Test d'intégration de Classes

Problème de test d'intégration

- 2 problèmes distincts : (1) tester les interactions des composants, (2) déterminer un ordre optimal d'intégration.
- Devrait s'assurer que le composant développé (classes/clusters/sous-systèmes) interagit correctement afin d'obtenir la fonctionnalité désirée.
- Dans un contexte non-OO, le test d'intégration se concentre sur la bonne interface entre les "unités", i.e., paramètres entrée/sortie des sous-routines, valeurs de retour, lecture et écriture des fichiers SGBD (Système de Gestion de Bases de Données) [*DBMS Data Base Management System].*
- Dans un contexte OO, nous nous concentrons sur les classes, clusters, sous-systèmes.
- Le polymorphisme et le dynamic binding conduisent à tester une à plusieurs invocations des mêmes interfaces.

Niveaux d'intégration OO

- L'intégration des membres dans une seule classe.
- L'intégration de deux classes ou plus à travers l'héritage (cluster).
- L'intégration de deux classes ou plus à travers un contenant [containment] (cluster).
- L'intégration de deux classes associées/clusters ou plus pour former un composant (I.e., sous-système).
- L'intégration des sous-systèmes dans une application simple.

Fautes d'intégration

- Deux types d'erreurs d'intégration sont distingués (Haley et Zweben) : erreurs de *domaine* et erreurs de *calcul*.
- Une erreur de domaine se produit quand une donnée spécifique suit le mauvais chemin dû à une erreur dans le flot de contrôle du programme.
- Une erreur de calcul existe quand une donnée spécifique suit un chemin correct, mais une erreur dans une instruction d'affectation amène au calcul par une mauvaise fonction pour une ou plusieurs des variables de sortie.
- Dans les deux cas, il est possible que l'erreur à l'entrée ne se reflète pas dans une erreur à la sortie, puisque la propagation vers la sortie peut être dépendante de l'exécution d'un chemin particulier.
- Le test d'intégration ne peut ignorer la structure interne des classes intégrées (i.e., ne peut pas être simplement une boîte noire).
  - Mais certains chercheurs, en raison de la complexité des tests d'intégration boîte blanche, défendent une approche plus centrée sur les fonctionnalités.

Fautes d'intégration OO

- Mauvais objet rattaché au message (cible polymorphique).
- Appel d'une méthode incorrecte dû à une erreur de codage ou d'un binding inattendu pendant l'exécution.
- Le client envoie un message qui viole les pré-conditions du serveur.
- Le client envoie un message qui viole les contraintes séquentielles (séquences d'opérations légales).

Test des interactions des classes

- Se concentrera sur l'intégration des classes. Des considérations similaires peuvent être faites au sujet de plus grands composants qui sont intégrés.
- Quand de nouvelles classes sont intégrées, nous devons tester leur intégration/interaction avec les classes existantes.
- Nous considérons une paire de classes à la fois : client, serveur.
- ***Portée de l'interaction*** : niveau méthode ou niveau classe.
  - ***Niveau Méthode*** : Une occurrence de la classe du serveur est soit construite ou passée comme un paramètre dans une méthode de la classe client+ appel à l'occurrence du serveur.
  - ***Niveau classe*** : Une occurrence de la classe du serveur est déclarée comme un attribut de la classe client.

Exemple

```
class A {
  private B b;
  public A() {
    …;
  }
  public void A_m1() {
    b.B_m1();
  }
  public void A_m2(B bb) {
    bb.B_m1();
  }
  public void A_m3() {
    b.B_m2();
  }
  public void A_m4() {
    b.B_m3();
  }
  …
}
```

```
class B {
  public B() {
    …;
  }
  public void B_m1{} {
    …;
  }
  public void B_m2{} {
    …;
  }
  public void B_m3{} {
    …;
  }
}
```

Définition du problème

- Paire de classes : *client*, *serveur*.
- Stratégie : réutiliser les cas de test utilisés pour tester la classe *client*.
- Contrainte : nous ne pouvons pas nous permettre de repasser autant de cas de tests que durant le test de classe (le *serveur* était alors représenté par un stub).
- Nous voulons nous concentrer sur l'exercice des interactions entre le *client* et le *serveur* (souci d'efficacité).
- Nous supposons que le code source est disponible (boîte blanche).

Problèmes

- Besoin d'analyser, pour chaque cas de test (séquence d'exécution de méthodes), comment la classe *serveur* cible est appelée.
  - Nous avons besoin d'analyser le flot de contrôle du *client* et d'identifier les objets sur lesquels les méthodes *serveur* sont invoquées.
  - Ceci détermine quelles séquences de méthodes *serveur* peuvent être exécutées par les cas de test du *client*.

Stratégie

- Nous considérons seulement une paire *client-serveur* à la fois.

1. Pour chaque méthode client, on génère un graphe de contrôle annoté intégrant les appels avec les autres méthodes *client*.
2. Les chemins faisables dans un graphe de flot de contrôle sont utilisés pour dériver les séquences de méthodes *serveur* qui peuvent être exécutées par la méthode *client* correspondante.
3. Ces séquences sont à leur tour utilisées pour identifier les séquences de méthodes *serveur* qui peuvent être déclenchées par les séquences de la classe *client*.
4. Nous identifions et enlevons ensuite les séquences *serveur* redondantes (pour des séquences *client* différentes).
5. Enfin, nous identifions le sous-ensemble minimal des séquences de test du *client* qui servent à exercer les séquences *serveur* restantes : celles qui ont besoin d'être ré-exécutées avec la classe *serveur* réelle (au lieu de son stub) pour tester les interactions.

L. C. Briand, Y. Labiche and Y. Wang,
"A Comprehensive and Systematic Methodology for Client-Server Class Integration Testing "
Proceedings of the International Symposium on Software Reliability Engineering (ISSRE'03)

Exemple

```
1   public class A {
2     private B b1, b2;
3     private int dA1, dA2, dA3;
4     public A(){
5       b1 = new B(); b2 = new B();
6       dA1 = 0; dA2 = 0; dA3 = 0;
7     }
8     public int mA1(int i, B b){
9       i++;
10      if ( i > 0 ){
11        dA1 = b1.mB1(i);
12        dA2 = b1.mB2(i);
13      } else {
14        dA1 = b2.mB2(dA1);
15        dA2 = b2.mB1(dA2);
16      }
17      dA3 = b.mB3(b1.mB3(dA1));
18      return dA3;
19    }
20    public void mA2(B bb){
21      dA2 = bb.mB1(dA1);
22      mA1(dA2, bb);
23    }
24    public int mA3(int i){
25      B aB1 = new B();
26      B aB2 = new B();
27      int j = aB1.mB1(i);
28      if ( i > j)
29        return aB1.mB1(j);
30      else
31        return aB2.mB2(j);
32    }
33    public int mA4(int i){
34      B aB3 = new B();
35      // Class C, not shown, has
36      // a static attribute VAR
37      // of type int.
38      C.VAR = 12
39      int j = b1.mB1(i);
…
45    }
46    public void mA5(){
47      B aB = new B();
48      dA2 = aB.mB2(2);
49      mA1(1, b2);
50    }
51    public boolean mA6(){
52      FileWriter out;
53      out = new FileWriter(new File("F.txt"));
54      if ( dmA1 > 10 )
55        out.write('T');
56      else
57        out.write('F');
58      out.close();
59      b1.mB5();
60      C.VAR = 0;
61    }
62  }
```

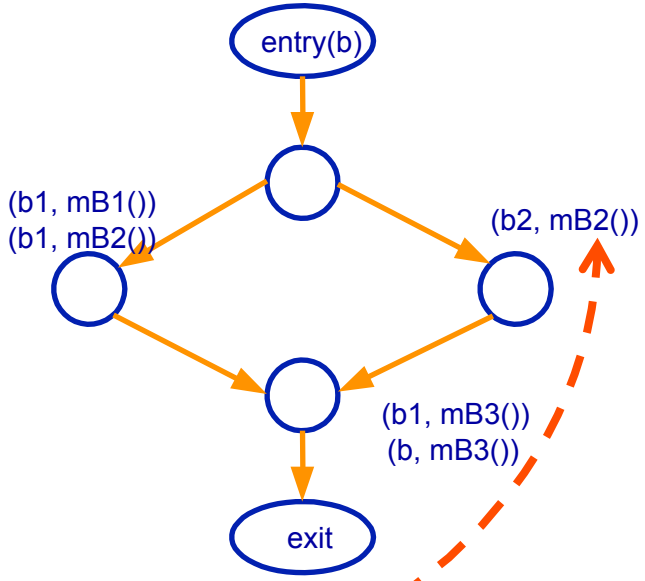
```
1   public class B {
2     private int dB1, dB2;
3     public B(){ dB1 = 1; }
4     public int mB1(int j){
5       dB1 = dB1 - j;
6       j = (dB1 + dB2) * C.VAR;
7       return j;
8     }
9     public int mB2(int j){
10      dB2 = dB2 - j;
11      if (dB2>0) {
12        dB1 -= j;
13        return dB1;
14      } else
15        dB1 += j;
16      for (…)
17        …
18      return dB2;
19    }
20    public int mB3(int i){
21      return (dB1 - dB2) * i;
22    }
23    public boolean mB4(){
24      FileReader in;
25      in = new FileReader(new File("F.txt"));
26      char c;
27      c = in.read();
28      in.close();
29      if (c == 'T')
30        return true;
31      else
32        return false;
33      }
…
39  }
```

*Étape 1 - CFG (Control flow graph) **annoté***

- Le CFG (graphe de flot de contrôle) de la méthode *client*.
- Les nœuds sont annotés avec la séquence d'appels aux méthodes du *serveur* qui y sont exécutées.
- Nous voulons aussi connaître les instances spécifiques de *serveurs* sur lesquelles les appels sont exécutés, puisque nous sommes intéressés aux séquences d'appel sur la même instance.
  - Nous devons identifier de manière unique les instances du *serveur* sur lequel les appels sont faits.
    - ✓ Noms symboliques : les appels sont décrits par paires (nom symbolique, nom de la méthode).
- Les nœuds d'entrée/sortie montrent, pour la classe *serveur* cible, les objets qui sont *passés à* / *retournés par* la méthode.
- Construire des CFGs annotés requiert une analyse de code statique et dynamique.

Étape 1 - Exemple de CFG annoté

```
public class A{
  private B b1, b2;
  private int dA1, dA2, dA3;…
  public int mA1(int i, B b){
    i++;
    if ( i > 0) {
      dA1 = b1.mB1(i);
      dA2 = b1.mB2(i);
    } else {
      dA1 = b2.mB2(dA1);}
    dA3 = b.mB3(b1.mB3(dA1));
    return dA3;  }…}
```



- A : mA1() a un paramètre formel de type B, la classe *serveur*, nommé b.
- Noter que, pour simplifier, nous utilisons les noms d'attribut et de paramètre au lieu des différents noms symboliques.

Étape 1 - CFG inter-procédural (1)

- Les méthodes *client* peuvent s'appeler les unes, les autres.
- Ceci donne des séquences plus compliquées de méthodes *serveur*.
- Nous avons besoin de combiner le flot de contrôle des méthodes *client*.

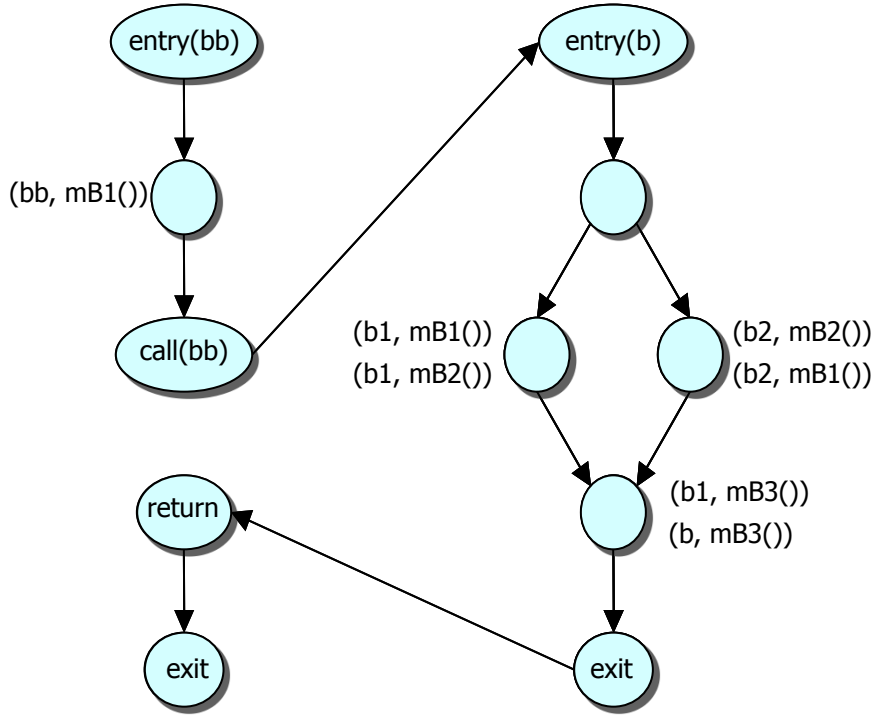
→Les graphes de flot de contrôle inter-procéduraux (ICFG).

Étape 1 - CFG inter-procédural (2)

- Des nœuds spécifiques sont créés pour connecter les graphes de flot de contrôle des méthodes qui s'appellent les unes, les autres.
- Les endroits d'appel d'autres méthodes sont divisés en nœuds d'appels et nœuds de retour.
- Les nœuds d'appel sont connectés aux nœuds d'entrée des méthodes qu'ils invoquent.
- Les nœuds de sortie sont connectés au nœud de retour de tous les endroits d'appel qui invoquent la méthode.
- Pour les variables de classe *serveur*, les nœuds d'appel indiquent les paramètres réels utilisés dans l'appel tandis que les nœuds de retour indiquent les variables auxquelles les valeurs de retour de l'appel sont assignées.

Étape 1 - Exemple ICFG

```
20    public void mA2(B bb){
21      dA2 = bb.mB1(dA1);
22      mA1(dA2, bb);
23    }
```



ICFG pour méthode `mA2()`

- Durant l'appel à mA1() en mA2(), le paramètre réel bb est remplacé par le paramètre formel b dans la méthode mA1(), impliquant que le dernier appel d'une instance de la classe B exécuté en mA1() est exécuté sur le paramètre bb.
- Cas spéciaux : constructeurs, appels statiques → nom de la classe *serveur*

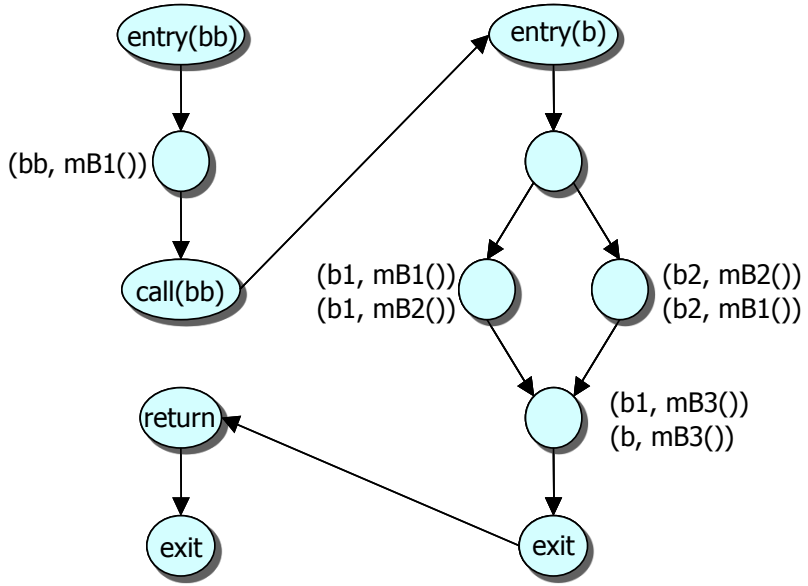
Étape 2 - Séquences des méthodes de serveur

- Le ICFG annoté est utilisé pour obtenir les séquences des méthodes *serveur* qui sont déclenchées par la méthode *client* correspondante.
- Pour chaque méthode *client*, les chemins dans le ICFG correspondant sont alors déterminés.
- Problèmes :
  - boucles et appels récursifs (boucles infinies), et
  - chemins infaisables.
- Mapping entre les paramètres réels et formels : occurrences des paramètres formels remplacés avec les paramètres réels.
- À partir de chaque chemin dans le ICFG de la méthode client, nous déterminons la séquence des appels de méthodes *serveur* et nous séparons les séquences correspondant aux différents noms symboliques impliqués.
- Nous obtenons ainsi les séquences de méthodes *serveur* déclenchées par une simple méthode *client*.

Étape 2 - Exemple

Méthodes serveurs appelées par deux chemins dans l'ICFG de `mA2()` .	Séquences correspondantes de la méthode serveur pour chaque nom symbolique (instance du serveur)
(`bb` , `mB1()`), (`b1` , `mB1()`), (`b1` , `mB2()`), (`b1` , `mB3()`), (`bb` , `mB3()`)	`bb` : `mB1().mB3()` `b1` : `mB1(). mB2(). mB3()`

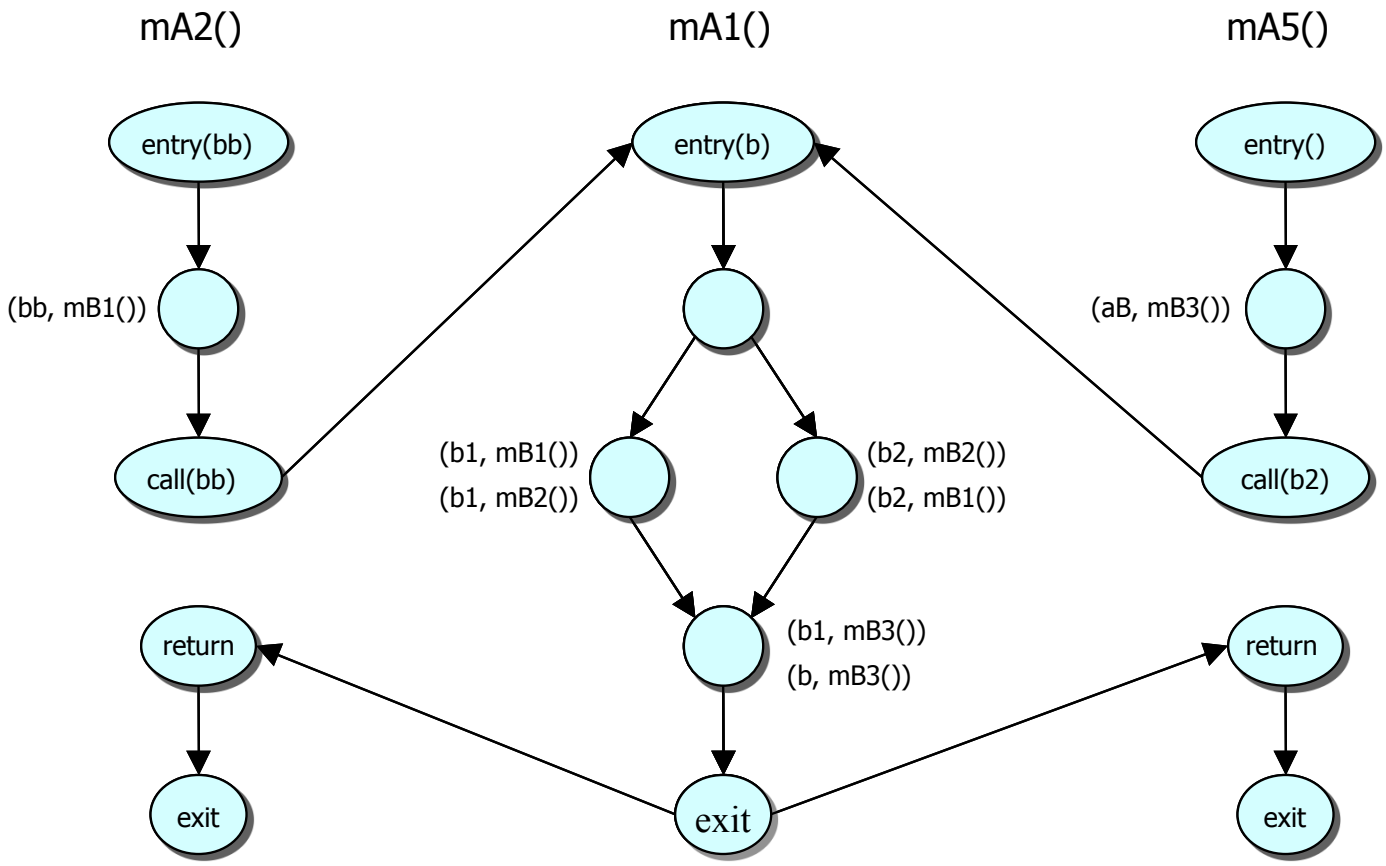
(`bb` , `mB1()`), (`b2` , `mB2()`), (`b2` , `mB1()`), (`b1` , `mB3()`), (`bb` , `mB3()`)	`bb` : `mB1().mB3()`
	`b2` : `mB2(). mB1()`
	`b1` : `mB3()`



Étape 3 - Séquences de méthodes client

- Les séquences de méthodes *serveur* qui nous intéressent sont celles déterminées par l'enchaînement de séquences de méthodes *serveur* obtenu à partir des séquences des méthodes *client*.
- Nous devons aussi nous assurer, comme précédemment, que les noms symboliques identifient de manière unique les instances du *serveur*.
- Chaque séquence de méthodes *client* déclenche zéro, une, ou plusieurs séquences de méthodes *serveur*.
- Ces séquences sont des paires de la forme (*symbName, seqServ*) où *symbName* est un nom symbolique identifiant l'objet sur lequel la séquence des méthodes serveurs *seqServ* est exécutée.
  - *symbName* peut aussi référer au nom de la classe du serveur quand *seqServ* est une séquence des méthodes statiques (incluant les constructeurs) de la classe serveur.
- Exemple : `mA1().mA2().mA5()` ont été définies et s'exécutent durant le test de classe A.

Étape 3 - Exemple ICFG



Séquences du serveur déclenchées par mA1(), mA5()

Méthodes de serveur déclenchées par les deux chemins dans l'ICFG de **`mA1()`**	Séquences de la méthode serveur pour chaque nom symbolique (instance de serveur)
(b1, mB1()), (b1, mB2()), (b1, mB3()), (b, mB3())	b1: mB1().mB2().mB3()
	b: mB3()
(b2, mB2()), (b2, mB1()), (b1, mB3()), (b, mB3())	b2: mB2().mB1()
	b: mB3()

Méthodes serveurs déclenchées par les deux chemins dans l'ICFG de **`mA5()`**	Séquences de la méthode serveur pour chaque nom symbolique (instance de serveur)
(aB, mB3()), (b1, mB1()), (b1, mB2()), (b1, mB3()), (b2, mB3())	aB: mB3()
	b1: mB1().mB2().mB3()
	b2: mB3()
(aB, mB3()), (b2, mB2()), (b2, mB1()), (b1, mB3()), (b2, mB3())	aB: mB2()
	b1: mB3()
	b2: mB2().mB1().mB3()

Séquences du serveur pour l'attribut b2

<table>
<tr><td rowspan="2">Hypothèse 1</td><td>Séquences de méthodes client</td><td colspan="3">mA1().mA2().mA5()
↙ ↓ ↘</td></tr>
<tr><td>Séquences de méthodes serveur déclenchées par les méthodes clients individuelles sur b2.</td><td>mB2().mB1()</td><td>mB2().mB1()</td><td>mB2().mB1().mB3()
mB3()</td></tr>
<tr><td rowspan="3">Hypothèse 2</td><td>Séquence de méthodes client</td><td colspan="3">mA1().mA2().mA5()
↙ ↓ ↘</td></tr>
<tr><td rowspan="2">Séquences de méthodes serveur déclenchées par les méthodes clients individuelles sur b2.</td><td>mB2().mB1().mB3()</td><td>mB1().mB2()mB1()
.mB3()</td><td>mB2().mB1().mB3()</td></tr>
<tr><td>mB3()</td><td>mB1().mB3()</td><td>mB3()</td></tr>
</table>

- Hypothèse 1 : Les attributs b1 et b2, et les paramètres b et bb référent des instances distinctes.
- Hypothèse 2 : b2=b=bb, en mA1() et mA2().

Étape 4 - Éliminer la redondance

- Ignorer les séquences de méthodes *client* qui ne déclenchent aucune séquence de méthodes *serveur*, vu que nous sommes seulement intéressés par l'interaction entre un client et une classe *serveur*.
- Si les méthodes impliquées dans la séquence S_1 de méthodes *client*, apparaissent toutes dans le même ordre (dans la séquence) dans une séquence S_2 de méthodes *client*, alors les séquences de méthodes *serveur* déclenchées par S_1 sont comprises dans les séquences des méthodes *serveur* déclenchées par S_2 → S_1 peut être retirée.
- Deux séquences différentes de méthodes *client* peuvent déclencher des séquences de méthodes *serveur* identiques ou se chevauchant (i.e., il y a une relation d' « inclusion » entre les séquences déclenchées) bien que concernant des instances (noms symboliques) différentes.
  - Nous avons besoin d'une stratégie systématique pour tirer profit de ces redondances afin de réduire le nombre de séquences de test de classe client exécutées.

Étape 4 - Critères de redondance

- **Critère 1 – Séquences de méthodes *serveur***

  Pour chaque paire de n-uplets (($cseq_i$, $sseq_j$, $name_k$) , ($cseq_m$, $sseq_n$, $name_l$)) tels que $sseq_j \subseteq sseq_n$, ($cseq_i$, $sseq_j$, $name_k$) peut être éliminé, à moins qu'une ou plusieurs méthodes dans $cseq_i$ n'apparaissent dans aucun autre n-uplet.

- **Critère 2 – Séquences de méthodes *serveur* et origine de l'instance du serveur**

  Pour chaque paire de n-uplets ($cseq_i$, $sseq_j$, $name_k$) et ($cseq_m$, $sseq_n$, $name_l$) tels que $sseq_j \subseteq sseq_n$ et $name_k$ et $name_l$ ont le même genre, e.g., un attribut, ($cseq_i$, $sseq_j$, $name_k$) peut être éliminé, à moins qu'une ou plusieurs méthodes dans $cseq_i$ n'apparaissent dans aucun autre n-uplet.

- **Critère 3 – Les séquences de méthodes serveur et l'instance du serveur** :
  Pour chaque paire de n-uplets ($cseq_i$, $sseq_j$, $name_k$) et ($cseq_m$, $sseq_n$, $name_l$) tels que $sseq_j \subseteq sseq_n$ et $name_k = name_l$, ($cseq_i$, $sseq_j$, $name_k$) peut être éliminé, à moins qu'une ou plusieurs méthodes dans $cseq_i$ n'apparaissent dans aucun autre n-uplet.

Étape 4 - Exemple

n-uplets pour les méthodes client (seules mA1, mA2, mA3, mA4 sont considérées)	Séquences de méthodes client (fournies)	n-uplets pour les séquences de méthodes client par nom symbolique de l'instance du serveur
(mA1, mB1.mB2.mB3, b1) (mA1, mB2.mB1, b2) (mA1, mB3, b) (mA2, mB1.mB3, bb) (mA2, mB1.mB2.mB3, b1) (mA2, mB2.mB1, b2) (mA3, mB1.mB1, aB1) (mA3, mB1, aB1) (mA3, mB2, aB2) (mA4, mB1, b1) (mA4, mB1.mB1, b1) (mA4, mB2, aB3) (mA4, mB2.mB2, aB3)	mA1.mA2 mA2.mA1 mA3 mA4	(mA1.mA2, mB1.mB2.mB3.mB1.mB2.mB3, b1) (mA1.mA2, mB2.mB1.mB2.mB1, b2) (mA1.mA2, mB3, b) (mA1.mA2, mB1.mB3, bb) (mA2.mA1, mB1.mB2.mB3.mB1.mB2.mB3, b1) (mA2.mA1, mB2.mB1.mB2.mB1, b2) (mA2.mA1, mB3, b) (mA2.mA1, mB1.mB3, bb) (mA3, mB1.mB1, aB1) (mA3, mB1, aB1) (mA3, mB2, aB2) (mA4, mB1, b1) (mA4, mB1.mB1, b1) (mA4, mB2, aB3) (mA4, mB2.mB2, aB3)

Étape 4 - Exemple (2)

Critère de Redondance 1	Critère de Redondance 2
(mA1.mA2,mB1.mB2.mB3.mB1.mB2.mB3, b1) (mA1.mA2, mB2.mB1.mB2.mB1, b2) (mA2.mA1, mB1.mB3, bb) (mA3, mB1.mB1, aB1) (mA4, mB2.mB2, aB3)	(mA1.mA2,mB1.mB2.mB3.mB1.mB2.mB3,b1) (mA1.mA2, mB2.mB1.mB2.mB1, b2) (mA2.mA1, mB1.mB3, bb) (mA3, mB1.mB1, aB1) (mA4, mB1.mB1, b1) (mA4, mB2.mB2, aB3)

Critère de Redondance 3
(mA1.mA2,mB1.mB2.mB3.mB1.mB2.mB3, b1) (mA1.mA2, mB2.mB1.mB2.mB1, b2) (mA1.mA2, mB3, b) (mA1.mA2, mB1.mB3, bb) (mA2.mA1, mB1.mB3, bb) (mA3, mB1.mB1, aB1) (mA3, mB2, aB2) (mA4, mB1.mB1, b1) (mA4, mB2.mB2, aB3)

aB1 variable locale - b1 attribut

b différent de b1, bb

aB2 différent de b1, b2, aB3

Étude de cas

- Jadvisor : ordonnanceur de classe et planificateur de cours pour étudiants, et implanté en Java.
- Mutation operators on interface faults
- Comparaison : Test de partition de catégorie (Test fonctionnel de boîte noire).
- Résultat : Critère aide à l'amélioration du coût-efficacité.

	Technique de test (et critère)			
	No critère	Critères 1 & 2	Critère 3	Partition par catégorie
# de n-uplets	14	6	9	NA
# de tests de cas	28	12	18	38
# de mutants non-équivalents supprimés.	28	25	28	23
Score de mutation	100%	89%	100%	82%
# de mutants opérationnels	0	3	0	5

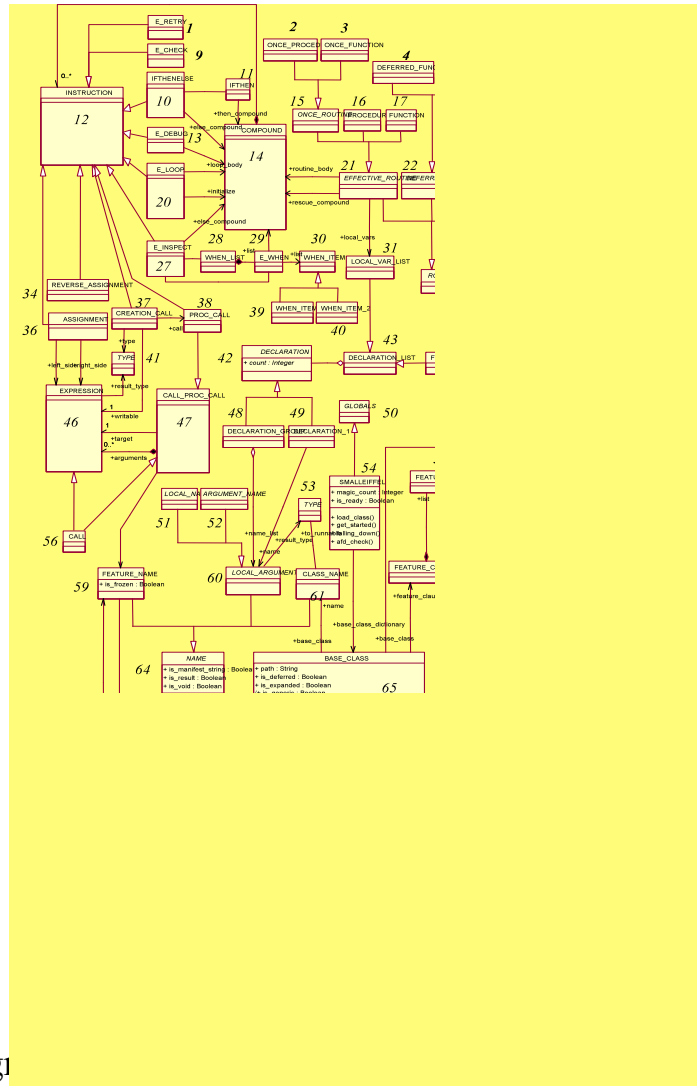
Ordre d'intégration

Problème d'ordre d'intégration

- Il n'est pas conseillé d'exécuter une intégration de classe *big-bang*. L'intégration devrait être faite par étape, mais cela mène à des stubs.
- Il n'est pas toujours faisable de construire un stub qui est plus simple que la pièce de code réelle qu'il simule.
- La génération de stub ne peut pas être automatisée parce que cela requiert une compréhension des sémantiques de fonctions simulées.
- Le potentiel de fautes pour certains stubs peut être le même ou pire que celle de la fonction réelle.
- Minimiser le nombre de stubs développés génère d'importantes économies.
- En général, les graphes de dépendance des classes ne forment pas des hiérarchies simples mais des réseaux complexes.

La plupart des systèmes OO

- Forte Connectivité
- Interdépendances cycliques



Systèmes simples avec cycles

Systèmes	Classes	As	Ag	I	Cycles simples	LOC
ATM	21	49	15	4	30	1390
Ant	25	70	2	11	654	4093
SPM	19	58	10	4	1178	1198
BCEL	45	244	4	46	416,091	3033
DNS	61	234	12	30	16	6710

- ATM (*Automated Teller Machine*) guichet automatique.
- Ant : outil basé Java similaire à l'outil Make d' UNIX (ouverture des projets sources).
- SPM (*Security Patrol Monitoring*) Patrouille de contrôle de sécurité.
- BCEL (*Byte Code Engineering Library*) Librairie d'ingénierie de bytecode : outil pour manipuler les fichiers de classe Java (ouverture des projets sources).
- DNS (*Domain Naming System*) Système de résolution d'adresses IP : réseau nommant les services en Java.

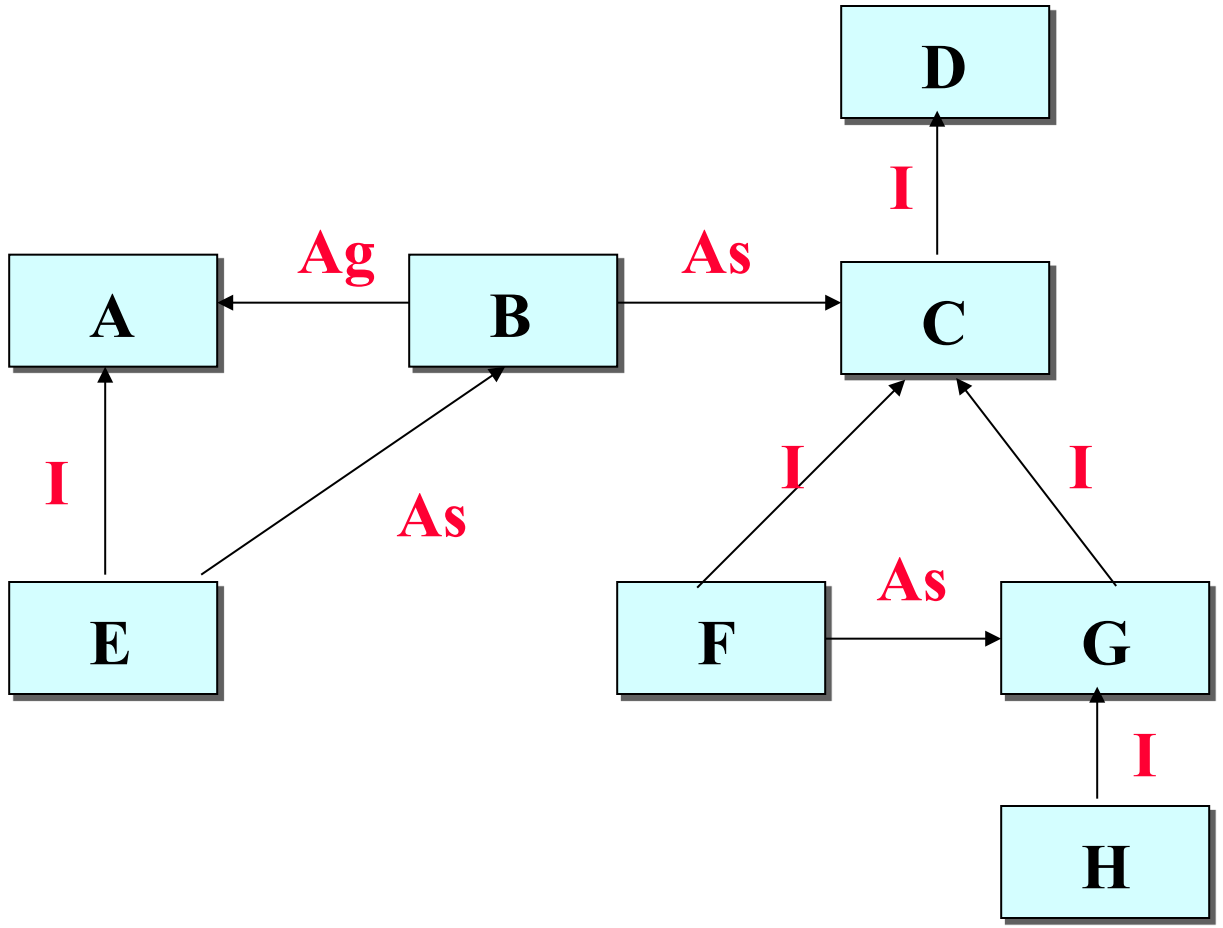
Stratégie de Kung et al

- Vise à produire un ordre partiel des niveaux de tests basé sur les diagrammes de classes.
- Les classes en test à un certain niveau devraient dépendre seulement des classes testées précédemment.
- Aucun stub n'est requis dès que les classes testées précédemment peuvent être incluses dans les niveaux de tests subséquents.
- L'information de dépendance de classe peut soit provenir de l'ingénierie inverse, soit de la documentation de conception (e.g., UML).

Test ORD de Kung et al

- Les deux visent le test d'intégration et le test de régression.
- Diagramme de relation objet (*Object Relation Diagram* (ORD)).
- L'ORD pour un programme P est un digraphe (graphe orienté) à branches étiquetées où les nœuds représentent les classes dans P, et les branches représentent les relations d'héritage, d'agrégation et d'association.
- Pour chaque paire de classes C1 et C2
  - Une branche étiquetée **I** de C1 à C2 si C1 est un enfant de C2.
  - Une branche étiquetée **Ag** de C1 à C2 si C1 contient une ou plusieurs instances de C2 (classe agrégée)
  - Une branche étiquetée **As** de C1 à C2 si C1 dépends de C2 (appelle une méthode de C2, utilise des données de C2, ou a des méthodes qui prennent en paramètre C2). C2 est une classe associée.

Exemple d'ORD

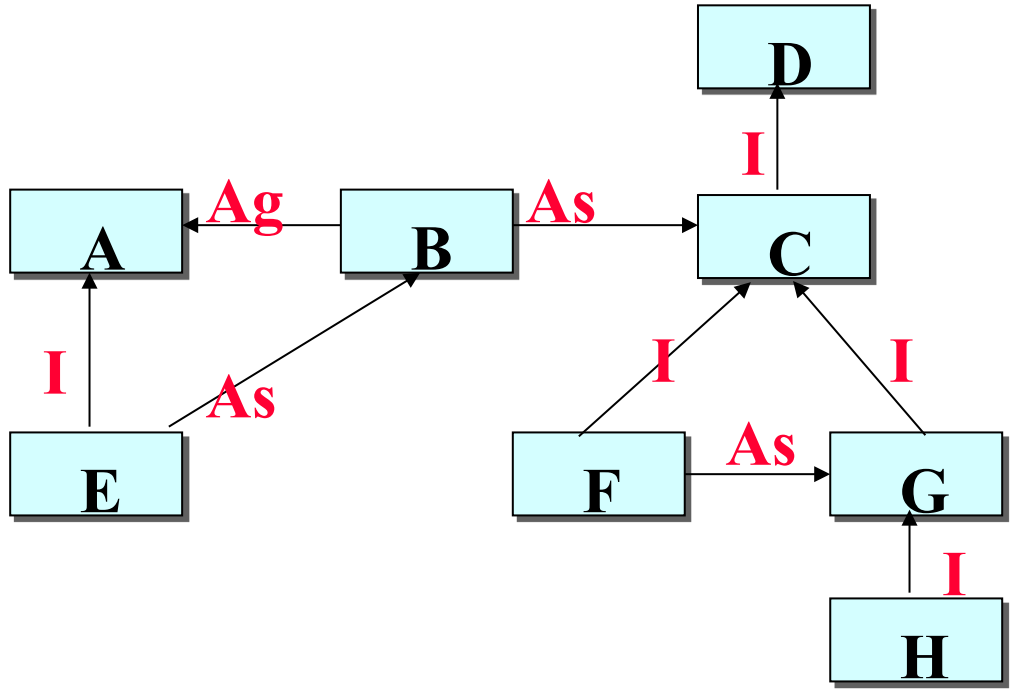


Classe coupe-feu (CFW)

- Identifiez l'effet d'un changement de classe au niveau de classe.
- **CFW**(X) pour une classe X : ensemble de classes qui *peuvent* être affectées par un changement à la classe X, I.e., qui devraient être testées de nouveau quand la classe X est changée.
- Supposons que l'ORD n'est pas modifié par le changement, pour un test adéquat CFW(X) doit inclure : les classes enfants de X, les classes agrégées de X et les classes associées avec X.

Exemple coupe-feu

Classe X	CFW(X)
A	B, E
B	E
C	B, E, F, G, H
D	B, C, E, F, G, H
E	
F	
G	F, H
H	



Dérivation du coupe-feu

- R = {<C1,C2> | il y a une branche étiquetée dirigée de C1 à C2 dans l'ORD}
- CFW(X) = {Ck | <X,Ck> $\in R^+$}, où R^+ est la fermeture transitive et irréflexive de R.
- CFW(X) contient toutes les classes Ck telles qu'il y a un chemin dirigé de Ck à X dans l'ORD.
- Ensemble de classes changées : S = {X1, …, Xq}, CFW(S) = CFW(X1) $\cup$ … $\cup$ CFW(Xq)

Ordre (partiel) de test

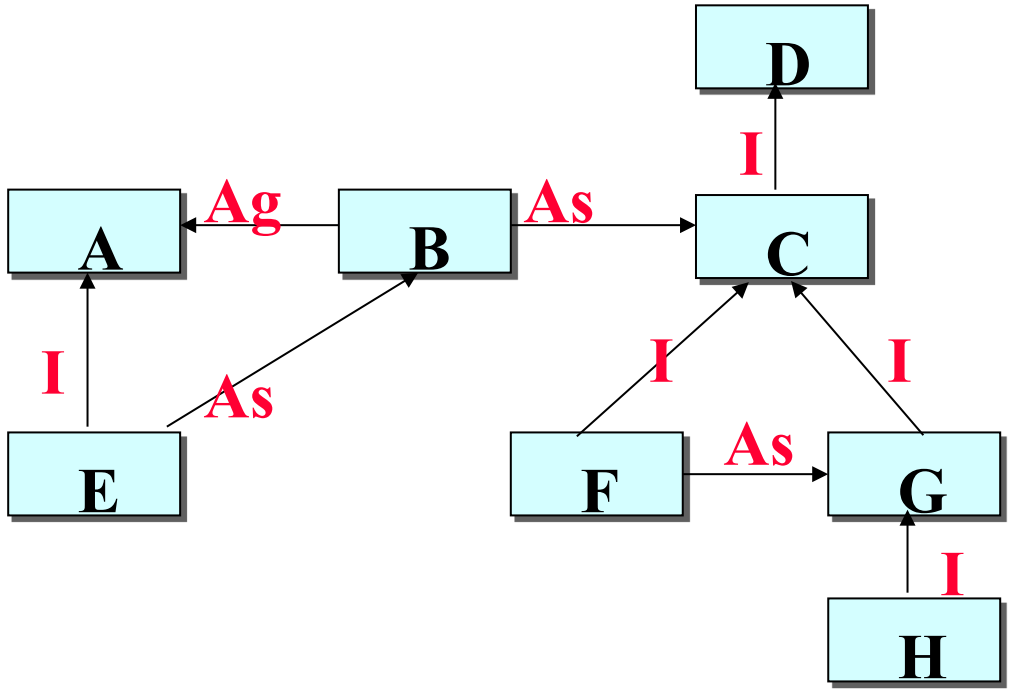
- Fournit au testeur une feuille de route pour mener un test d'intégration.
- L'ordre désirable est celui qui minimise le nombre de stubs ou l'effort de création des stubs.
- Teste les classes indépendantes en premier, puis teste les classes dépendantes en se basant sur leurs relations. e.g., teste les classes composants avant les classes qui les contiennent et permet au testeur d'utiliser les classes composants réelles au lieu des stubs.
- De manière similaire, un enfant devrait être testé après ses classes parents.

Ordre d'intégration pour des ORDs acycliques

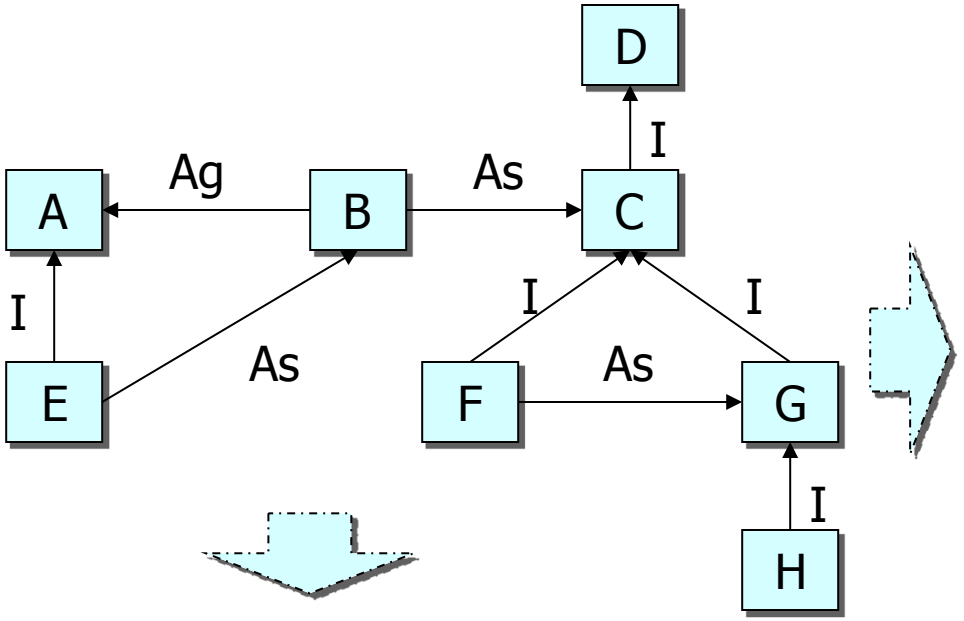
- Peut générer un ordre de test qui assure que X soit testé *avant* toutes les classes de CFW(X).
- Les stubs ne sont pas requis.
- 4 niveaux de tests successifs:
- Plusieurs classes au même niveau => ordre *partiel.*
- Exemple de test de régression : le code change dans A et D => CFW({A,D}) = {B, C, E, F, G, H} → *toutes* les classes doivent être testées de nouveau.
- La technique de Kung et al est simplement un tri topologique, en termes de théorie de graphe (visiter chaque sommet avant ses successeurs).

Exemple d'ordre de test

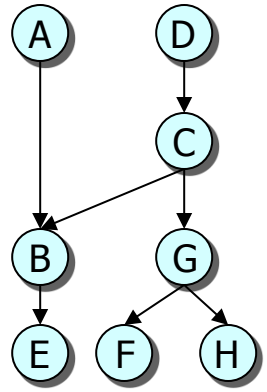
Niveau de test	Classe(s)
1	A, D
2	C
3	B, G
4	E, F, H



Étapes



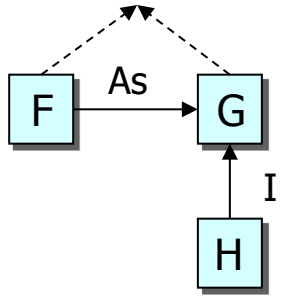
Classe X	CFW(X)
A	B, E
B	E
C	B, E, F, G, H
D	B, C, E, F, G, H
E	
F	
G	F, H
H	



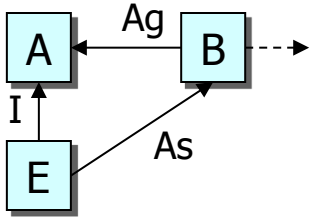
Niveau de test	Classe(s)
1	A, D
2	C
3	B, G
4	E, F, H

Classes abstraites, Dynamic Binding

- Kung et al ne tiennent pas compte de la relation dynamique (pendant l'exécution) entre les classes.
  - La classe F associée avec G qui est un parent de la classe H.
  - Dû au polymorphisme, F peut être dynamiquement associée avec H et, par conséquence, devrait être testée avec H.

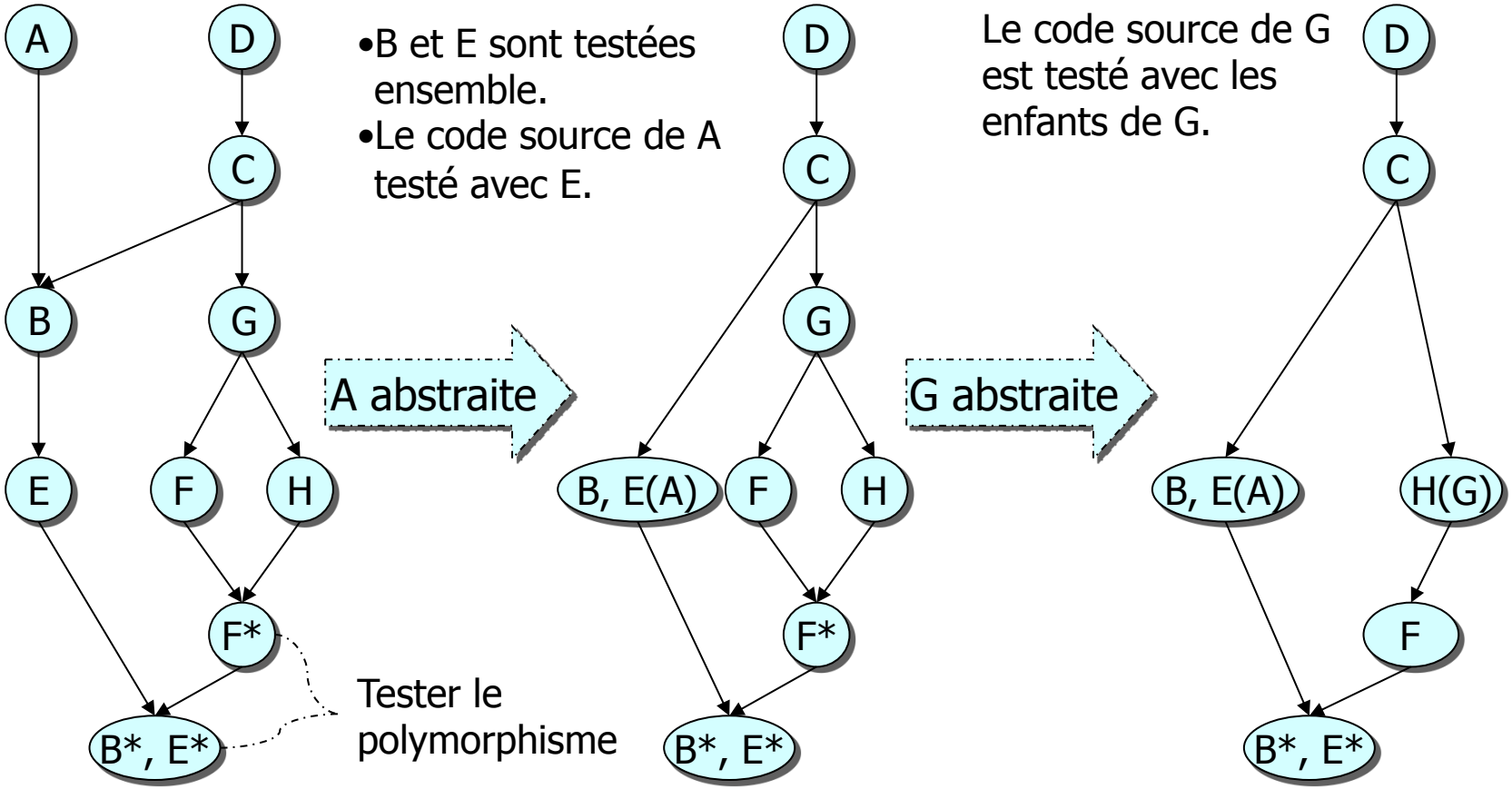


- Quelques niveaux de test deviennent (partiellement) infaisables à cause des classes abstraites.
  - Si A est une classe abstraite, la tester au niveau 1 est infaisable.
  - Tester B au niveau 3 requiert maintenant qu'un enfant de la classe E soit instancié (au lieu de A). E ne peut pas être testée après B.



Niveau de test	Classe(s)
1	A, D
2	C
3	B, G
4	E, F, H

Classes abstraites, Dynamic Binding **(2)**



B*: Tester les interfaces polymorphiques de B/E, B/(F, G, H)
E*: Tester les interfaces polymorphiques de E/B/(F,G,H)
Y(X): Tester Y avec du code hérité de X

Les ORDs cycliques

- En pratique, les ORDs sont souvent cycliques.
  - Cela est dû aux besoins de performance, à la faible conception, aux changements tardifs, …
  - Le tri topologique ne peut pas être appliqué.
- *cluster* :
  - Ensemble maximal de nœuds qui sont mutuellement atteignables par la relation R^+ (composant fortement connexe en théorie de graphe).
- *Cycle interrompu* : identifie et élimine une branche du cluster et répète ceci jusqu'à ce que le graphe (dans le cluster) devienne acyclique.
  - Quelle relation de branche(s) doit-on éliminer ?
    - ✓ *Associations*
    - ✓ Théorème : tous les cycles orientés contiennent toujours au moins une branche pour une association.
  - Chaque suppression d'association résulte en au moins un stub.
  - Utiliser l'ordre topologique sur le graphe acyclique résultant.
  - S'il y a un choix, il serait sage de sélectionner l'association qui mène au développement du stub le plus simple possible.

Questions

- Classe « stubbée » versus stub
  - Les stubs doivent rester simples, e.g., flot de contrôle séquentiel.
  - Il est mieux d'avoir un stub personnalisé pour chaque classe cliente de la classe stubbed.
- Éliminer les associations versus les relations d'héritage et d'agrégation.
  - Les agrégations impliquent les interactions intenses, e.g., patron de conception State.
  - Plusieurs méthodes héritées, non-prioritaires doivent être testées de nouveau dans les sous-classes.

Conclusions

- Des algorithmes existent et peuvent être utilisés pour trier l'intégration des classes.
- Les stratégies pour interrompre les cycles ont besoin d'être investiguées. Certaines approches sont basées sur les méta-heuristiques.
- Les interfaces de classe client-serveur peuvent être employés en réutilisant les cas de test de la classe client et en employant les chemins de couplage interclasse.
- Peut être généralisé à l'intégration des systèmes.
- Problème: Que penser de l'intégration des (tiers) composants pour lesquels aucun code source ou information de conception n'est disponible ?