Définir les cas de tests

Types de tests

- Tests fonctionnels
  - Aussi appelés « boîte noire » (*black box tests*)
  - Identifient les fautes en se basant uniquement sur les entrées et les sorties
  - Ne tiennent pas compte du comportement interne du programme
    - Ex: chemin exécuté pour le calcul, cas spéciaux
- Tests structurels
  - Aussi appelés « boîte blanche » (*white box tests*) ou « boîte de verre » (*glass box tests*)
  - Exploitent la connaissance de la structure du code

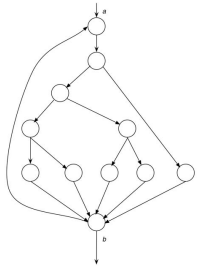
Tests exhaustifs

- Est-il possible de tester un programme **complèment**?
  - c'est-à-dire, trouver toutes les erreurs présentes
- Tests fonctionnels :
  - Pour tester le programme complètement, il faudrait tester toutes les entrées possibles
  - Une infinité de cas en général

Tests exhaustifs

- Tests structurels
  - Programme de ~ 50-100 lignes
  - 1 boucles (1-20 itérations), 4 conditions imbriquées à l'intérieur de la boucle

```
do
   if (…) then
       if (…) then
          if (…) then
          else …
       else …
          if (…) then
          else …
   else …
while (i < 20)
```



~ 10^{14} chemins d'exécution possibles ($5^1 + 5^2 + \ldots + 5^{20}$)
3174 ans pour tester à raison d'un chemin par milliseconde!

Tests exhaustifs

- En général, tester un programme de façon exhaustive est impossible
- Il faut choisir un sous-ensemble des tests qui maximise la probabilité de détecter les erreurs
- Approche couramment utilisée : déterminer un ensemble de tests fonctionnels qui seront complémentés de tests structurels

Test fonctionnels

Test fonctionnels

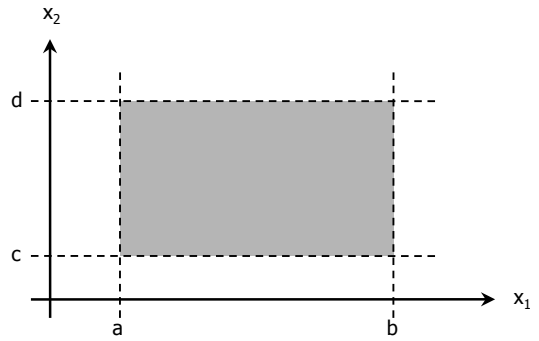
- L'identification de tests fonctionnels repose sur l'identification de valeurs à utiliser dans le domaine des entrées du programme
- Le SST peut être vu comme une fonction

$$F : D \rightarrow I$$

  - Les entrées de F possèdent souvent des contraintes (possiblement implicites) :

$$a \leq x_1 \leq b \, , \, c \leq x_2 \leq d$$

Domaine d'une fonction à 2 variables



Tests par valeurs de marges

- Pour chaque variable d'entrée, choisir 5 valeurs :
  - Minimum
  - Juste au dessus du minimum
  - Nominale
  - Just en dessous du maximum
  - Maximum

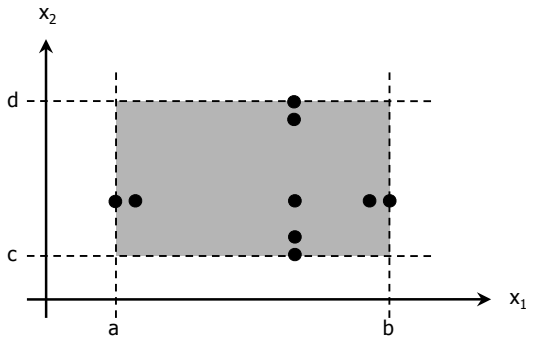
Hypothèse de faute unique

- En pratique, une défaillance est rarement causée par par plus d'une faute
- Sous cette hypothèse, les tests sont générés en gardant toutes les variables sauf une à leur valeur nominale, et en laissant cette variable prendre toutes ses valeurs extrêmes
  - En général, $4n + 1$ tests pour n variables

Valeurs de marges - 2 variables

- $<x_{1nom}, x_{2min}>$
- $<x_{1nom}, x_{2min+}>$
- $<x_{1nom}, x_{2nom}>$
- $<x_{1nom}, x_{2max-}>$
- $<x_{1nom}, x_{2max}>$
- $<x_{1min}, x_{2nom}>$
- $<x_{1min+}, x_{2nom}>$
- $<x_{1nom}, x_{2nom}>$
- $<x_{1max-}, x_{2nom}>$
- $<x_{1max}, x_{2nom}>$

Valeurs de marges



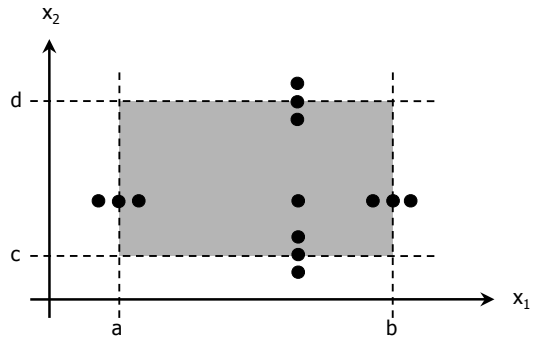
Limites

- Présume que les variables sont indépendantes
  - Par exemple, pour un programme qui manipule des dates (3 variables - j,m,a):
    - Les erreurs reliées au début du mois et de fin de mois seront probablement détectées
    - Les tests ne seront pas adéquats pour les cas intéressants du mois de février ou des années bissextiles, pour lesquelles les différentes variables interagissent
- Souvent utile pour les quantités physiques
  - Marges utiles pour une température ou une altitude
  - Marges peu utiles pour un NIP ou # de téléphone

Variation - Robustesse

- Ajoute des valeurs légèrement à l'extérieur des intervalles
  - Un peu en dessous du minimum (min-)
  - Un peu au dessus du maximum (max+)
- Permet de tester le traitement des exceptions

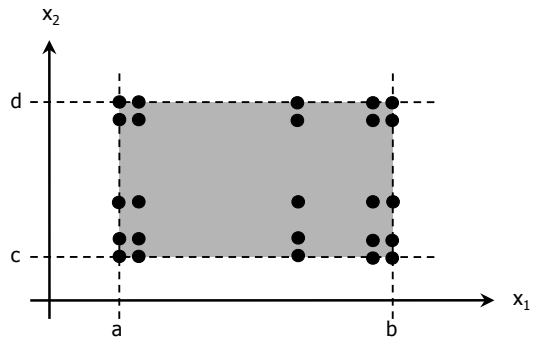
Variation - Robustesse



Variation - Pire cas

- Rejette l'hypothèse de la faute unique
  - Permet de détecter les erreurs dues à plusieurs fautes simultanées
- Utilise le produit cartésien des valeurs de chacune des variables
  - En général, 5^n tests plutôt que $4n + 1$

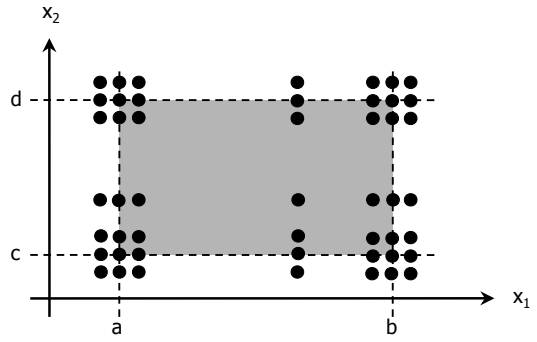
Variation - Pire cas



Variation - Pire cas robuste

- Ajoute la robustesse à l'analyse des pires cas
  - 2 valeurs ajoutées pour chaque variable
  - En général, 7^n tests plutôt que 5^n ou $4n + 1$

Variation - Pire cas robuste



Variation : valeurs spéciales

- En pratique, les tests des pires cas produisent une grande quantité de tests dont l'efficacité à identifier des bogues est relativement faible
- Alternative : utiliser les connaissances du domaine pour déterminer des cas à tester
  - p.ex.: 28 février, 29 février, années bissextiles

Triangles - Valeurs de marge

Test	a	b	c	Résultat
M1	100	100	1	Isocèle
M2	100	100	2	Isocèle
M3	100	100	100	Équilatéral
M4	100	100	199	Isocèle
M5	100	100	200	Pas un triangle
M6	100	1	100	Isocèle
M7	100	2	100	Isocèle
M8	100	199	100	Isocèle
M9	100	200	100	Pas un triangle
M10	1	100	100	Isocèle
M11	2	100	100	Isocèle
M12	199	100	100	Isocèle
M13	200	100	100	Pas un triangle

Triangles - Pire cas

Test	a	b	c	Résultat
WC1	1	1	1	Équilatéral
WC2	1	1	2	Pas un triangle
WC3	1	1	100	Pas un triangle
WC4	1	1	199	Pas un triangle
WC5	1	1	200	Pas un triangle
WC6	1	2	1	Pas un triangle
WC7	1	2	2	Isocèle
WC8	1	2	100	Pas un triangle
WC9	1	2	199	Pas un triangle
WC10	1	2	200	Pas un triangle
WC11	1	100	1	Pas un triangle
WC12	1	100	2	Pas un triangle
WC13	1	100	100	Isocèle
...				

Tests par partionnement

- Les valeurs de marges produisent souvent des tests
  - redondants
  - incomplets
- Alternative : partionner les entrées en **classes d'équivalences**
  - sous-ensembles disjoints dont l'union est l'ensemble complet
  - générer un test par classe d'équivalence
- Implications
  - Complétude (union est l'ensemble complet)
  - Non-redondance (ensembles disjoints)

Sélection des classes d'équivalence

- Les tests qui appartiennent à la même classe d'équivalence devraient identifier les mêmes erreurs
  - la redondance entre les tests peut être grandement réduite
- Comment identifier les classes d'équivalence appropriées pour chaque variable ?
  - dépend du domaine

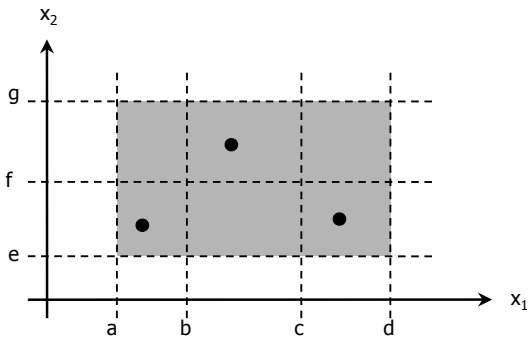
Exemples de classes d'équivalences

- Plage de valeurs (ex: 1..100)
- Nombre de valeurs
  - ex: au moins 3 valeurs spécifiées
- Ensemble de valeurs prédéfinies
  - ex: « auto », « maison », « bateau »
- « Doit être »
  - ex: le 1er caractère doit être une lettre

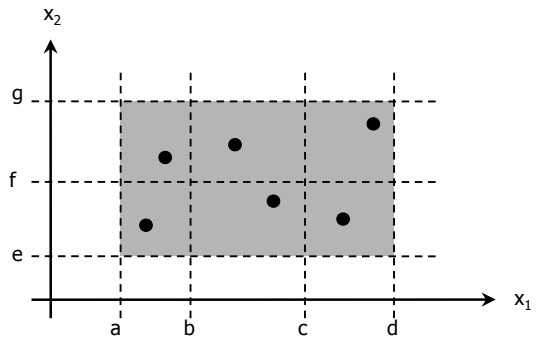
Génération de tests

- Générer des tests à partir de classes d'équivalence peut utiliser deux stratégies:
  - **Équivalence faible** : chaque test utilise une valeur de chacune des classes d'équivalences valides
    - Utilise l'hypothèse de faute unique
    - Nombre de tests égal au nombre maximal de partitions pour toutes les variables
  - **Équivalence forte** : un test par combinaison de classes d'équivalences valides pour toutes les variables
    - Rejette l'hypothèse de faute unique
    - Nombre de tests égal au produit du nombre de partitions pour toutes les variables

Partionnement - Équivalence faible



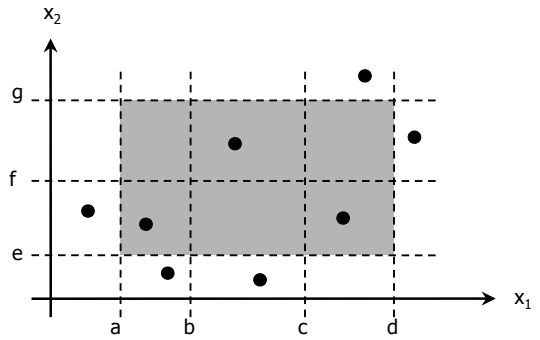
Partionnement - Équivalence forte



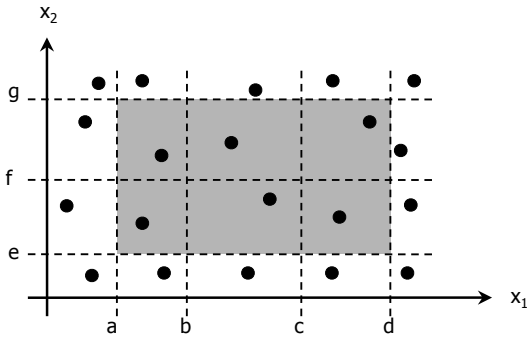
Robustesse

- Le concept de robustesse peut être combiné au partionnement en classes d'équivalence
  - Les tests utilisent aussi les classes d'équivalences invalides
- Équivalence faible & robuste :
  - Pour les entrées invalides, des valeurs valides sauf une valeur qui sera invalide
- Équivalence forte & robuste :
  - Inclure les classes d'équivalence invalides dans le produit cartésien des valeurs possibles

Partionnement - Équival. faible robuste



Partionnement - Équiv. forte robuste



Triangles - Partitions

Test	a	b	c	Résultat
WN1	5	5	5	Équilatéral
WN2	2	2	3	Isocèle
WN3	3	4	5	Scalène
WN4	4	1	2	Pas un triangle
WR1	-1	5	5	a invalide
WR2	5	-1	5	b invalide
WR3	5	5	-1	c invalide
WR4	201	5	5	a invalide
WR5	5	201	5	b invalide
WR6	5	5	201	c invalide

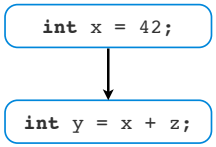
Test structurels

Graphe de flot de contrôle

- Un graphe de programme est un graphe dirigé où
  - les noeuds représentent les instructions
  - les arcs représentent le flot de contrôle possible entre les instructions
    - arc $s_i \rightarrow s_j$ implique que s_j peut s'exécuter directement à la suite de s_i

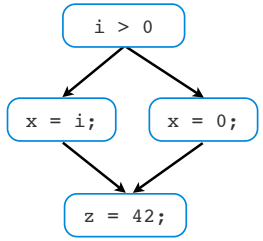
Graphe de flot de contrôle


```
int x = 42;
int y = x + z;
```



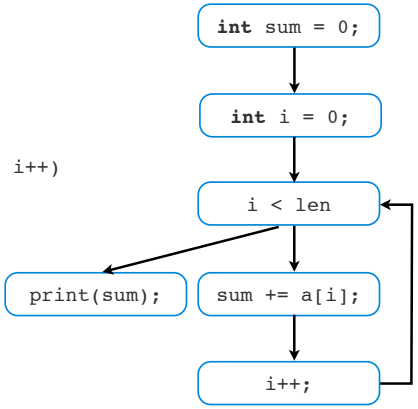
Graphe de flot de contrôle


```
if (i > 0) {
    x = i;
} else {
    x = 0;
}
z = 42;
```



Graphe de flot de contrôle


```
int sum = 0;
for (int i = 0; i < len; i++)
{
    sum += a[i];
}
print(sum);
```

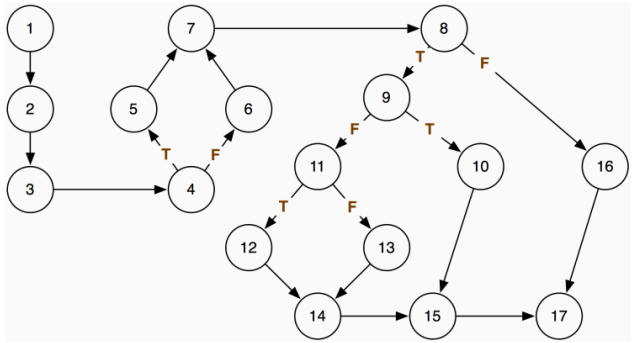


Exemple - Triangles


```
1 output("Enter 3 integers:")
2 input(a, b, c)
3 output("Side a b c: ", a, b, c)
4 if (a < b + c) and (b < a+c) and (c < a+b)
5 then isTriangle ← true
6 else isTriangle ← false
7 fi

8 if isTriangle
9 then if (a = b) and (b = c)
10      then output("equilateral")
11      else if (a ≠ b) and (a ≠ c) and (b ≠ c)
12           then output("scalene")
13           else output("isosceles")
14           fi
15      fi
16 else output("not a triangle")
17 fi
```


Triangles - Graphe de flot de contrôle



Chemins d'exécution

- Chaque exécution valide d'un programme correspond à un chemin valide dans son graphe de flot de contrôle
  - Débute à l'entrée
  - Se termine à la sortie

Des chemins aux tests

- Un test peut être généré à partir d'un chemin d'exécution qu'on désire emprunter
  - L'ensemble des conditions évaluées détermine des **contraintes** pour chacune des variables
  - Ces contraintes peuvent être résolues (souvent automatiquement) afin de déterminer un ensemble de valeurs

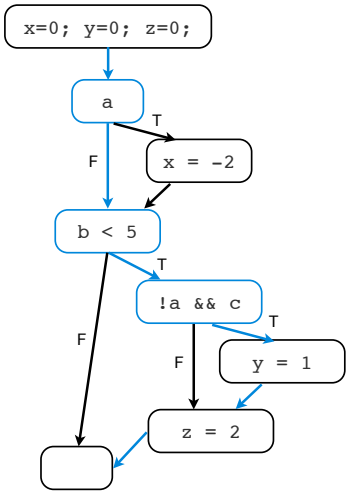
Des chemins aux tests


```
function f(a, b, c) {
    var x=0, y=0, z=0;

    if (a)
        x = -2;

    if (b < 5)
        if (!a && c)
            y = 1;
        z = 2;
}
```

b < 5 && !a && c, p. ex.
a = 0, b = 0, c = 1



Couverture

- La **couverture** est une mesure d'exhaustivité des tests effectués
  - Idéalement, la couverture devrait être exprimée en fonction de tous les chemins d'exécution possibles
  - En pratique, des approximations sont utilisées
- Un ensemble de tests devraient toujours viser la couverture complète d'un programme
  - En pratique, il peut être impossible de couvrir tout le programme d'après une certaine approximation
  - Maximiser la couverture maximise augmente la probabilité de détecter les erreurs

Couverture des instructions

- Nécessite que chaque instruction du programme soit exécutée au moins une fois
- Un critère de couverture assez faible :

```
public int foo(int x, int y) {
    if (x != 0 && y != 0)
        y=x+1;
    return y;
}
```

  - Si un test appelle `foo(1,1)`, toutes les instructions sont exécutées, mais le cas où y est retourné sans modification n'est pas testé.
  - La condition (`x!=0`) peut être erronée sans que les tests ne le démontrent.

Couverture des décisions / branches

- Nécessite que chaque décision prenne chaque valeur possible au moins un fois (en général vrai ou faux)
  - Les conditions incluent les boucles (while, for, do-while, etc.), les instructions conditionnelles (if-else, switch)
  - Implique la couverture des instructions

```
public int foo(int x, int y) {
    if (x != 0 && y != 0)
        y=x+1;
    return y;
}
```

  - Il faudrait écrire au moins 2 tests, par exemple :
    - `foo(0,1), foo(1,1)`

Couverture des conditions

- Nécessite que chaque condition qui fait partie d'une décision prenne chaque valeur possible au moins un fois (en général vrai ou faux)

```
public int foo(int x, int y) {
    if (x != 0 && y != 0)
        y=x+1;
    return y;
}
```

  - Il faudrait écrire au moins 2 tests, par exemple :
    - `foo(0,0), foo(1,1)`

Couverture des conditions

- La couverture de conditions n'implique pas la couverture de branches :

```
public int foo(int x, int y) {
    if (x != 0 && y != 0)
        y = x+1;
    return y;
}
```

  - Considérez les tests suivants :
    - `foo(0,1), foo(1,0)`
  - Chaque condition prend les deux valeurs possibles, mais l'instruction `y = x+1` n'est jamais exécutée

Couverture des branches/conditions

- Chaque décision **et** chaque condition d'une décision doivent prendre toutes les valeurs possibles
  - Combinaisons des deux approches précédentes
- Certaines valeurs peuvent en masquer d'autres

```
public int foo(int x, int y, int z) {
    if (x != 0 && (y != 0 || z < 0))
        y=x+1;
    return y;
}
```

  - Le test `foo(0,1,1)` aurait pu exposer le fait que la condition `z<0` est erronée, mais comme la condition `x!=0` masque le reste de l'expression, l'erreur n'est pas détectée.

Couverture de conditions multiples

- Toutes les combinaisons de conditions dans toutes les décision doivent être couvertes au moins une fois :
  - Implique toutes les approches précédentes

```
public int foo(int x, int y) {
    if (x != 0 && y != 0)
        y = x+1;
    return y;
}
```

  - Il faudrait écrire au moins 4 tests, par exemple :
    - `foo(0,0), foo(1,0), foo(0,1), foo(1,1)`

Boucles

- Les boucles sont souvent la source de fautes, il faut donc bien les tester
  - Tester la condition d'entrée de la boucle
  - Utiliser des valeurs de marges pour la variable d'induction
  - Tester les boucles imbriquées de l'intérieur vers l'extérieur

Triangles - Couverture des chemins

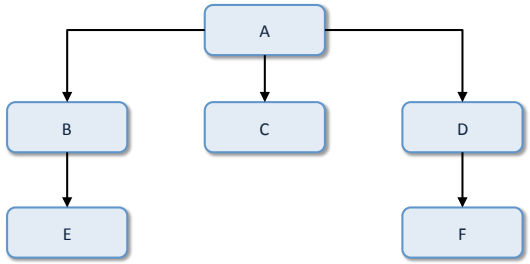
Test	Chemin	a	b	c	Résultat
P1	1-4,5,7-8,16-17				Impossible
P2	1-4,6,7-8,16-17	4	1	2	Pas un triangle
P3	1-4,5,7-8,9,10,15,17	5	5	5	Équilatéral
P4	1-4,6,7-8,9,10,15,17				Impossible
P5	1-4,5,7-8,9,11,13-15,17	2	2	3	Isocèle
P6	1-4,6,7-8,9,11,13-15,17				Impossible
P7	1-4,5,7-8,9,11,12,14,15,17	3	4	5	Scalène
P8	1-4,6,7-8,9,11,12,14,15,17				Impossible

Tests d'intégration

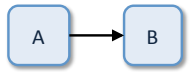
Stratégies de test

- Non-incrémental (« Big bang »)
  - chaque module est testé indépendamment, puis les modules sont assemblés pour former le système complet.
- Incrémental:
  - le prochain module est intégré aux modules testés existants avant d'être lui-même testé

Exemple - Test non-incrémental



- Comment gérer les dépendances entre les tests?
  - A dépend de B



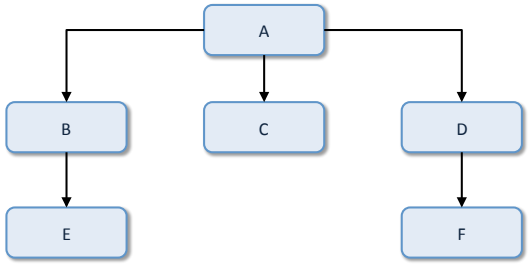
Pilotes & modules souches

- Chaque module à tester nécessite
  - un pilote (*driver*), qui effectue les tests
  - Les modules A, B et D nécessite des modules souches (stubs) qui jouent le rôle de leurs dépendances lors des tests

Pilotes & modules souches

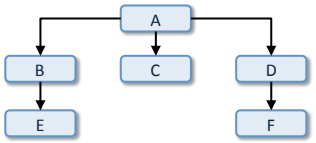
- Pilotes
  - Lancent l'exécution des tests pour un module particulier
  - Une alternative consiste à utiliser xUnit
- Modules souches
  - Jouent le rôle d'un autre module durant l'exécution des tests
  - Maintiennent un minimum d'état pour simuler le comportement réel d'un module dans le cadre d'un test précis
  - Sont généralement coûteux à développer

Exemple - Test incrémental



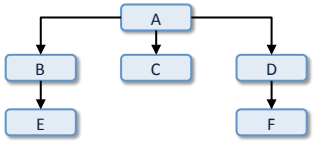
- Comment choisir l'ordre d'exécution des tests?

Ordonnancement des tests



- Une option possible
  - Tester E, C, F en parallèle
  - Tester B avec E, D avec F
  - Tester A avec B, C et D
- Remarques
  - La création de modules souches n'est plus nécessaire
  - Les pilotes doivent être créés pour chaque module testé

Ordonnancement des tests



- Une alternative
  - Tester A
  - Tester B, C et D avec A
  - Tester E avec B, F avec D
- Remarques
  - La création de pilotes n'est plus nécessaire
  - Les modules souches doivent être créés pour chaque module testé

Observations

- Le test incrémental
  - est moins coûteux que le test non-incrémental
    - nécessite des pilotes ou des modules souches, mais pas les deux à la fois
  - permet d'identifier les erreurs de communication entre modules plus rapidement
  - facilite le débogage
  - teste les modules de façon plus approfondie
    - Un vrai module est utilisé à la place d'un pilote ou d'un module souche, et donc reçoit plus d'attention lors de tests
  - Réduit la possibilité d'exécuter des tests en parallèle

Test incrémental - Direction

- Haut en bas
  - Permet de démontrer les fonctions du système plus rapidement, et donc de faciliter la validation
  - Permet de détecter plus efficacement les erreurs structurelles dans un système
  - Lorsque les fonctions d'entrée / sortie sont ajoutées
    - de vraies données peuvent être utilisées comme tests
    - l'observation des résultats de tests est généralement simplifiée

Test incrémental - Direction

- Bas en haut
  - Retarde la vérification et la validation du système en entier
    - Le système n'existe que lorsque le dernier module est ajouté
  - Implémentation et observation des tests beaucoup plus facile

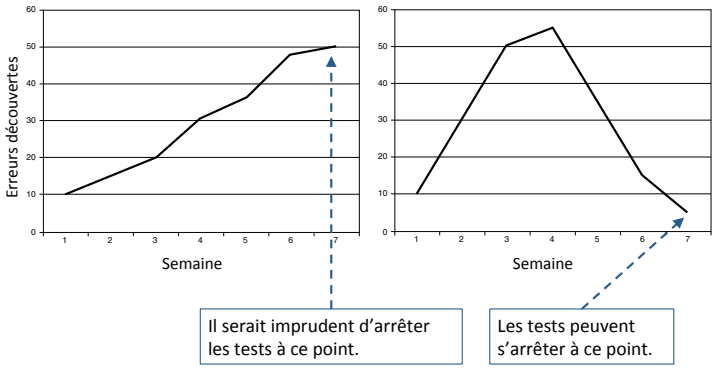
Critères d'arrêt

Quand arrêter de tester?

- Arrêter lorsque le temps réservé aux tests est écoulé
  - Ce critère peut être satisfait même si aucun test n'est effectué!
- Arrêter lorsque les tests ne révèlent plus d'erreurs
  - Aussi indépendant de la qualité des tests
- Arrêter lorsqu'une méthodologie a été suivie pour la création des tests et que les tests ne révèlent plus d'erreurs
  - Assure un certain niveau de qualité des tests
  - Ne convient pas aux tests pour lesquels une méthodologie n'est pas disponible (ex: tests de système)
  - Est subjectif et difficile à appliquer en pratique

Quand arrêter de tester?

- Arrêter lorsque la fréquence de découverte de nouvelles erreurs est en baisse



Quand arrêter de tester?

- Arrêter lorsqu'un nombre d'erreurs ont été trouvées
  - Ex: le programme sera testé jusqu'à ce que 90% des erreurs ait été trouvées, ou 3 mois, selon le dernier critère atteint
  - Nécessite un estimé du nombre d'erreurs dans le système
  - Quoi faire si l'estimé surestime le nombre d'erreurs?
    - Il se peut que le critère voulu ne soit jamais atteint
    - Il est difficile de déterminer si les erreurs n'existent pas dans le code, ou si les tests ne les ont simplement pas encore découverts
    - Une agence de test externe (indépendante) peut aider à faire un choix éclairé

Estimation du nombre d'erreurs

- Par comparaison avec des projets antérieurs
- Par étude des premières phases de tests
  - Un modèle statistique peut prédire le nombre d'erreurs en fonction de données initiales du projet
- Par utilisation de moyennes empiriques
  - Ex: 4 à 8 erreurs en moyenne par 100 lignes de code
- Par introduction d'erreurs simulées
  - Des erreurs sont introduites dans le programme, et les tests sont effectués pour savoir combien de ces erreurs sont identifiées
- Par comparaison de deux équipes indépendantes
  - Deux équipes indépendantes tentent d'identifier les erreurs en parallèle
  - Les erreurs identifiées par les deux équipes sont utilisées pour estimer le nombre total d'erreurs