Test d'intégration

Plan

- Test d'intégration :
  - stratégies,
  - critères.
- Conclusions :
  - génération des données de tests, outils.

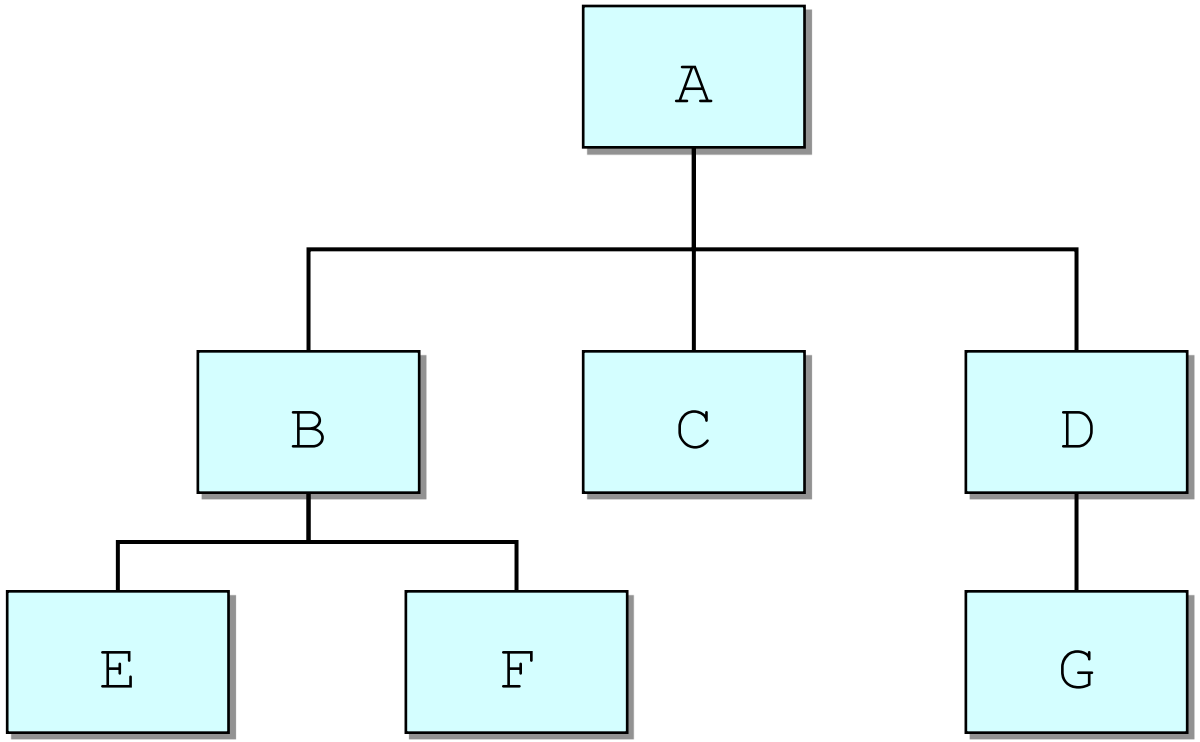
Remarques préliminaires I

- Une fois que les composants ont été testés à un niveau satisfaisant, il faut les combiner avec des systèmes opérationnels.
- Les composants sont probablement encore défectueux puisque les 'stubs' et les pilotes utilisés durant les tests unitaires sont seulement des approximations des composants qu'ils simulent.
- En plus, des anomalies au niveau des interfaces sont possibles.
  - e.g., suppositions concernant la sémantique des paramètres.
- L'ordre d'intégration ? Les pilotes et les 'stubs' sont coûteux et l'ordre affecte le coût du test.

Remarques préliminaires II

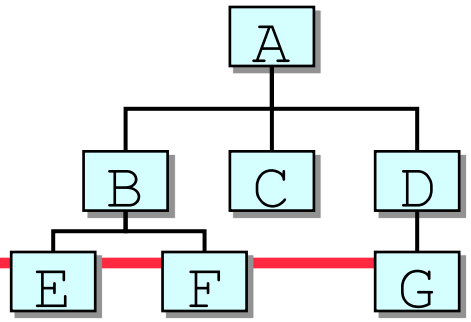
- L'intégration doit être planifiée de telle manière que lorsqu'une défaillance se produit, nous ayons une idée des causes probables
- Le développement d'un logiciel n'est pas séquentiel. Les activités se chevauchent : quelques composants peuvent être en cours de codage, d'autres en cours de test unitaire et d'autres en cours du test d'intégration.
- La stratégie d'intégration affecte l'ordre du codage, du test unitaire, du coût et de la minutie du test.

Exemple générique



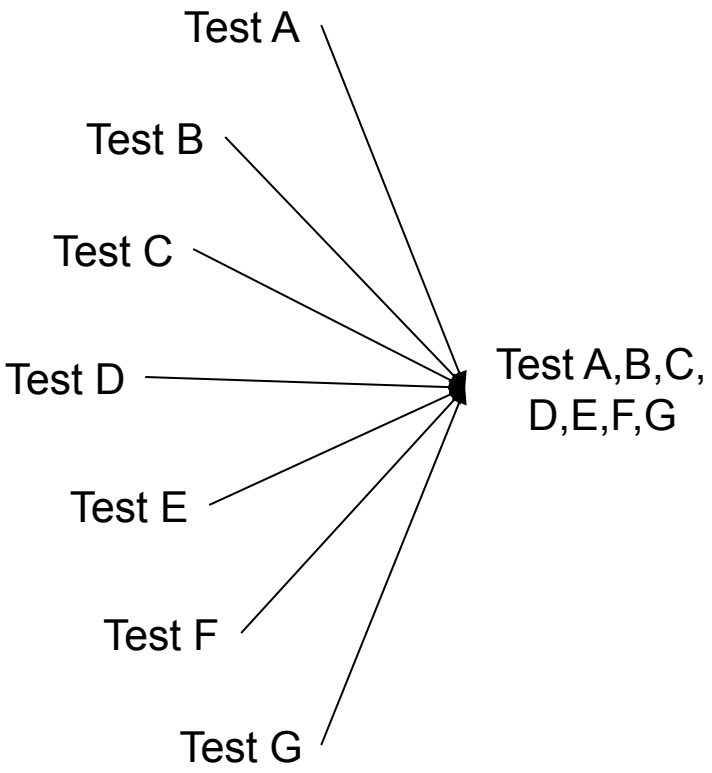
Exemple d'une hiérarchie de modules, définie par l'usage de dépendances entre les modules.

Intégration du Big Bang

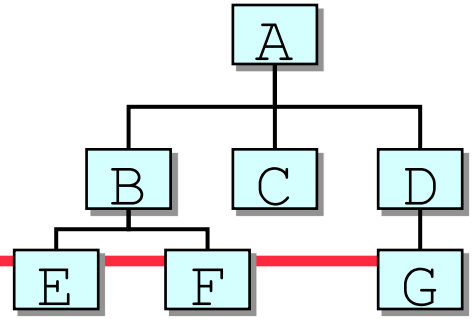


- Tous les composants sont introduits ensemble immédiatement dans un système et testés comme un système entier.
- OK pour des systèmes petits et bien structurés
- OK si on fait confiance aux composants
- ☹ Difficulté à trouver la cause principale en cas de défaillances.
- ☹ Attendre que tous les composants soit prêts.

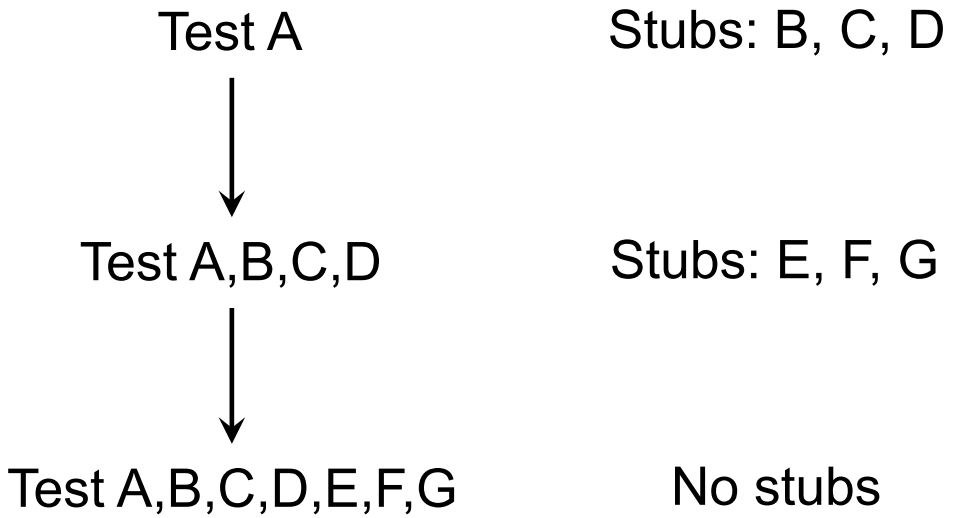
=> Pas recommandé en général.



Intégration descendante (de haut en bas) I



- Le système est testé niveau par niveau, en commençant par le plus haut niveau de l'arbre de dépendance
- Quand des composants qui n'ont pas encore été testés sont nécessaires, on utilise des "stubs" pour simuler leur(s) activité(s).

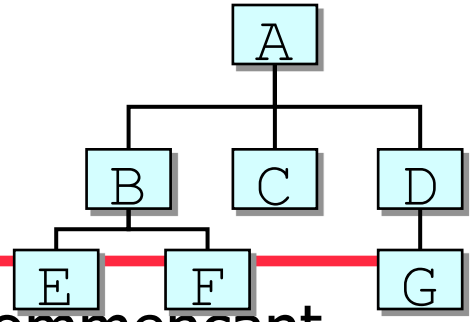


Intégration descendante (de haut en bas) II

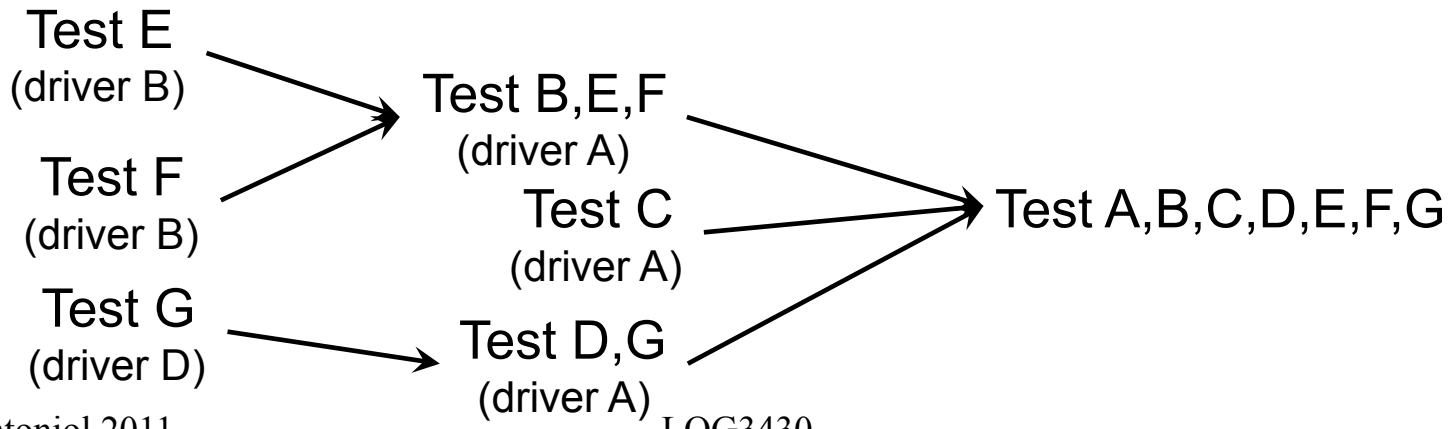
- ☺ En cas de bug, la zone suspecte est limitée
- ☺ Le test peut commencer assez tôt: dès que les composants de haut niveau ont été codés.
- ☺ On peut procéder en avance à une validation de la fonctionnalité principale
- ☺ Les composants peuvent être développés en parallèle.

- ☹ Des problèmes de performance peuvent apparaître tard dans le processus (du fait que les modules les plus bas soient testés en dernier).
- ☹ Écrire les (possiblement très nombreux) stubs peut s'avérer complexe et long, présente des risques accrus d'erreurs (lesquelles peuvent affecter la validité du test).
- ☹ Inadéquat si les spécifications des composants du niveau inférieur sont instables.

Intégration ascendante (de bas en haut) I



- Le système est testé niveau par niveau, en commençant par le plus bas niveau (les feuilles de l'arbre)
- Chacun des composants au niveau le plus bas de la hiérarchie est testé en premier (e.g., E, F et G).
- Si un problème se produit pendant le test B (avec E et F) → la cause est soit dans B, ou dans l'interface entre B et E ou entre B et F.

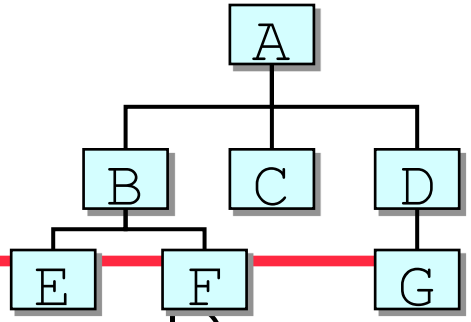


Intégration ascendante (de bas en haut) II

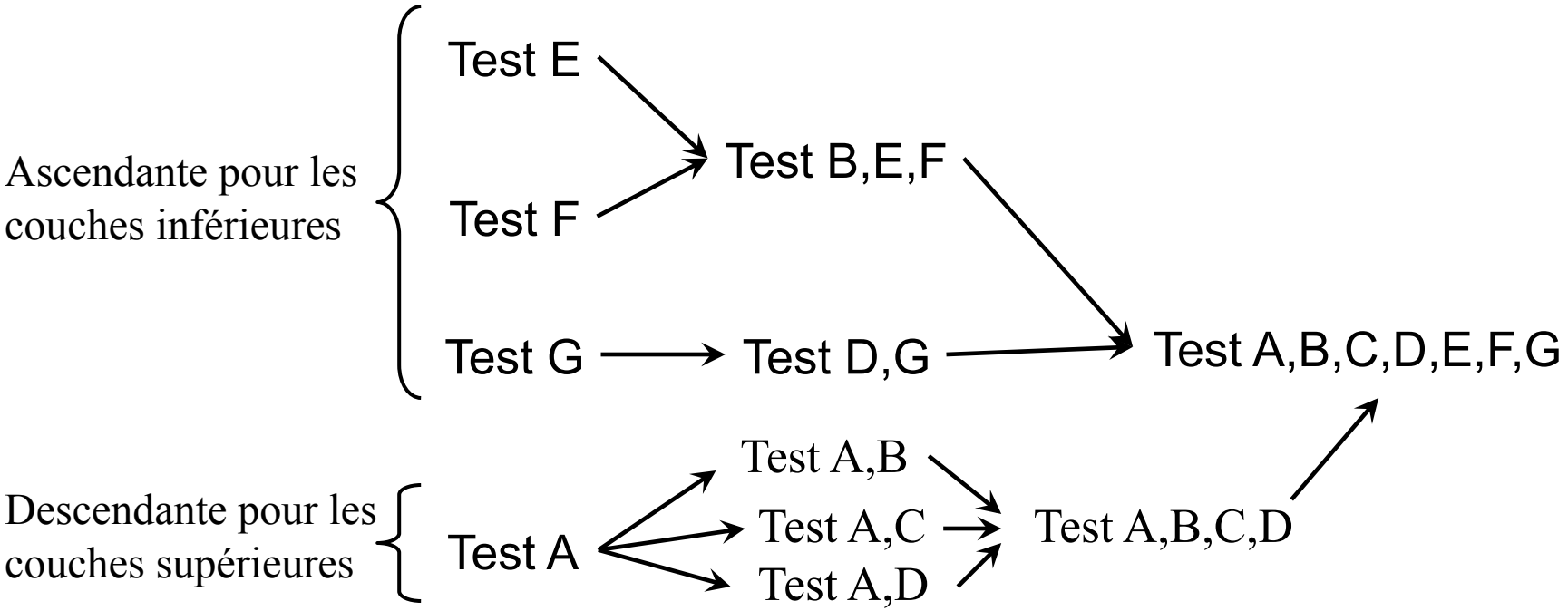
- ☺ Besoin de développer les composants pilotes mais pas les *stubs.*
- ☺ Les anomalies sont plus facilement isolées, en particulier, celles d'interface.
- ☺ Le test peut commencer tôt: dès qu'un composant "feuille" est prêt.
- ☺ Initialement, les tests peuvent être effectués en parallèle
- ☺ Favorable quand on a beaucoup de routines utilitaires, invoquées souvent par d'autres, aux niveaux les plus bas.
- ☺ Validation en avance des composants de performance critique.

- ☹ Problème : les composants au niveau supérieur peuvent être plus importants (activités majeures du système) mais sont les derniers à être testés, e.g., composants d'interface usager.
- ☹ Peut requérir un nombre important de tests.
- ☹ Les anomalies du niveau supérieur reflètent parfois des défauts de conception (décomposition en sous-système) et doivent être corrigés au plus tôt.

Intégration Sandwich I



1. Choisir une couche cible (celle du milieu dans notre exemple).
2. L'approche descendante est utilisée dans les couches supérieures.
3. L'approche ascendante dans les couches inférieures.



Intégration Sandwich II

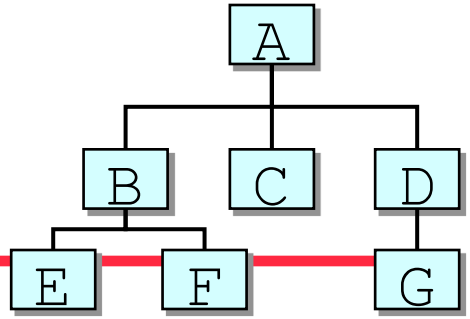
- ☺ Permet de tester l'exactitude des modules utilitaires dès le départ.
- ☺ Pas besoin de stubs pour les utilitaires.
- ☺ Permet de commencer le test d'intégration des composants supérieurs en avance: tests "contrôle" du composant faits plus tôt.
- ☺ Pas besoin des pilotes de test des couches supérieures.
- ☹ Mais peut ne pas tester minutieusement les composants individuels de la couche cible (e.g., C dans l'exemple n'est pas testé).

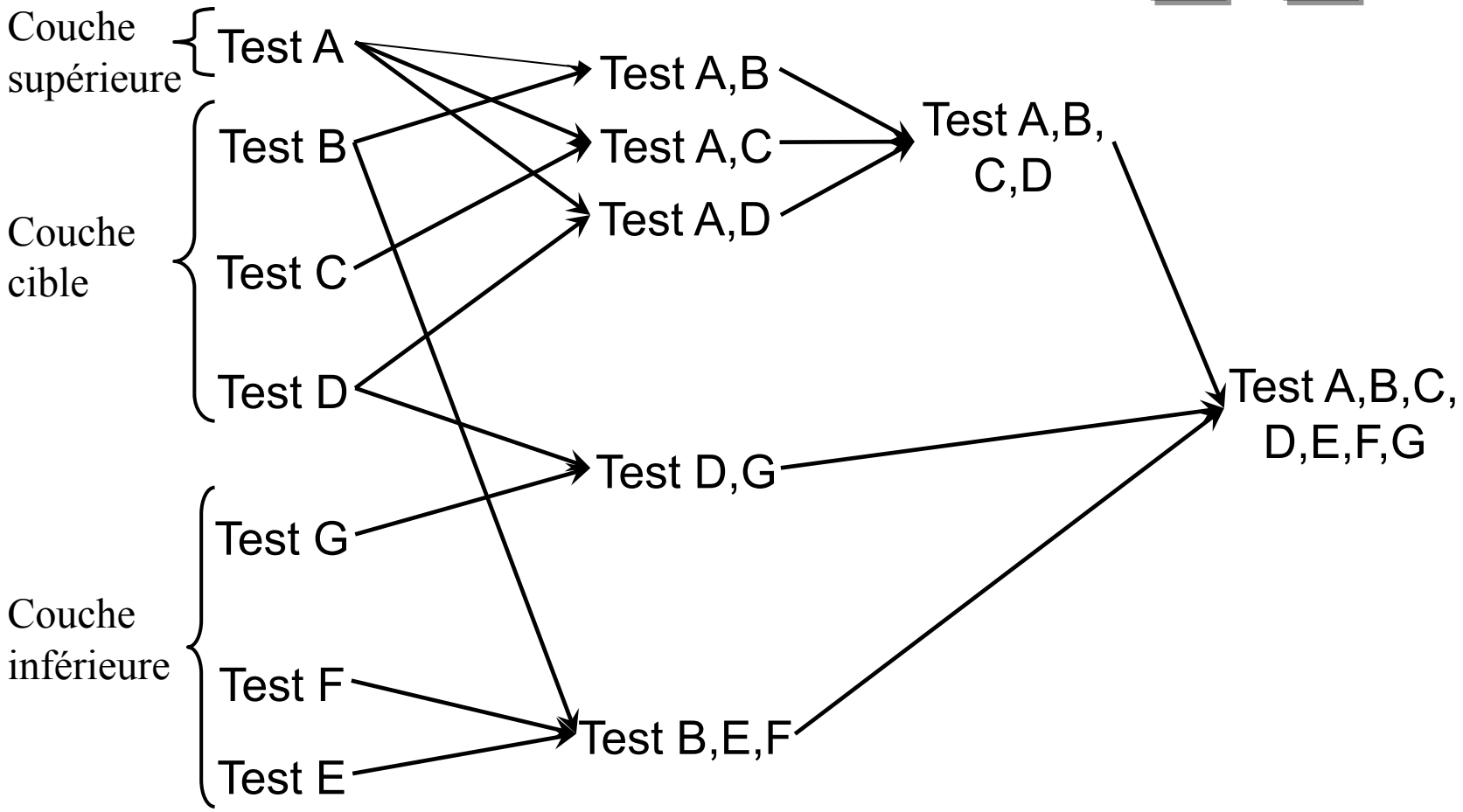
Test Sandwich modifié I

1. Tests de couches individuelles
   - Un test des couches supérieures avec des stubs en remplacement de la couche cible.
   - Un test de la couche cible avec des pilotes et stubs remplaçant respectivement les couches supérieure et inférieure.
   - Un test des couches inférieures avec des pilotes en remplacement de la couche cible.
2. Tests de couches combinées
   - La couche supérieure accède à la couche cible.
   - La couche inférieure est accédée par la couche cible.

- Plusieurs activités de tests peuvent être effectuées en parallèle –temps de test plus court .
- Mais des stubs et pilotes additionnels.

Test Sandwich modifié II





Critère de comparaison

- Les stubs, les pilotes
- Délai de livraison au marché
  - Le système que vous pouvez démontrer (e.g., pour usagers futurs)
- Date de début de l'intégration : nous le voulons le plus tôt possible.
- Capacité à tester les chemins :
  - Garantit la couverture de chemin des composants.
- Capacité de planifier et contrôler :
  - L'approche descendante est compliquée à cause de l'instabilité inhérente des composants du niveau inférieur, qui peut invalider le test d'intégration des composants supérieurs si leur spécification change. Rappelons qu'ils peuvent ne pas exister encore quand le test d'intégration des composants du niveau supérieur commence. L'approche Sandwich hérite du même probleme pour la même raison.

Stratégies d'intégration - revue

	Ascendant	Descendant	Big-Bang	Sandwich
Intégration Commence	Tôt	Tôt	Tard	Tôt
Délai de livraison du système	Tard	Tôt	Tard	Tôt
pilotes	Oui	Non	Non	Oui
stubs	Non	Oui	Non	Oui
Capacité de tester les chemins	Facile	Difficile	Facile	Moyenne
Capacité de planifier et contrôler	Facile	Difficile	Facile	Difficile

Plan

- Test d'intégration :
  - Stratégies,
  - critères.

- Conclusions :
  - génération des données de test, outils

Critère du couplage de base

- Réfère au test d'interfaces entre unités / modules pour s'assurer qu'ils ont des hypothèses cohérentes et communiquent correctement.
  - Le couplage entre 2 unités mesure les relations de dépendance entre 2 unités en reflétant les interconnections entre les unités ; les anomalies dans une unité peuvent affecter l'unité couplée. (Yourdon et Constantine, 1979).
  - Le critère de la couverture basée sur le couplage est proposé par Jin et Offutt (98): il vise spécifiquement le test d'intégration dans un contexte non-OO et est basé sur l'analyse du flot de données.

Z. Jin and J. Offutt. *Coupling-based Criteria for Integration Testing*. Journal of Software Testing, Verification, and Reliability, 1999.

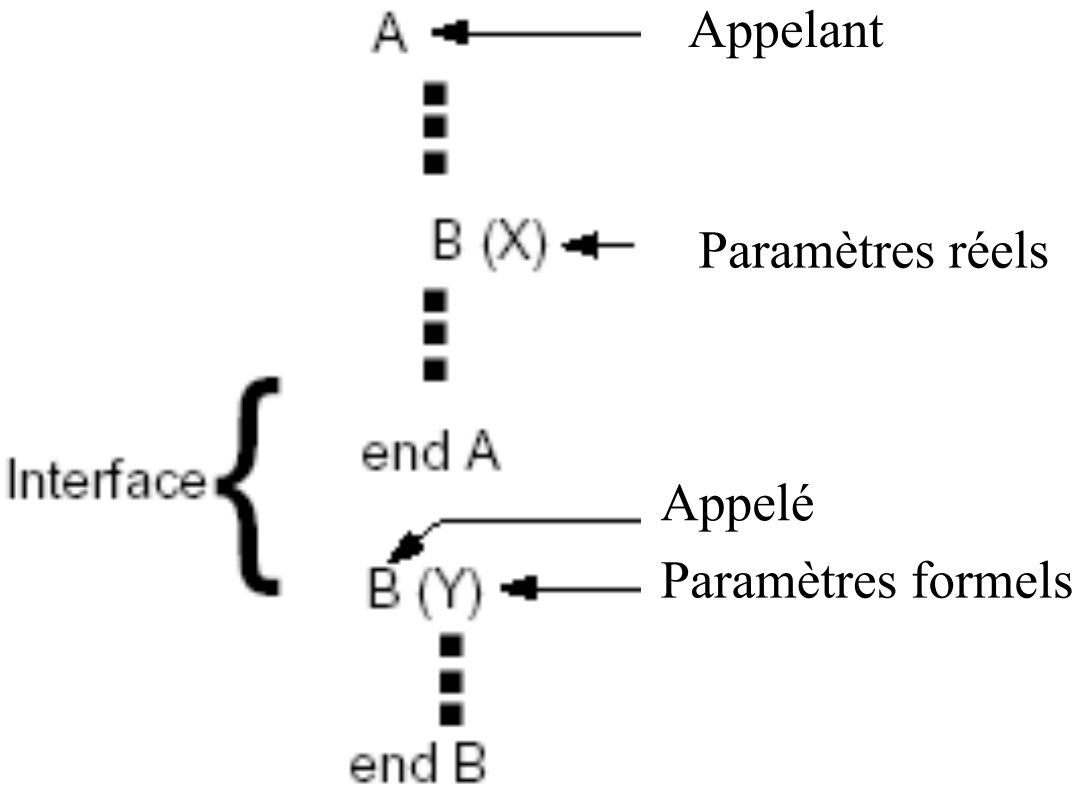
Principes généraux

- Chaque module à être intégré devrait avoir passé un test isolé (unitaire).
- Le test d'intégration doit être effectué à un niveau supérieur d'abstraction
  - en considérant un programme comme des blocs de structures atomiques dont on cible les interconnections.
- Objectif : *Guider* les testeurs durant le test d'intégration, aider à définir le critère pour déterminer quand *arrêter* le test d'intégration.

Définitions préliminaires

- L'*interface* entre 2 unités est une mise en correspondance des paramètres *réels (de l'appelant) aux paramètres formels (de l'appelé).*
- Les connections du flot de données entre les unités sont souvent complexes : une source riche d'anomalies.
- Durant le test d'intégration, nous voulons observer les *définitions* et les *usages* à travers les différentes unités.
- Pour augmenter notre confiance dans les interfaces, nous voulons nous assurer que les variables définies dans les unités *appelantes* sont proprement utilisées dans les unités appelées.
- On observe ici les définitions de variables *avant* les appels et les retours des unités appelées, et l'utilisation des variables juste *après* les appels et retours de l'unité appelée.
- Nous référons par *variables de couplage* les variables qui sont définies dans une unité et utilisées dans une autre.

Exemple court



Ammann & Offutt

Définitions préliminaires I

- Nous différencions trois types de couplage entre unités :
  - couplage de paramètres
  - couplage de données partagées (i.e., variables non locales partagées par plusieurs modules).
  - couplage de périphériques externes (i.e., références de plusieurs modules au même périphérique externe).

- Les concepts ici s'appliquent à tous les types de couplage, même si les discussions seront en termes de paramètres.

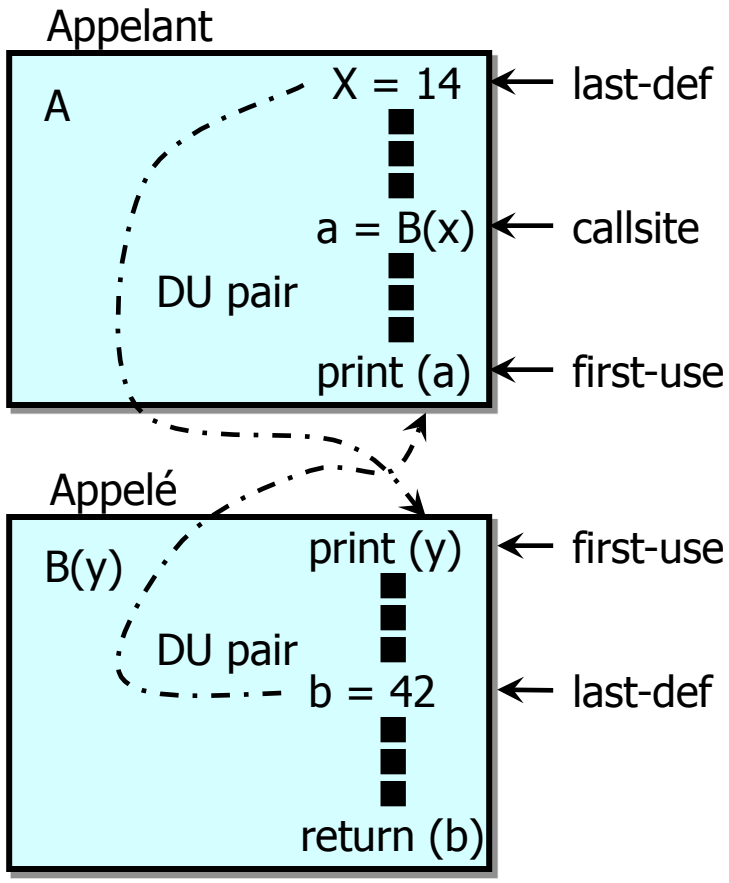
Définitions préliminaires II

- *Site d'appels (Call sites)* : les nœuds (CFG) de l'appelant (A) où l'appelé (B) est invoqué. Dans ce qui suit, on suppose qu'un paramètre réel x est passé de A à B et mappé à y

- *Last-Defs* : l'ensemble des nœuds (CFG) qui définissent x pour lesquels il y a un chemin def-clear du nœud à travers le site d'appels (call site) pour une utilisation dans une autre unité.

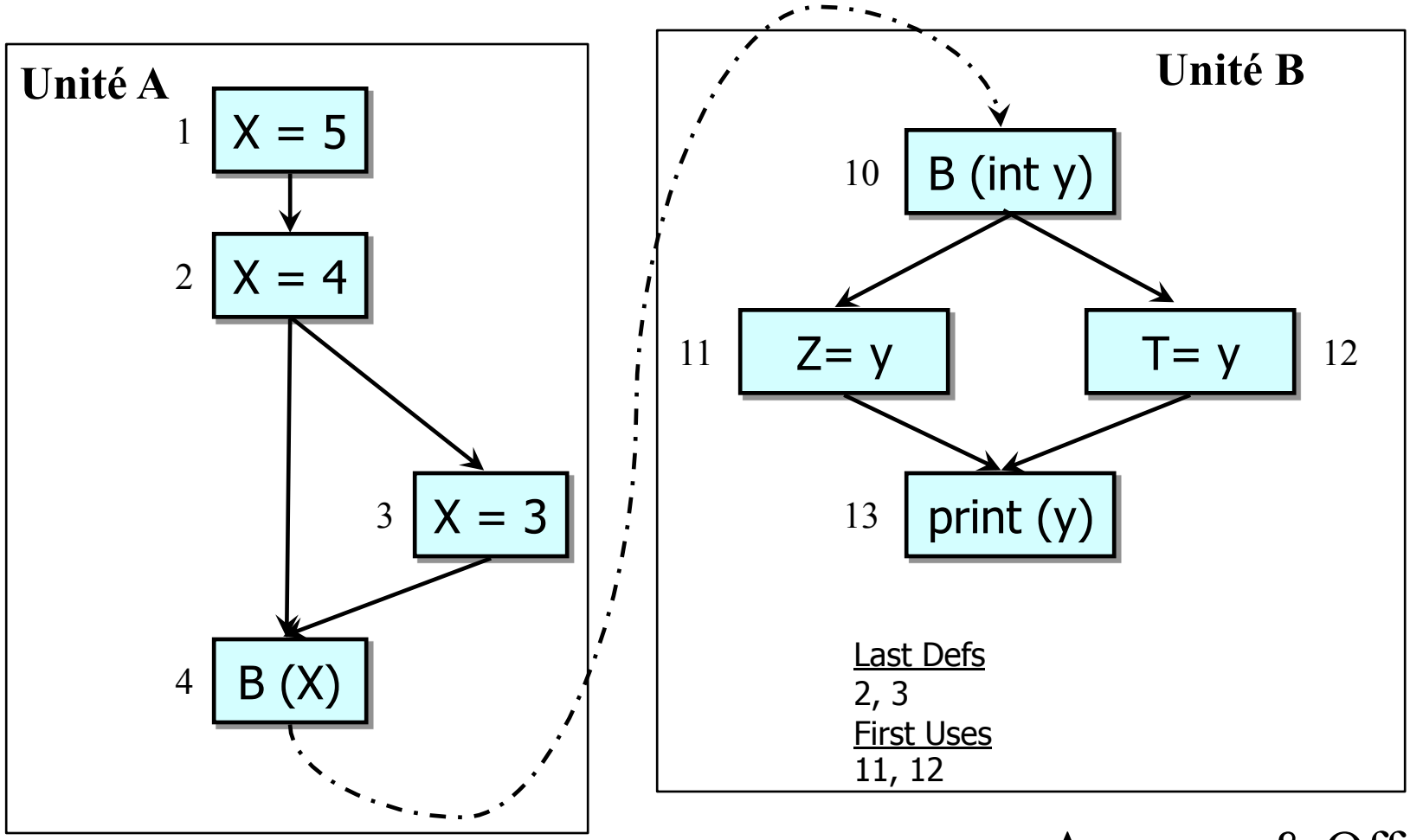
- *First-Uses* : l'ensemble des nœuds (CFG) qui utilisent y et pour lesquels il existe un chemin def-clear et use-clear entre le site d'appels (call site) (si l'utilisation est dans l'appeleur (caller) ou au point d'entrée (si l' utilisation est dans le répondeur (callee)) et les nœuds.

Exemple court



Ammann & Offutt

Exemple avec deux CFG's



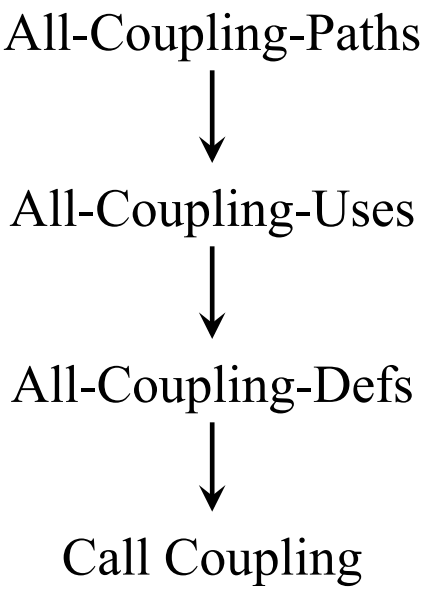
Ammann & Offutt

Chemins de couplage et critères

- Un couplage du-path (ou couplage de chemin) est un chemin d'un last-def à un first-use.
- Liste des critères (appliqués aux 3 types de couplage) :
  - call-coupling :
    - ✓ requiert l'exécution de tous les sites d'appel dans l'appelant.
  - all-coupling-defs :
    - ✓ requiert que pour chaque définition de couplage, au moins un chemin de couplage **à au moins un** couplage accessible utilisé soit exécuté.
  - all-coupling-uses:
    - ✓ requiert que, pour chaque définition de couplage, au moins un chemin de couplage pour **chaque** couplage accessible utilisé soit exécuté.
  - all-coupling-paths:
    - ✓ requiert que **tous** les chemins de couplage libres de boucle soient exécutés

Couplage de paramètres

- Hierarchie de “subsumption” :



Exemple étendu

```
1.   procedure QUADRATIC is
2.   …
3.   begin
4.     GET (Control_Flag);
5.     if (Control_Flag = 1) then
6.       GET(X); --- last-def-before-call (X)
7.       GET(Y); --- last-def-before-call (Y)
8.       GET(Z); --- last-def-before-call (Z)
9.     else
10.      X := 0; --- last-def-before-call (X)
11.      Y := 0; --- last-def-before-call (Y)
12.      Z := 0; --- last-def-before-call (Z)
13.    end if;
14.    OK := TRUE; --- last-def-before-call (OK)
15.    ROOT(X,Y,Z,R1,R2,OK); --- call-site
16.    if OK then --- first-use-after-call (OK)
17.      PUT(R1); --- first-use-after-call (R1)
18.      PUT(R2); --- first-use-after-call (R2)
19.    else
20.      PUT("No solution");
21.    end if;
22.  end QUADRATIC;
```

```
1.   procedure ROOT(A,B,C: in FLOAT;
2.                   ROOT1,ROOT2: out FLOAT;
3.                   Result: in out BOOLEAN) is
4.   …
5.   D: FLOAT
6.   …
7.   begin
8.     D := B**2-4.0*A*C;
9.          --- first-use-in-callee (A,B,C)
10.    if (Result and D < 0.0) then
11.         --- first-use-in-callee (Result)
12.      Result := FALSE
13.         --- last-def-before-return (Result)
14.      return;
15.    end if;
16.    ROOT1 := (-B+sqrt(D))/(2.0*A);
17.         --- last-def-before-return (ROOT1)
18.    ROOT2 := (-B-sqrt(D))/(2.0*A);
19.         --- last-def-before-return (ROOT2)
20.    Result := TRUE;
21.         --- last-def-before-return (Result)
22.  end ROOT;
```

Exemple II

- Last-def-before-call(QUADRATIC,15,x) = {6, 10}
- Last-def-before-return(ROOT,result) = {12,20}
- First-use-after-call(QUADRATIC,15,ROOT1) = {17}
- First-use-in-callee(ROOT,A) = {8}

Version Java

```
1  // Program to compute the quadratic root for two numbers
2  import java.lang.Math;
3
4  class Quadratic
5  {
6  private static float Root1, Root2;
7
8  public static void main (String[] argv)
9  {
10    int X, Y, Z;
11    boolean ok;
12    int controlFlag = Integer.parseInt (argv[0]);
13    if (controlFlag == 1)
14    {
15       X = Integer.parseInt (argv[1]);
16       Y = Integer.parseInt (argv[2]);
17       Z = Integer.parseInt (argv[3]);
18    }
19    else
20    {
21       X = 10;
22       Y = 9;
23       Z = 12;
24    }
25    ok = Root (X, Y, Z);
26    if (ok)
27       System.out.println
28          ("Quadratic: " + Root1 + Root2);
29    else
30       System.out.println ("No solution.");
31 }
32
33 // Three positive integers, finds the quadratic root
34 private static boolean Root (int  A, int B, int C)
```

Version Java (suite)

```
35  {
36     float D;
37
38     D = (float) Math.pow((double)B, (double)2-4.0*A*C);
39     if (D < 0.0)
40     {
41        Result = false;
42        return (Result);
43     }
44     Root1 = (float) ((-B + Math.sqrt(D)) / (2.0*A));
45     Root2 = (float) ((-B - Math.sqrt(D)) / (2.0*A));
46     Result = true;
47     return (Result);
48  } // End method Root
49
50  } // End class Quadratic
```

Exemple III

- T1 = (1, 1, 1, 1) I.e., (Control_Flag,X,Y,Z)
- T2 = (1, 1, 2, 1)
- T3 = (0, 1, 1, 1)
- {T1} satisfait “call-coupling”
- {T2,T3} satisfait “all-coupling-defs” , “all-coupling-uses”, “all-coupling-paths”

Étude de cas

- Le critère “All-coupling-uses”.
- La comparaison avec le test catégorie-partition.
- Le programme Mistix, C, 31 unités fonctionnelles, 65 appels de fonction, 533 LOCs, 21 anomalies propagées (mais 12 peuvent être détectées), les cas de test imaginés manuellement.
- Les anomalies, les tests categories-partition, et les tests “all-coupling-uses” ont été créés par différentes personnes.

Résultats

- La technique "coupling-based" performe mieux (11/12 vs 7/12) que la catégorie-partition avec la moitié de cas de tests (37 vs 72).
- Menaces pour la validité :
  - ✓ échantillons défectueux,
  - ✓ petits programmes,
  - ✓ Comparaisons avec autres techniques de test d'intégration manquantes.

- Selecting and Using Data for Integration Testing, Harrold and Soffa, IEEE Software, Marêteh 1991.
- A Study of Integration Testing and Software Regression at the Integration Level, Leung and White, IEEE Int. Conf. On Soft. Maintenance, 1990.

Remarques

- Les études empiriques sont rares mais elles sont cruciales puisque la théorie n'aide pas beaucoup à l'évaluation des stratégies de test.
- Mais de nombreuses questions restent en suspens:
  - Quelle représentativité pour les anomalies propagées (type, taille) ?
  - Quelle représentativité pour les programmes étudiés(taille, complexité) ?
  - Est-ce que les cas de tests sont dérivés indépendamment des anomalies (automatisation préférée) ?
  - Résultats – Étant donné que les anomalies propagées sont des échantillons d'une distribution, peut-on déterminer si les différences observées peuvent être obtenues par hasard?

Plan

- Couverture du flot de contrôle :
  - énoncé, limite, condition, couverture du chemin.
- Couverture du flot de données :
  - définitions-usages des données.
- Analyse de la couverture des données.
- Essai de mutation.
- Essai d'intégration :
  - stratégies,
  - critères.
- Conclusions :
  - production des tests de données, outils.

Tests générant des "Code-based"

- Pour trouver des données de test qui exécuteront un énoncé arbitraire Q à l'intérieur d'un programme source, le testeur doit travailler en reculant de Q à travers le flot de contrôle du programme jusqu'à un énoncé de données.
- Pour des programmes plus simples, cela revient à résoudre un ensemble d'inégalités simultanées dans les variables d'introduction du programme, chaque inégalité décrivant le chemin approprié à travers un conditionnel.
- Les conditionnels peuvent être exprimés en fonction des variables locales dérivées des données.

Exemple

```
int z;
scanf("%d%d", &x, &y);
if (x > 3) {
  z = x+y;
  y+= x;
  if (2*z == y) {
    /* statement to be covered */
…
```

Inégalités :

- x> 3
- 2(x+y)=x+y
  ⇔ x = -y

1 Solution:

X = 4

Y= -4

Problèmes

- La présence de boucles ou de récursivité dans le code rend en général impossible l'écriture et la résolution des inégalités.
- Chaque passage à travers une boucle peut altérer les valeurs des variables qui figurent dans un conditionnel suivant et le nombre de passages ne peut pas être déterminé par l'analyse statique.
- La couverture peut être de 100% mais le testeur peut malgré tout manquer quelques fonctionnalités (problème inhérent aux tests de boite blanche)

Outils

- Test de génération
  - Telcordia : http://legacy.cleanscape.net/products/testwise/
  - Parasoft Jtest, and C++Test : http://www.parasoft.com/

- Couverture de Code
  - gcov gcc/g++ outils de couverture
  - Rational Purify/Coverage: http://www.rational.com
  - Rational Test RT
  - Telcordia : http://xsuds.argreenhouse.com/
  - Test et couverture de serveur McCabe : http://www.mccabe.com
  - IPL Cantata et Cantata++: http://www.qcsltd.com/products.htm