : le plus faible
  - Couverture des instructions, **niveau 1**
- Tous les arcs / décisions (D) : test de chaque décision
  - Couverture des branchements et conditions, **niveau 2**
- Tous les chemins : le plus fort, impossible à réaliser s'il y a des boucles
  - Couverture exhaustive, **niveau 0**
  - De plus, attention, certains chemins peuvent ne pas être atteignables
- On peut baser des méthodes de couverture sur ces critères ou en utiliser de plus avancés

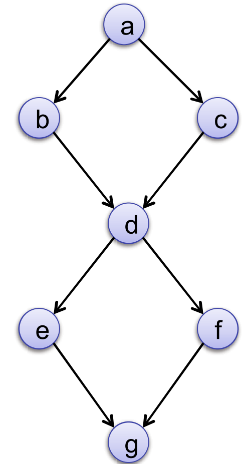
BB – Couverture du CFG

Comment effectuer la couverture du CFG ?

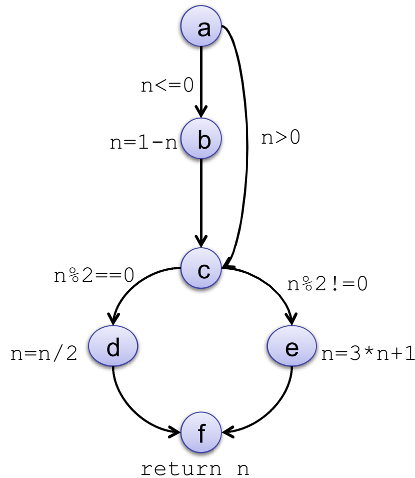
- Définir une donnée de test (DT) permettant de **sensibiliser** les chemins du CFG
  - Si un chemin est sensibilisé par une DT, il est dît **exécutable**
  - Si aucune DT ne permet de sensibiliser un chemin, il est dît **non exécutable**

Combien existe t'il de chemins pour une CFG ?

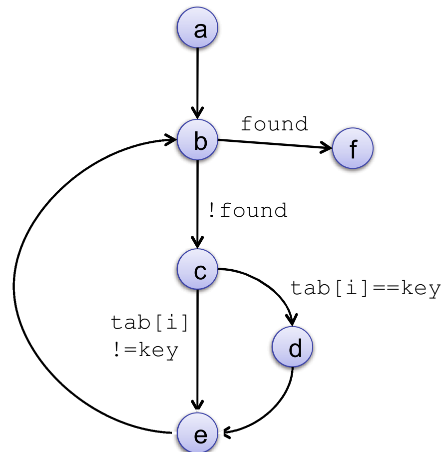
- Se déduit directement de l'expression algébrique du CFG
  - a(b + c)d(e + f)g → 1.(1+1).1.(1+1).1=4
  - Fournit le nombre de chemins exécutables et non exécutables



BB – Couverture du CFG



a (1 + b) c (e + d) f
soit 1.(1+1).1.(1+1).1 =4



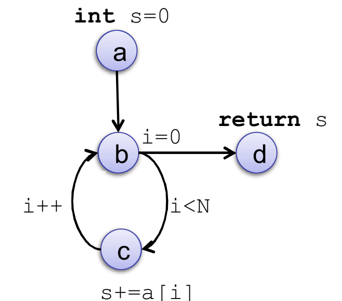
ab (c (1 + d) eb)* f
soit l'infini

BB – Couverture du CFG

```
int somme(int* a, int N){
 int s = 0;

 for (int i=0;i<N;i++)
  s+=a[i];

 return s;
}
```



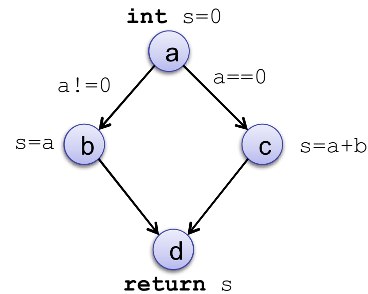
Expression des chemins :

$ab(cb)^N d$ → $1.1.(1.1)^N.1 = 1^N = N$

Niveau 1 – couverture de tous les nœuds

```
int somme(int a, int b){
 int s = 0;
 if (a==0)
   s = a;
 else
   s = a+b;
 return s;
}
```

Taux de couverture =
nb de nœuds couverts/nb total de nœuds



2 chemins de contrôle pour couvrir tous les nœuds

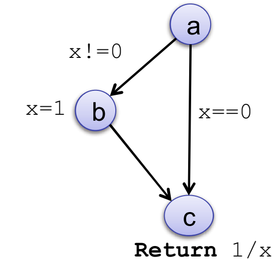
β_1 = abc

β_2 = acd

→ Détection de l'erreur avec le chemin β_1

Niveau 1 – couverture de tous les nœuds

```
int f(int x){
 if (x!=0)
   x = 1;
 return 1/x;
}
```



Le chemin de contrôle ***abc*** couvre tous les nœuds

→Sensibilisé par la DT = {x=2}

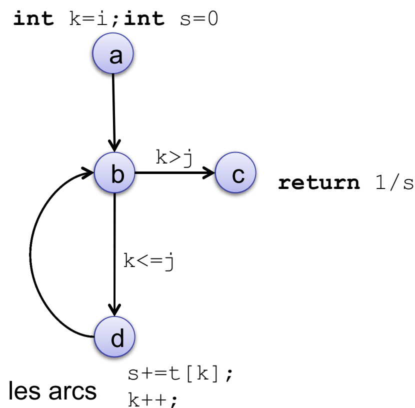
→ Mais pas de détection de l'erreur (division par 0)

Niveau 2 – couverture de tous les arcs

Calcul de l'inverse de la somme des éléments d'un tableau entre les indices **i** et **j**

Taux de couverture =
nb d'arcs couverts/nb total d'arcs

```
int somme(int* t, int i, int j)
{
   int k = i;
   int s=0;
   while(k<=j){
        s+=t[k];
        k++;
   }
   return 1/s;
}
```



La DT = {t = {1,2,3},i=0;j=2} permet de couvrir tous les arcs

Or, c'est la DT = {i=1,j=0} qui met le programme en erreur

BB – Couverture du CFG

- Remarque sur les couvertures de niveau 0, 1 et 2
  - N0 est impossible en présence de boucles : on ne peut pas tester tous les chemins distincts dans le cas d'une boucle avec un grand nombre (possiblement variable) d'itérations (chaque façon de terminer la boucle est un chemin distinct) ☹
    - *On couvrira plutôt tous les chemins indépendants ou toutes les portions linéaires de code suivies d'un saut*
  - N1 n'est pas suffisant : si une instruction `if` ne possède pas de branche `else`, on peut avoir exécuté toutes les instructions, mais sans avoir testé ce qui arrive lorsque la condition est fausse ☹
  - N2 implique N1 ☺

BB – Critère des chemins indépendants

- Définitions
  - Chemin : ensemble d'instructions de l'entrée à la sortie
  - Chemin indépendant : chemin introduisant un ensemble nouveau d'instructions ou le traitement d'une nouvelle condition (dans le cadre d'une suite de chemins que l'on teste)
- Principe du critère des chemins indépendants
  - Chaque chemin indépendant doit être parcouru au moins une fois
- Le critère *tous les chemins indépendants* vise à parcourir tous les arcs dans chaque configuration possible (et non pas au moins une fois comme dans le critère *tous les arcs*)
- Lorsque le critère *tous les chemins indépendants* est satisfait, cela implique :
  - le critère *tous les arcs* est satisfait
  - le critère *tous les nœuds* est satisfait
- Nombre minimal de cas de tests
  - Le nombre de chemins indépendants pour un graphe G est V(G) c'est-à-dire *Nb arcs – Nb nœuds + 2* ou *Nb conditions + 1* (If, While, for, case, and, …)

BB – Critère des chemins indépendants

- Procédure
  1. Evaluer le nombre minimal de cas tests à générer
  2. Produire une DT couvrant le maximum de nœuds de décisions du graphe
  3. Puisque qu'un chemin indépendant du précédent introduit un ensemble nouveau d'instructions ou le traitement d'une nouvelle condition, produire la DT qui modifie la valeur de vérité de la première instruction de décision de la dernière DT calculée.
  - Recommencer l'étape 3 jusqu'a la couverture de toutes les décisions.

BB – Critère des chemins indépendants

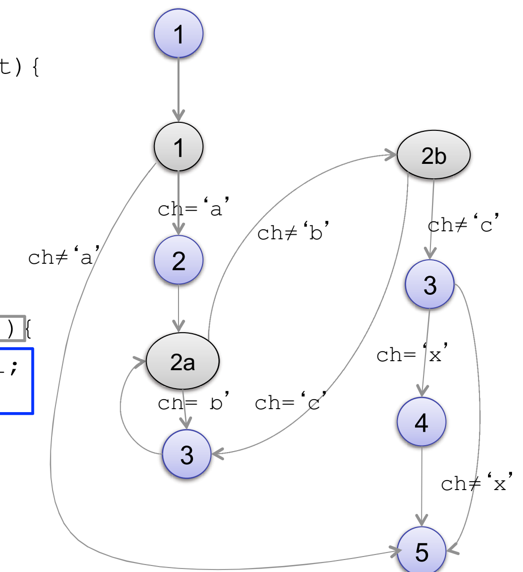
Exemple

```
bool goodstring(int count){
char ch;
bool good = false;
count := 0;
read(ch);
if ch = 'a' then {
   read(ch);
   while(ch=='b'||ch=='c'){
        count := count + 1;
        read(ch);
   }
   if ch = 'x'
        good = true;
}
return good;
```

BB – Critère des chemins indépendants

Exemple

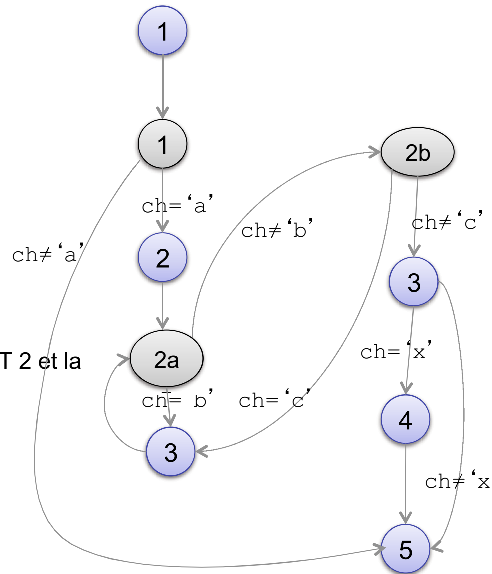
```
     bool goodstring(int count){
1    char ch;
     bool good = false;
     count := 0;
     read(ch);
1    if ch = 'a' then {
2       read(ch);
2 a-b   while(ch=='b'||ch=='c'){
3            count := count + 1;
             read(ch);
        }
3       if ch = 'x'
4            good = true;
     }
5 return good;
```



BB – Critère des chemins indépendants

Exemple

- Nombre minimal de DT
  - 12 – 9 + 2 = 5
- DT 1 = « abcx »
  - Couvre tous les nœuds
- DT 2 = « b »
  - Modifie la 1ère instruction de DT 1
- DT 3 = « acx »
  - Modifie la 1ère instruction de DT 2 et la 2ème de DT 1
- DT 4 = « ax »
- DT 5 = « aba »



BB – Exercice

```
int a=0, b=0, p=0;
read(a, b)
if (a<0)
   b = b-a;
if (b%2==0)
   p = b*b;
else
   p = 2*a;
println p;
```

1. Fournir le graphe de flot de contrôle
2. Donner des DTs pour tous les nœuds,
3. Donner des DTs pour tous les arcs,
4. Donner des DTs pour tous les chemins indépendants
5. Donner des DTs pour toutes les PLCS

Pour conclure

- Le test de logiciels est une activité très coûteuse qui gagne à être supportée par des outils.
- Les principales catégories d'outils concernent :
  - L'exécution des tests
  - La gestion des campagnes
  - Le test de performance
  - La génération de tests fonctionnels
  - La génération de tests structurels
- N'oubliez pas l'importance du test !!!