Les tests d'intégration

Yassamine Seladji

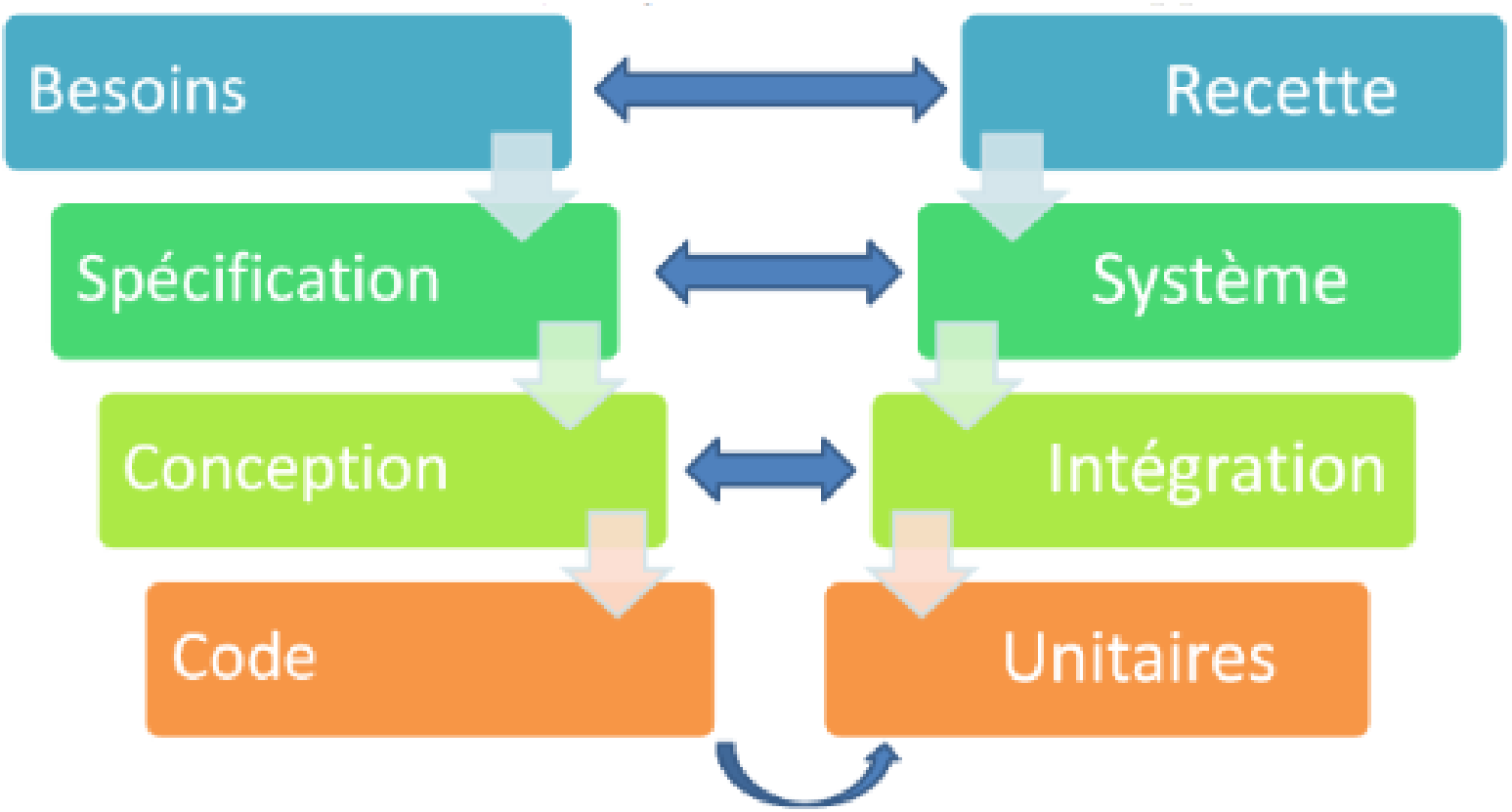
yassamine.seladji@gmail.com

3 février 2021

Introduction

- L'objectif des tests d'intégration est de tester l'interaction de modules testés unitairement.
- Ils permettent de tester une version complète et cohérente du logiciel.
- Des modules peuvent être testé unitairement, mais leur intégration peut provoquer des dysfonctionnements.

Introduction



Introduction

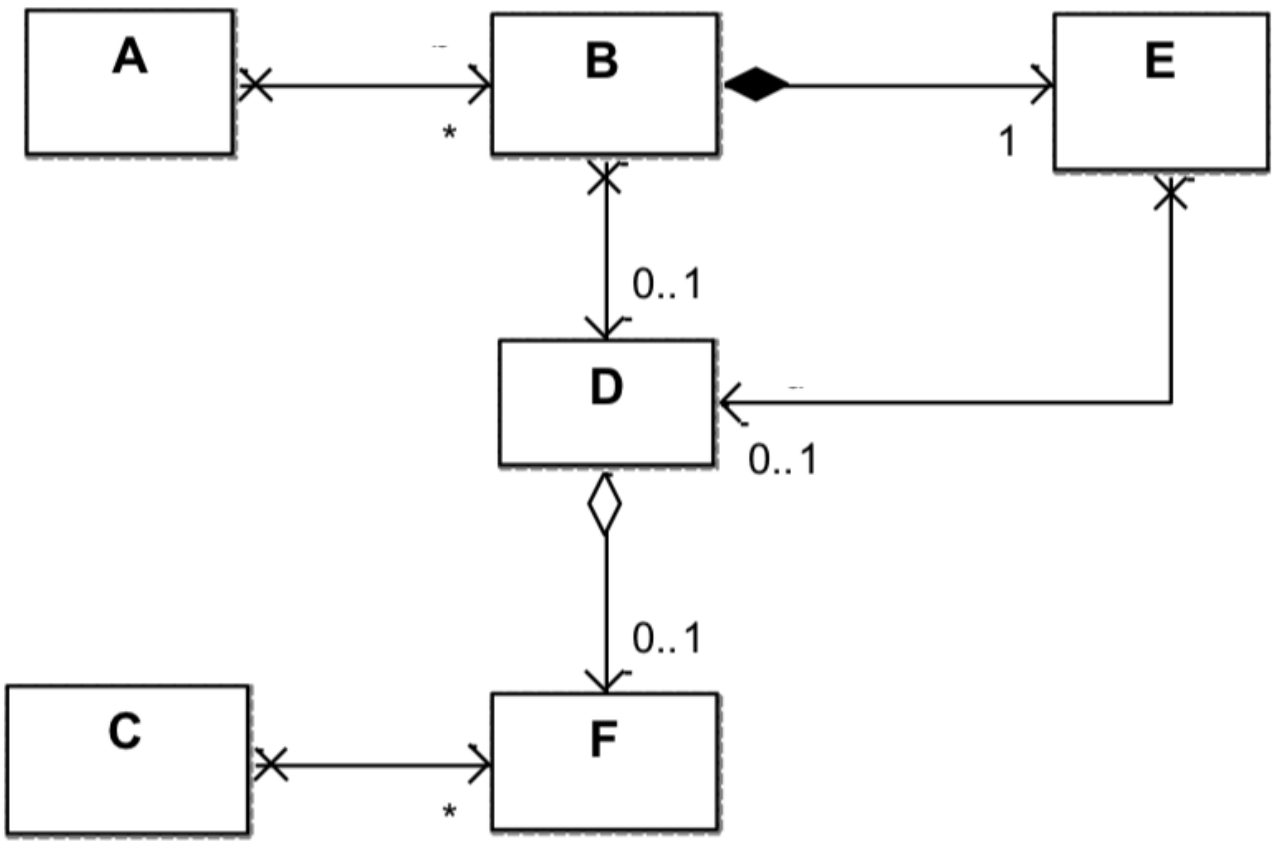
- **But** : Validation des sous systèmes logiciels entre eux.
  - Tests d'intégration Logiciel/Logiciel (interface entre composants logiciels).
  - Tests d'intégration Logiciel/Matériel (interface entre le logiciel et le matériel).
- **Quand ?** : Dés qu'un sous-système fonctionnel (module, objet) est entièrement testé unitairement.
- **Type de tests** : des interfaces.

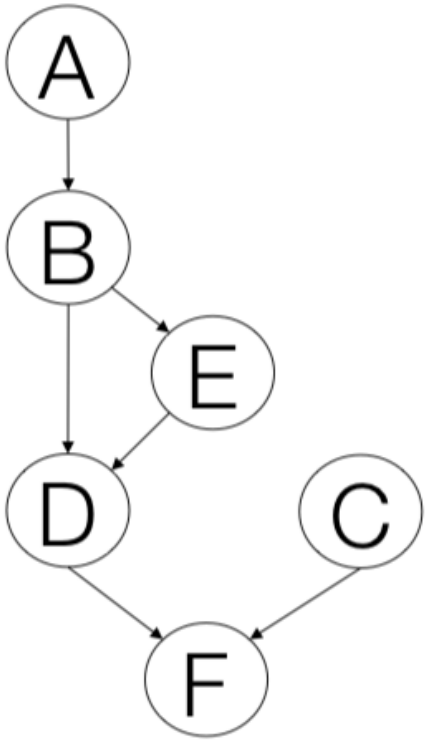
Introduction

Les tests d'intégration doivent :

- tester les interactions entre classes (modules).
- identifier les dépendances entre classes (modules).
- Modéliser les dépendances entres chaque classe (module) et son environnement.
- Choisir un ordre pour l'intégration.

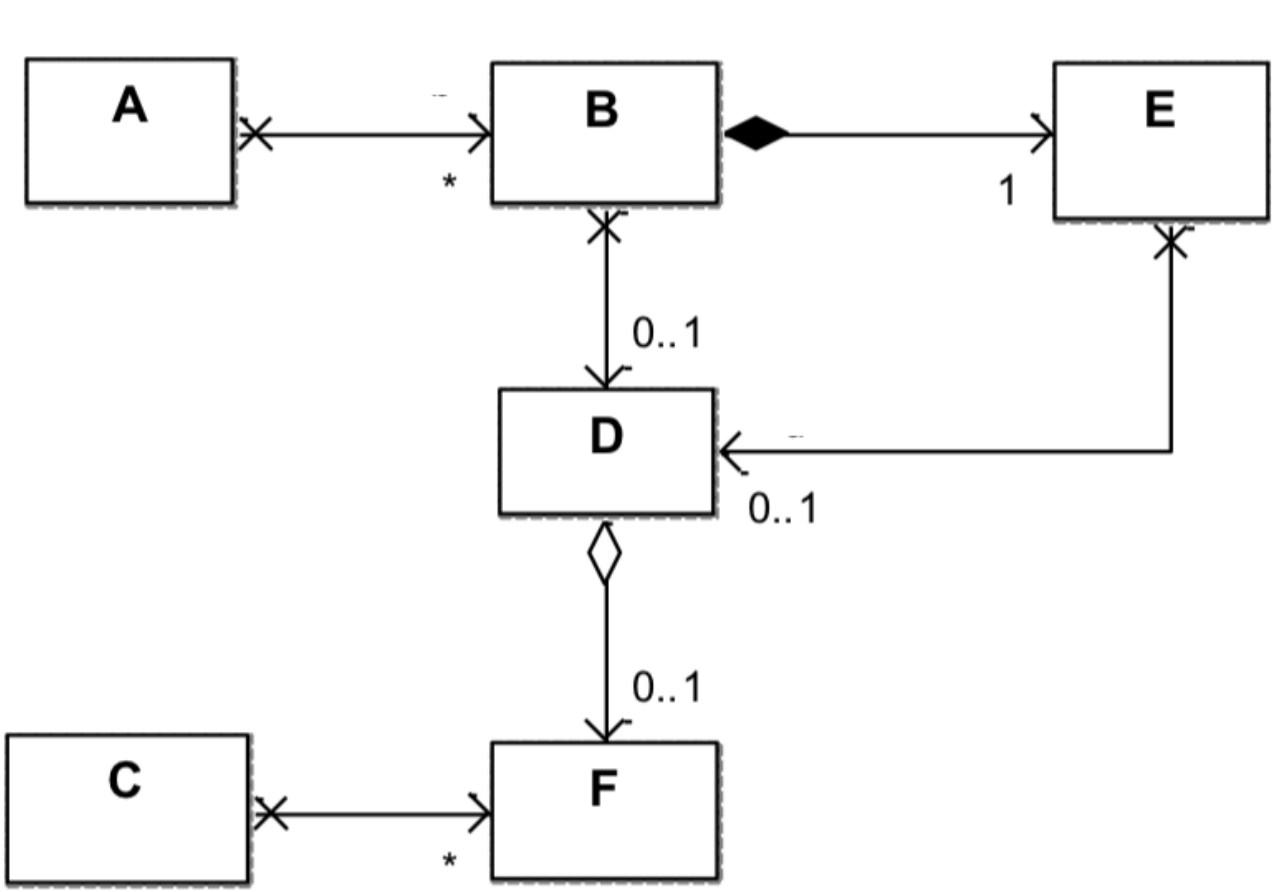
Exemple : graphe acyclique

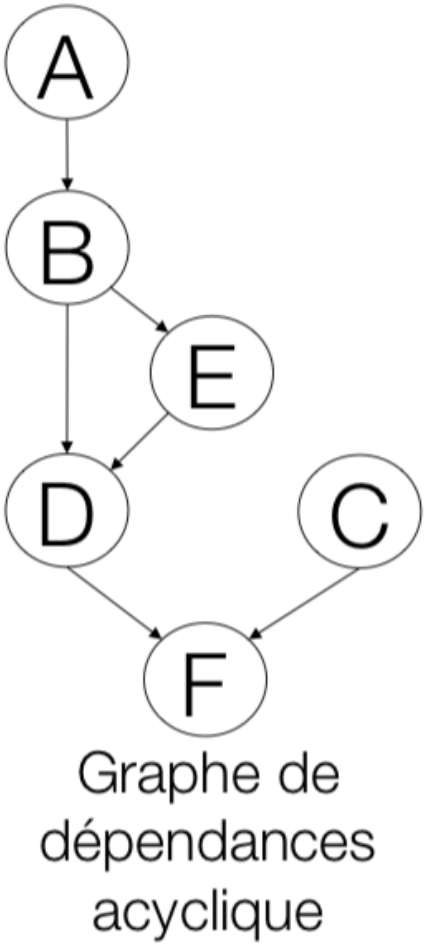




Graphe de dépendances acyclique

Exemple : graphe acyclique

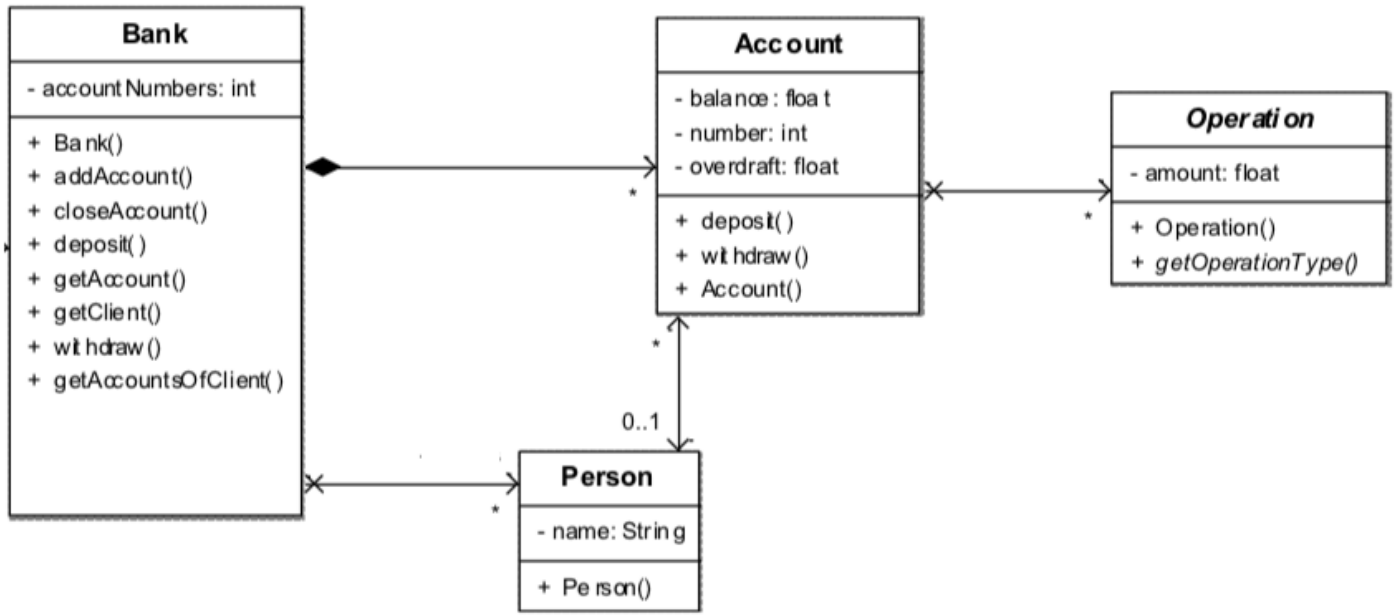




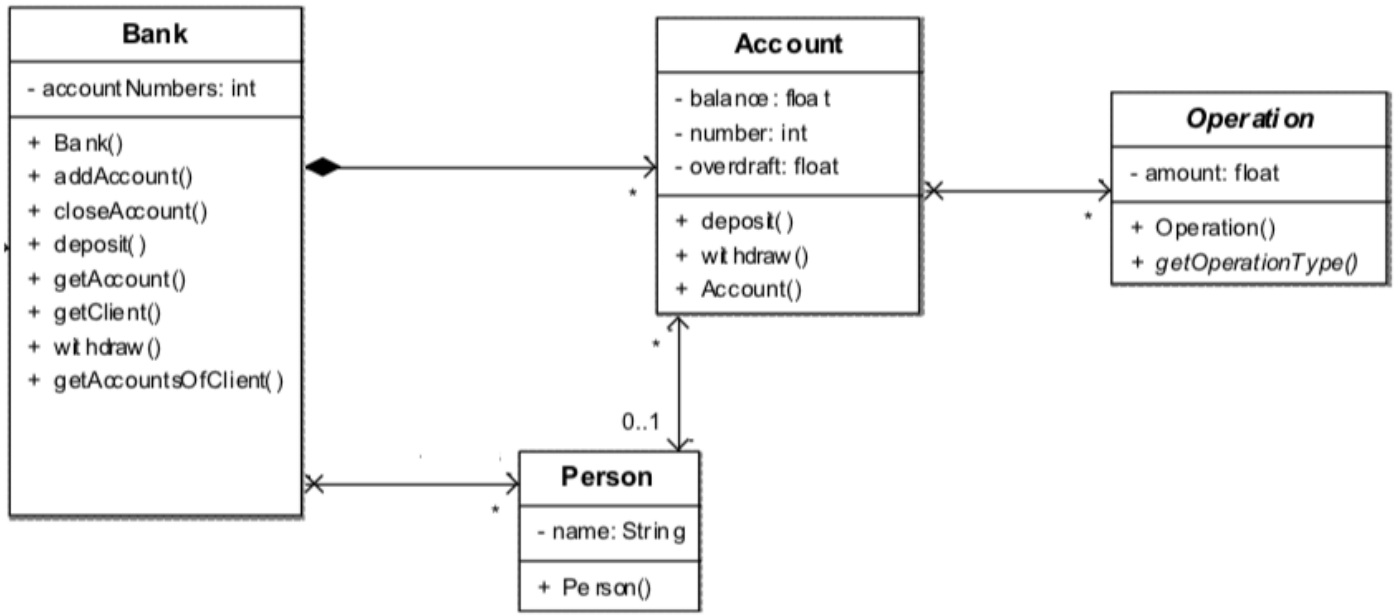
Graphe de dépendances acyclique

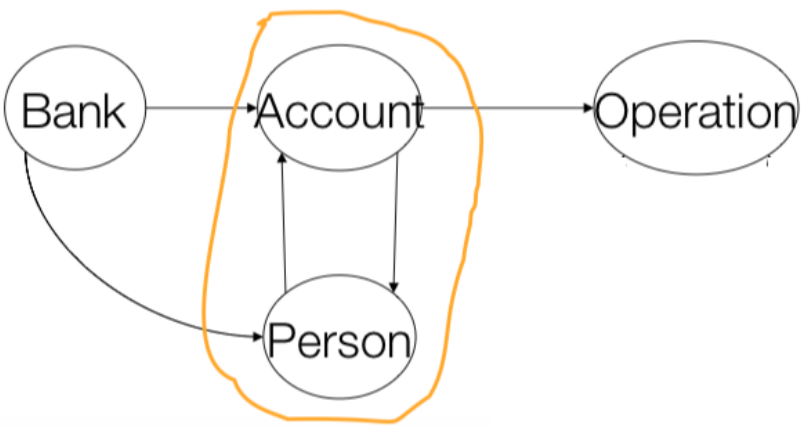
L'ordre pour le test : F,(C,D),E,B,A.

Exemple : graphe cyclique



Exemple : graphe cyclique





Exemple : graphe cyclique

- Il faut casser les cycles.
- Implémenter des simulateurs de classes, **Stub**.
- Un **Stub** possède l'interface de la classe simulée avec un comportement contrôlé.

Exemple : graphe cyclique

```
public Person getClient(String name) {
        Iterator it = this.clientsIterator();
        while (it.hasNext()){
                Person p = (Person)it.next();
                if(p.getName()==name){
                        return p;
                }

        }
        return null;
}
```

Exemple : graphe cyclique

```
public Person getClient(String name) {
        Iterator it = this.clientsIterator();
        while (it.hasNext()){
                Person p = (Person)it.next();
                if(p.getName()==name){
                        return p;
                }

        }
        return null;
}
```

Stub 1 :

```
public Person getClient(String name) {
        return null;
}
```

Stub 2 :

```
public Person getClient(String name) {
        return new Person("toto");
}
```

Exemple : graphe cyclique

```
public int addAccount(String name, float amount, float overdraft) {
        this.accountNumbers++;
        Person p = getClient(name);
        //if a client named name already exists in the bank's set of clients
        if (p!=null){
                Account a = new Account(p, amount, overdraft, accountNumbers);
                p.addAccounts(a);
                this.addAccounts(a);
        }
        //if the client does not exist, add it tp the bank's list of clients and create account
        else{
                Person client = new Person(name);
                this.addClients(client);
                Account a = new Account(client, amount, overdraft, accountNumbers);
                client.addAccounts(a);
                this.addAccounts(a);
        }
        return accountNumbers;
}
```

Exemple : graphe cyclique

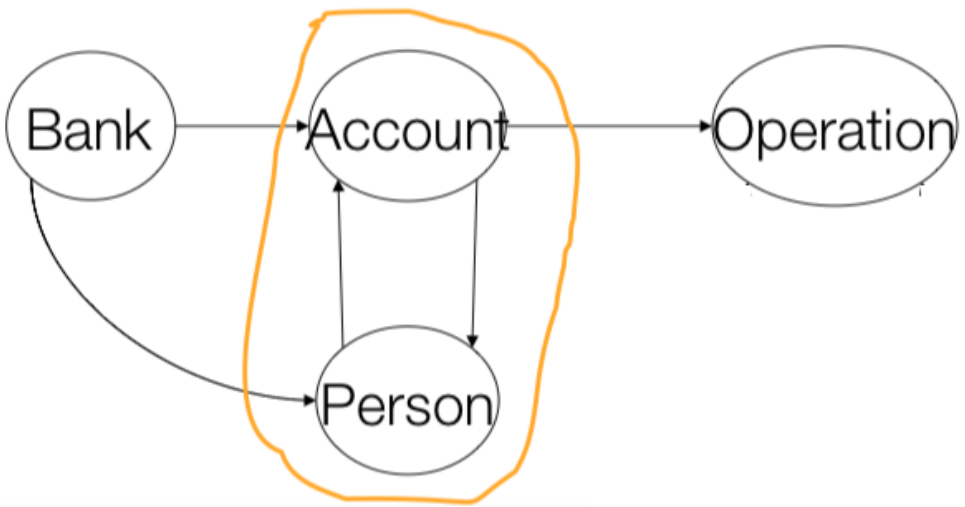
Stub 1 :

```
public int addAccount(String name, float amount, float overdraft) {
        return 10000000;
}
```

Stub 2 :

```
public int addAccount(String name, float amount, float overdraft) {
        return 1;
}
```

Exemple : graphe cyclique



- Déterminer les méthodes de **Person** utilisées par **Account**.
- Définir le Stub lié à la classe **Person**.

Exemple : graphe cyclique

```
public class Person {
  /*
   * Initializes the name of the person with the param n
   * Creates a new vector to intialize the acounts set
   */
  public Person(String n){
    name = n;
    accounts = new Vector(); }

  public String getName(){return name;}
}
```

Exemple : graphe cyclique

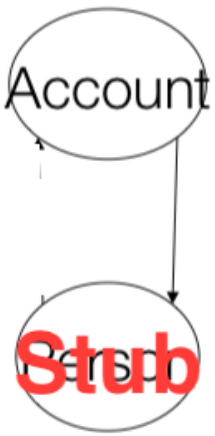
```
public class Person {
  /*
   * Initializes the name of the person with the param n
   * Creates a new vector to intialize the acounts set
   */
  public Person(String n){
    name = n;
    accounts = new Vector(); }

  public String getName(){return name;}
}
```

Stub de la classe Person :

```
public class Person {
  /*
   * Initializes the name of the person with the param n
   * Creates a new vector to initialize the accounts set
   */
  public Person(String n){ }

  public String getName(){return ("toto");}
}
```

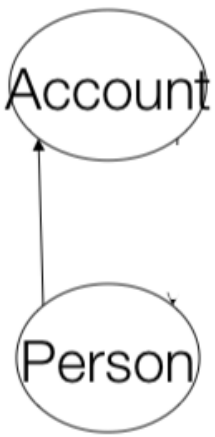
Exemple : graphe cyclique

1 tester la classe **Account** avec le **Stub** de **Person**



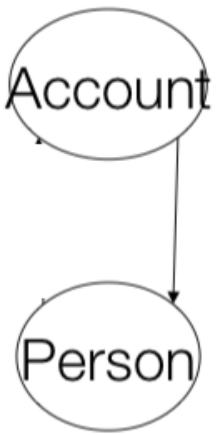
Exemple : graphe cyclique

1. Tester la classe **Account** avec le **Stub** de **Person**
2. Tester la classe **Person** avec **Account**.



Exemple : graphe cyclique

1. Tester la classe **Account** avec le **Stub** de **Person**
2. Tester la classe **Person** avec **Account**.
3. Tester la classe **Account** avec la vrai classe **Person**



Exemple industriel

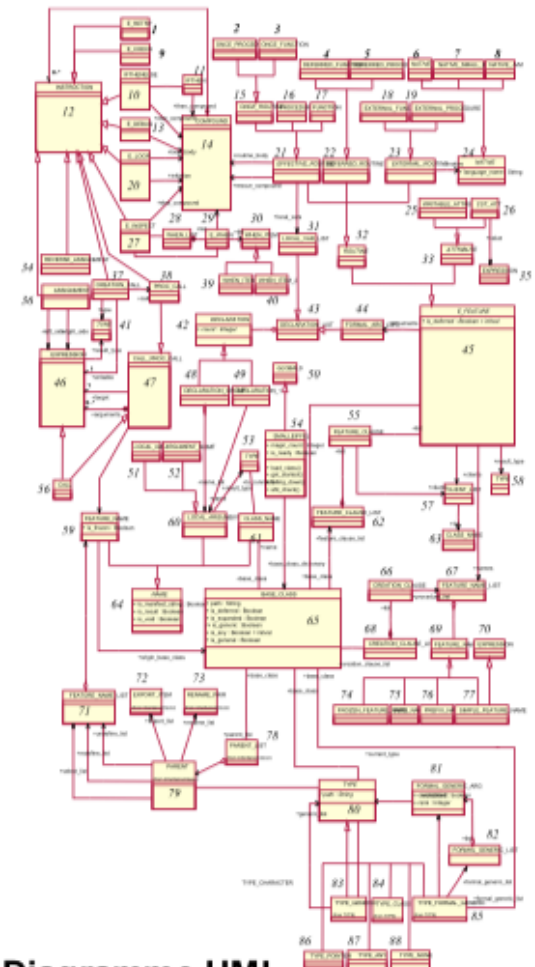
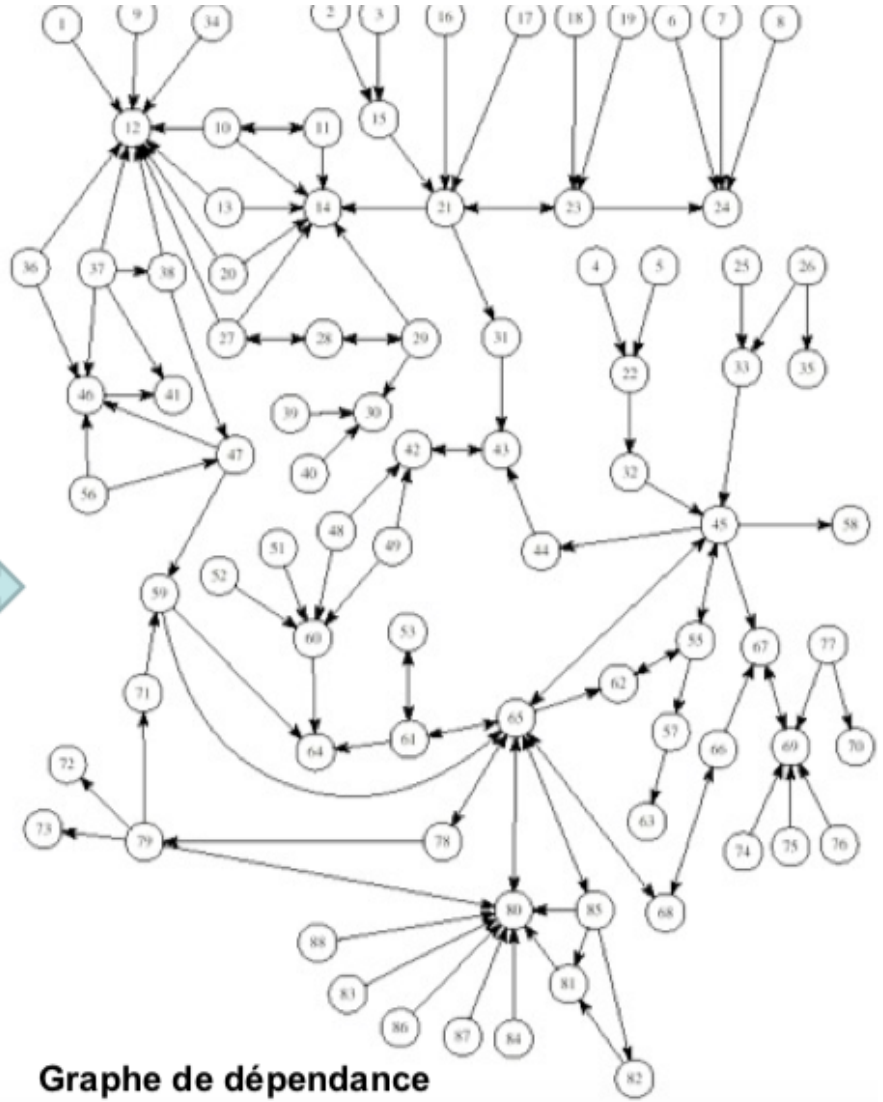


Diagramme UML



Graphe de dépendance

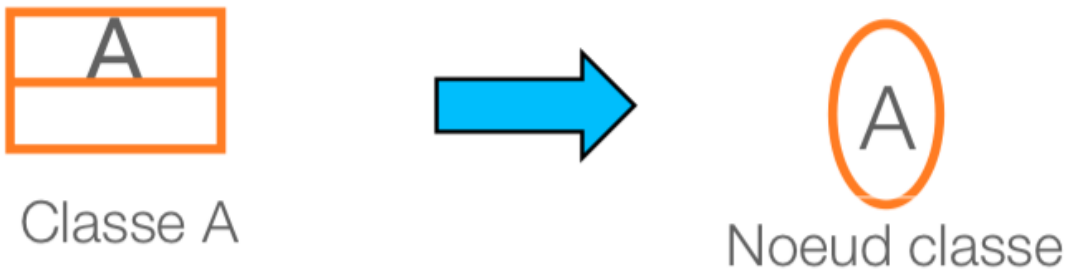
Graphe de dépendance de tests

- Définir des stratégies d'intégration.
- Construire le graphe de dépendances de tests (GDT) à partir d'UML .
- Plusieurs types de dépendances à prendre en compte :
  - Héritage.
  - client/serveur
  - classe - classe.
  - méthode - classe.

Graphe de dépendance de tests

Les types de nœuds :

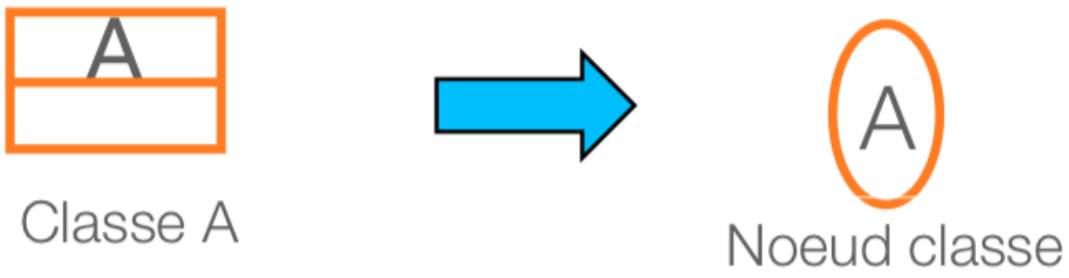
- classe :



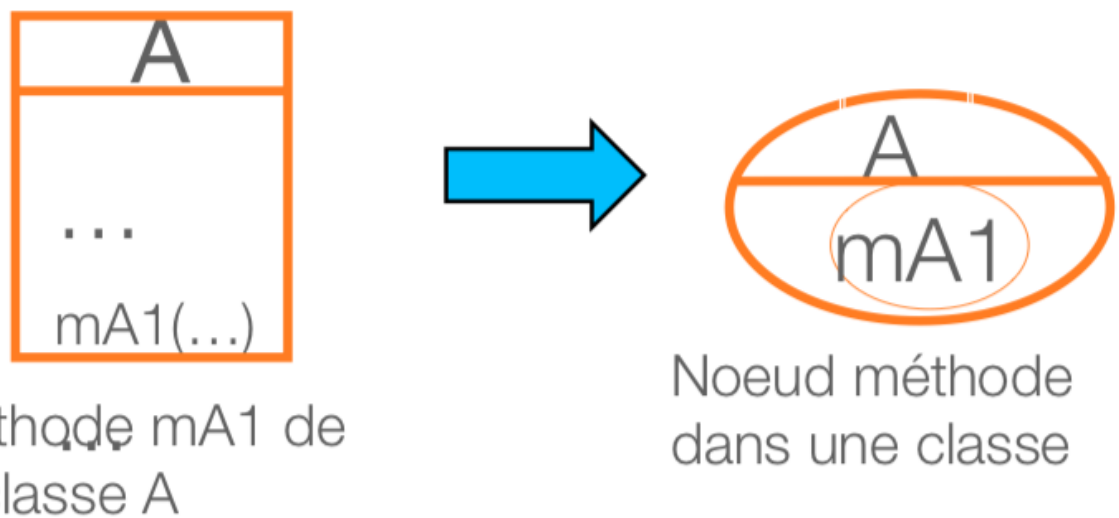
Graphe de dépendance de tests

Les types de nœuds :

- classe :



- méthodes :

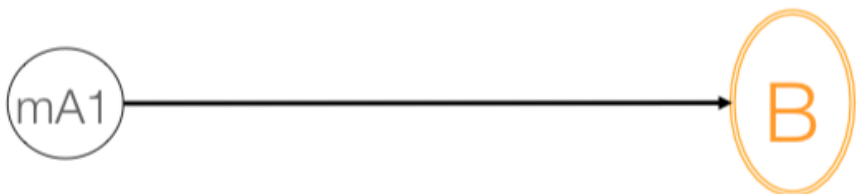


Graphe de dépendance de tests

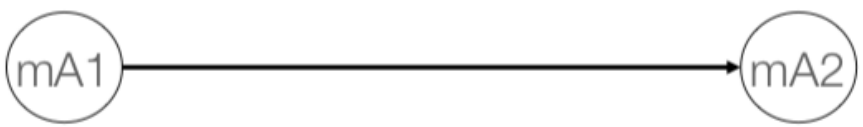
Les types d'arcs :

- classe à classe :

- méthode à classe :

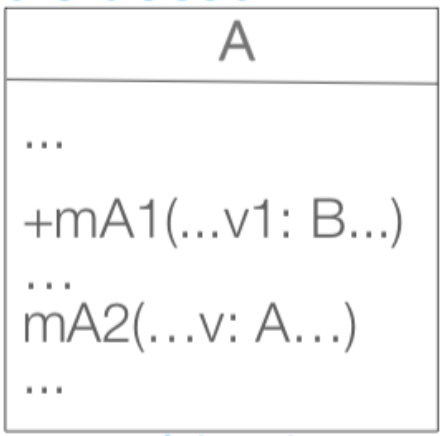


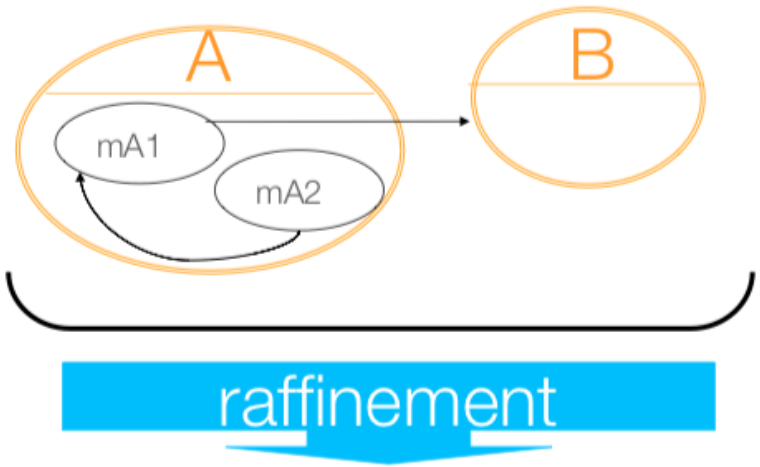
- méthode à méthode :



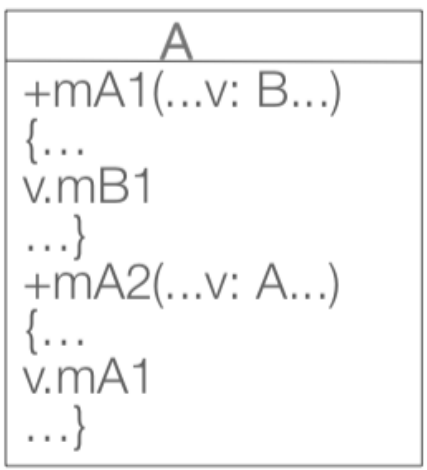
Graphe de dépendance de tests

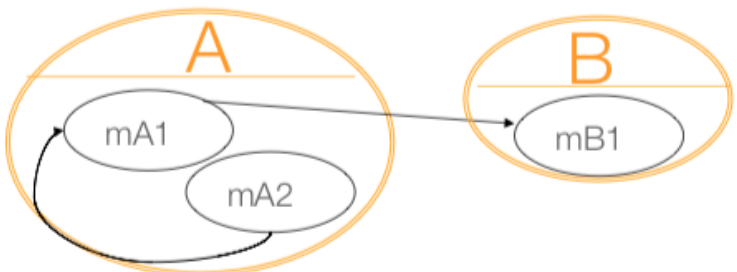
Méthode vers classe



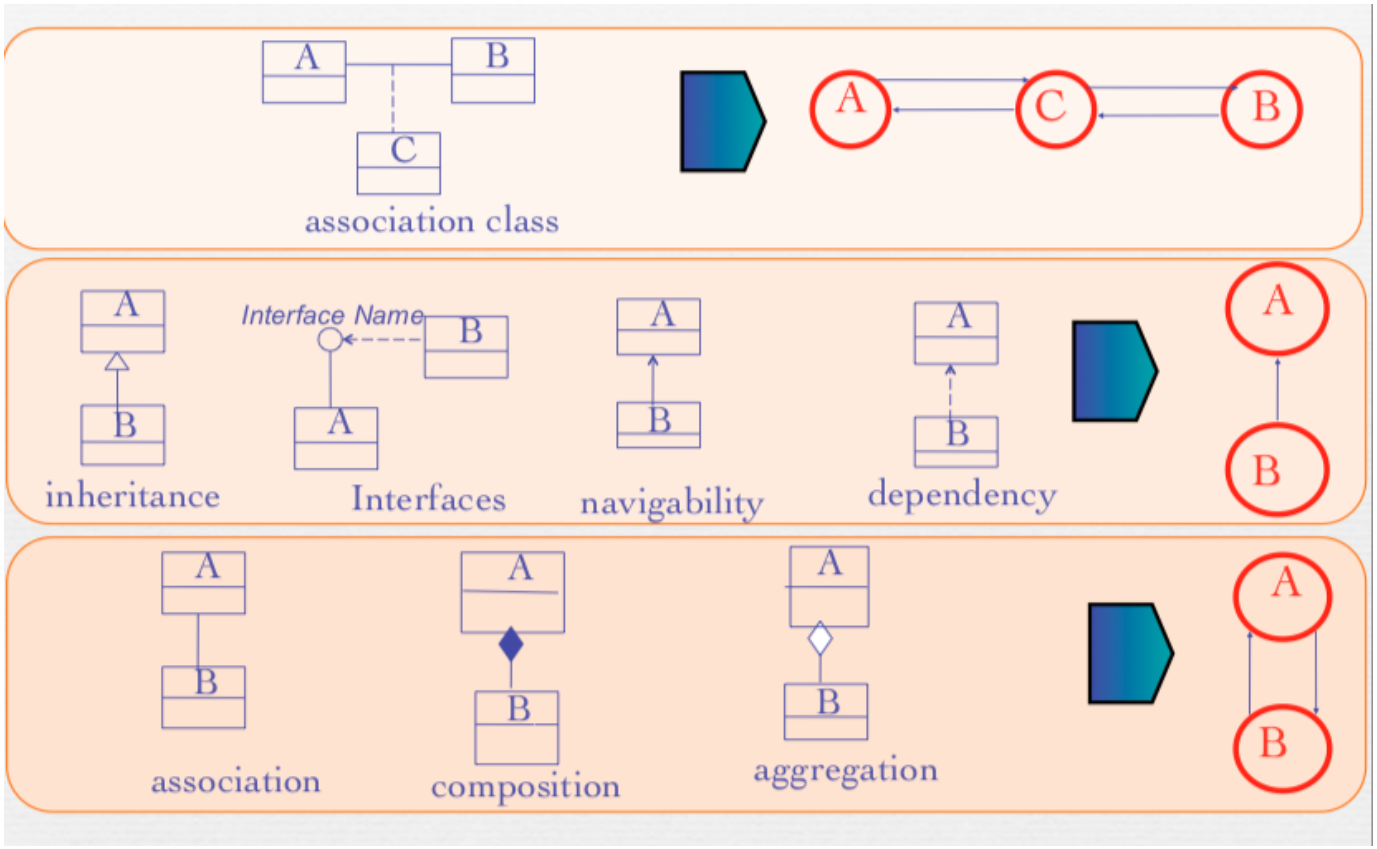


Méthode vers méthode

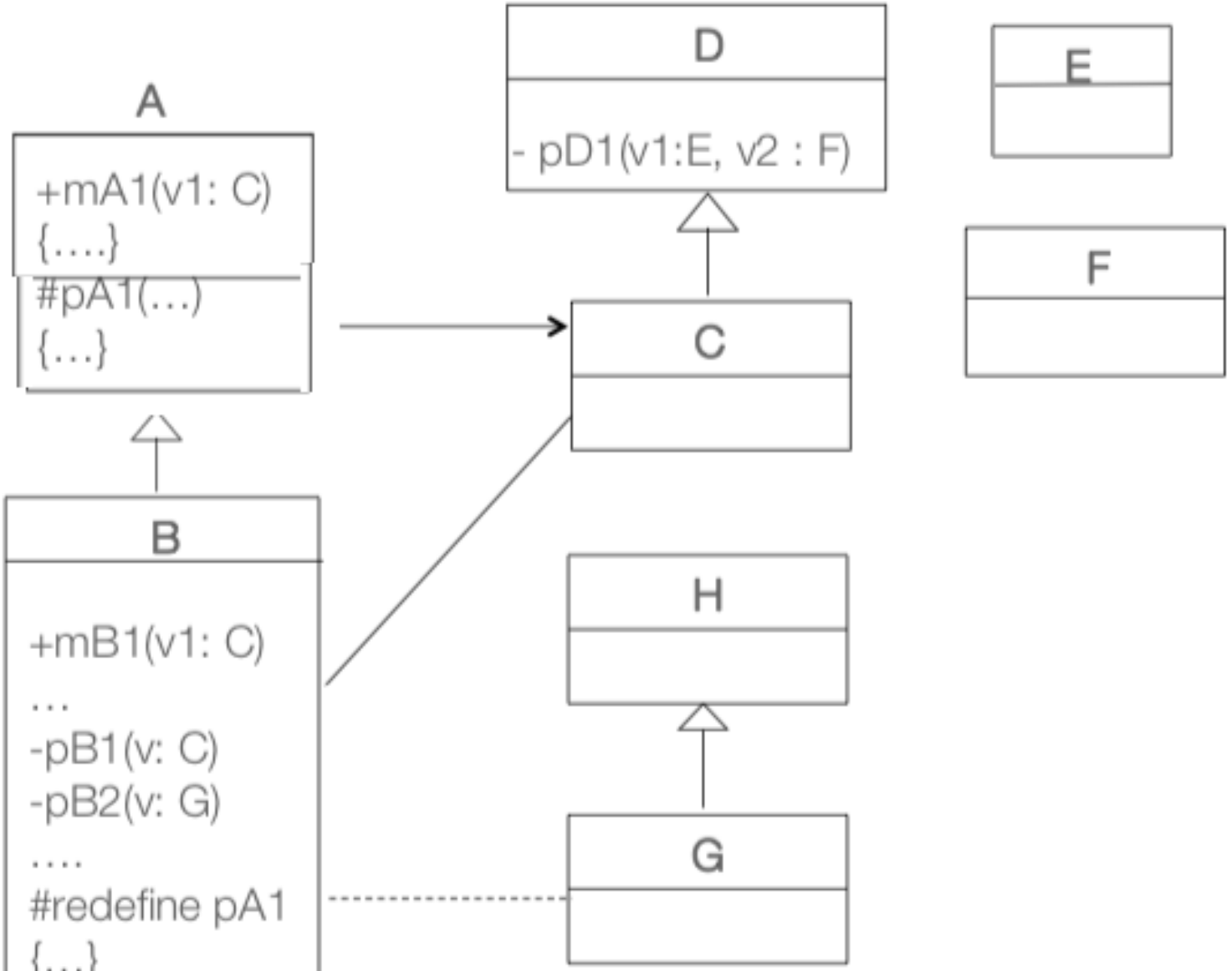




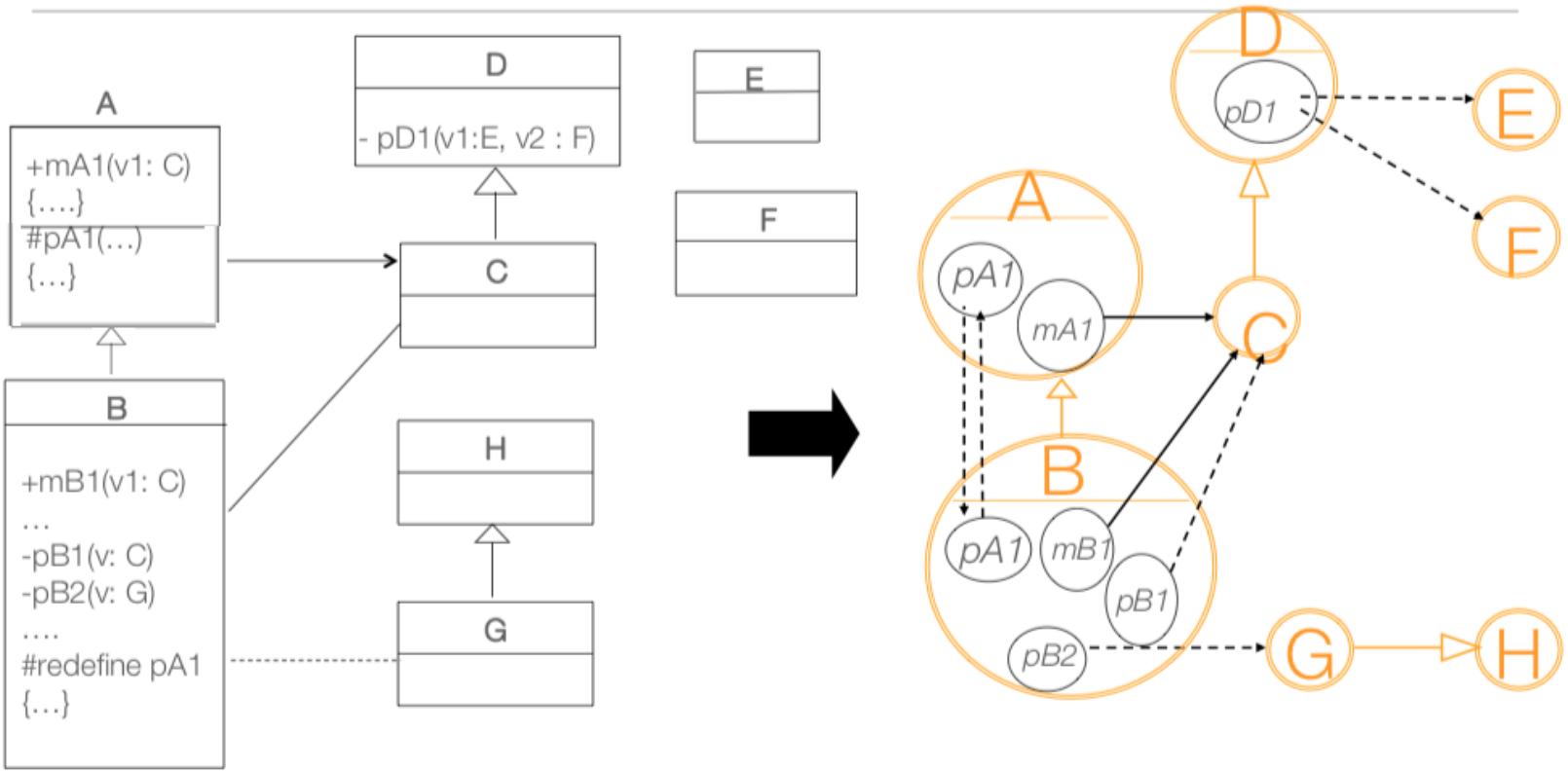
Graphe de dépendance de tests



Graphe de dépendance de tests



Graphe de dépendance de tests



Stratégies d'intégration

Comment utiliser le GDT pour choisir un ordre d'intégration ?

Stratégies d'intégration

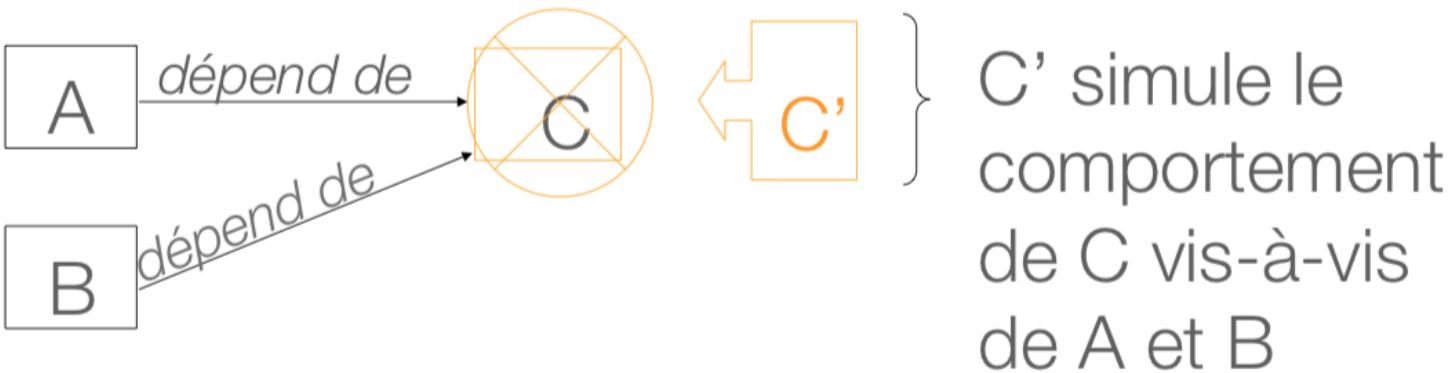
Comment utiliser le GDT pour choisir un ordre d'intégration ?

- Écrire le minimum de stub.

Stratégies d'intégration

Deux types de stub :

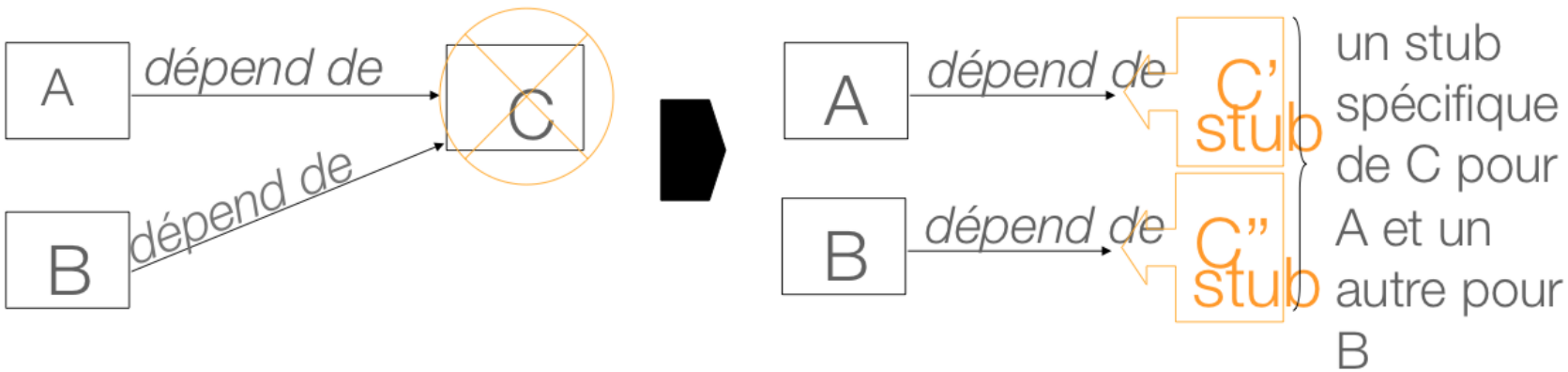
- Le stub réaliste : il simule tout les comportements.



Stratégies d'intégration

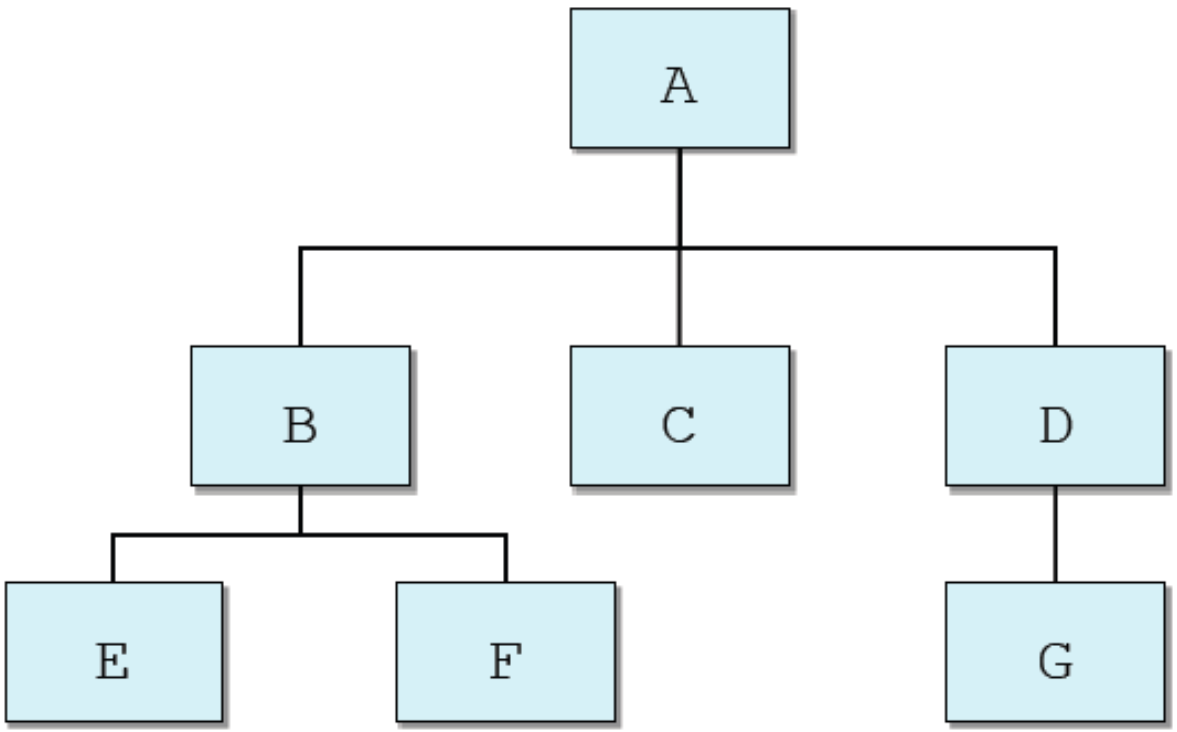
Deux types de stub :

- Le stub réaliste : il simule tous les comportements.
- Le stub spécifique : il simule le comportement d'un seul client.



Stratégies d'intégration

Exemple de hiérarchie de composants :



Stratégies d'intégration

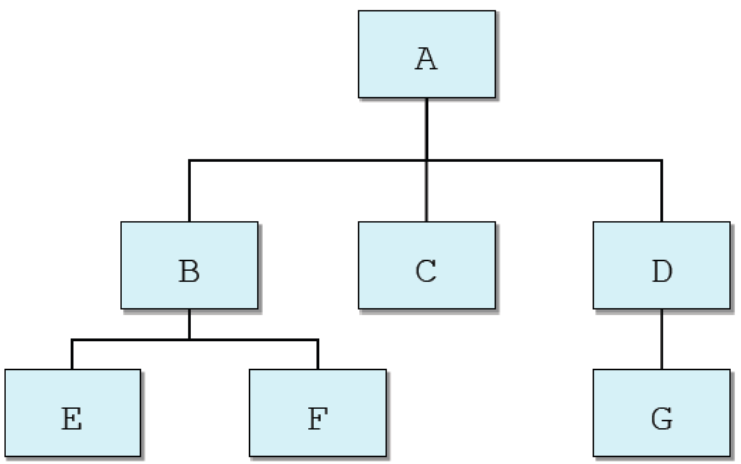
- La stratégie du big-bang : tester tous les modules ensemble.
- La stratégie top-down : descendante.
- La stratégie bottom-up : ascendante.

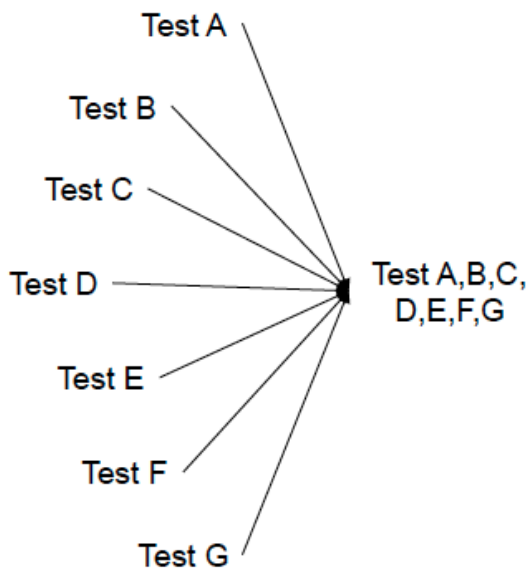
La stratégie big-bang

- Une intégration massive.
- Tous les composants a testés sont intégrés en une seul étape.

La stratégie big-bang

- Une intégration massive.
- Tous les composants a testés sont intégrés en une seul étape.



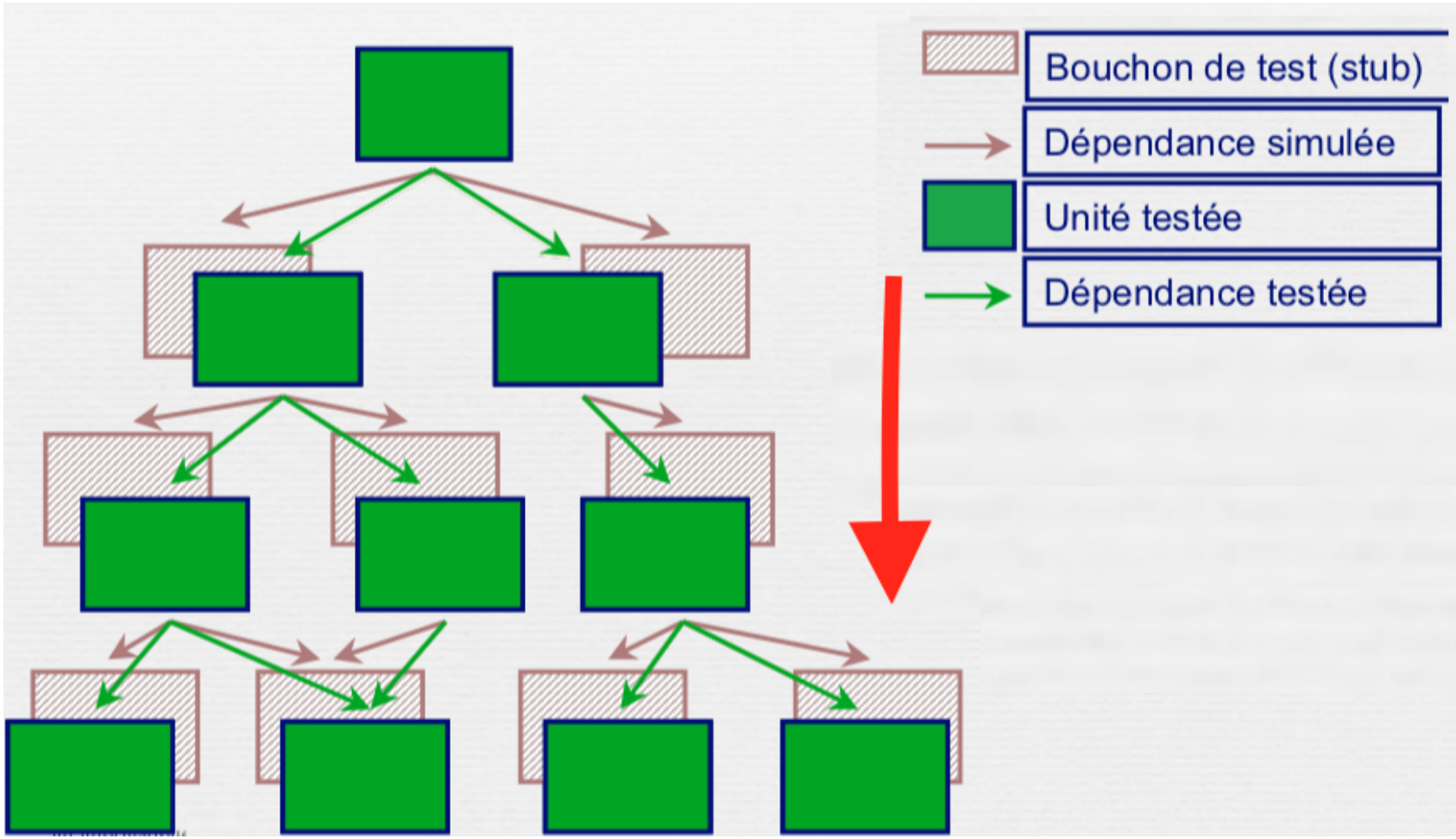


La stratégie big-bang

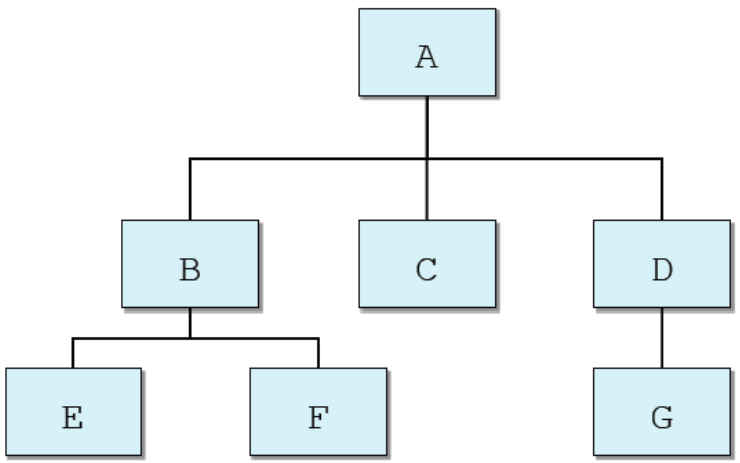
Les inconvénients :

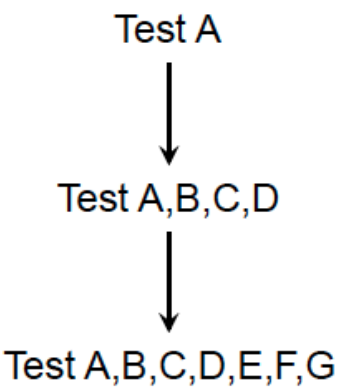
- Les causes des erreurs sont difficile à détecter.
- La complexité induit des tests manquants.
- Les tests d'intégration commencent que lorsque tous les composants ont été testé unitairement.

La stratégie descendante



La stratégie descendante





La stratégie descendante

- Utilisation de Stub.
- Détection précoce des défauts d'architecture.

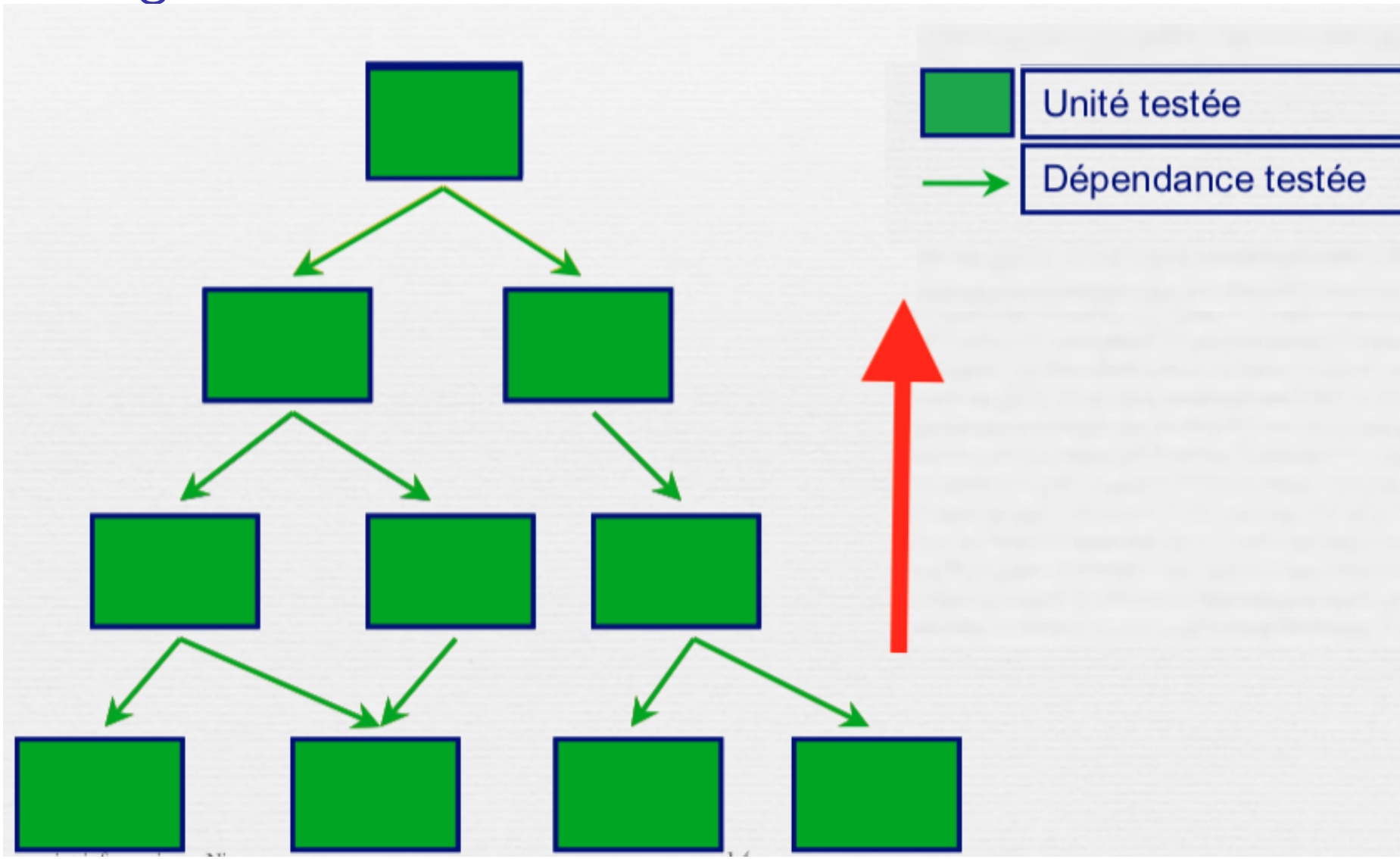
La stratégie descendante

- Utilisation de Stub.
- Détection précoce des défauts d'architecture.

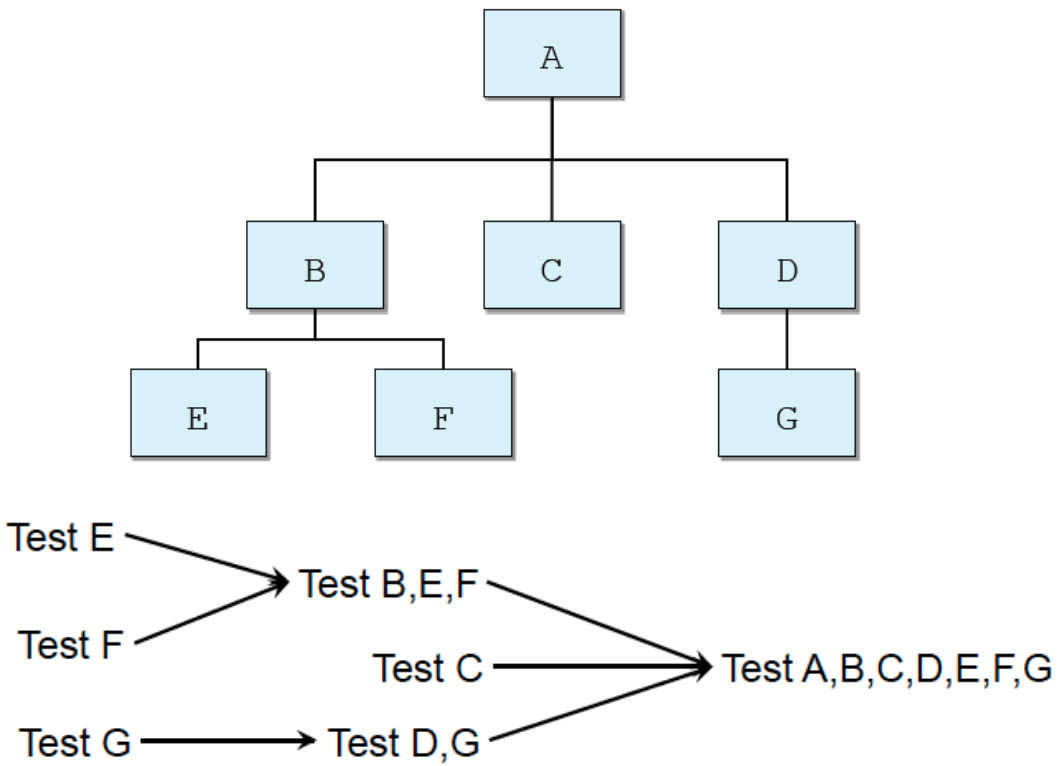
Les inconvénients :

- Plusieurs stubs à mettre en place
- Multiplie le risque d'erreurs lors de l'utilisation des stubs.

La stratégie ascendante



La stratégie descendante



La stratégie ascendante

- Pas de stub à mettre en place.
- Les jeux de tests sont définit facilement.
- La démarche est naturelle.

La stratégie ascendante

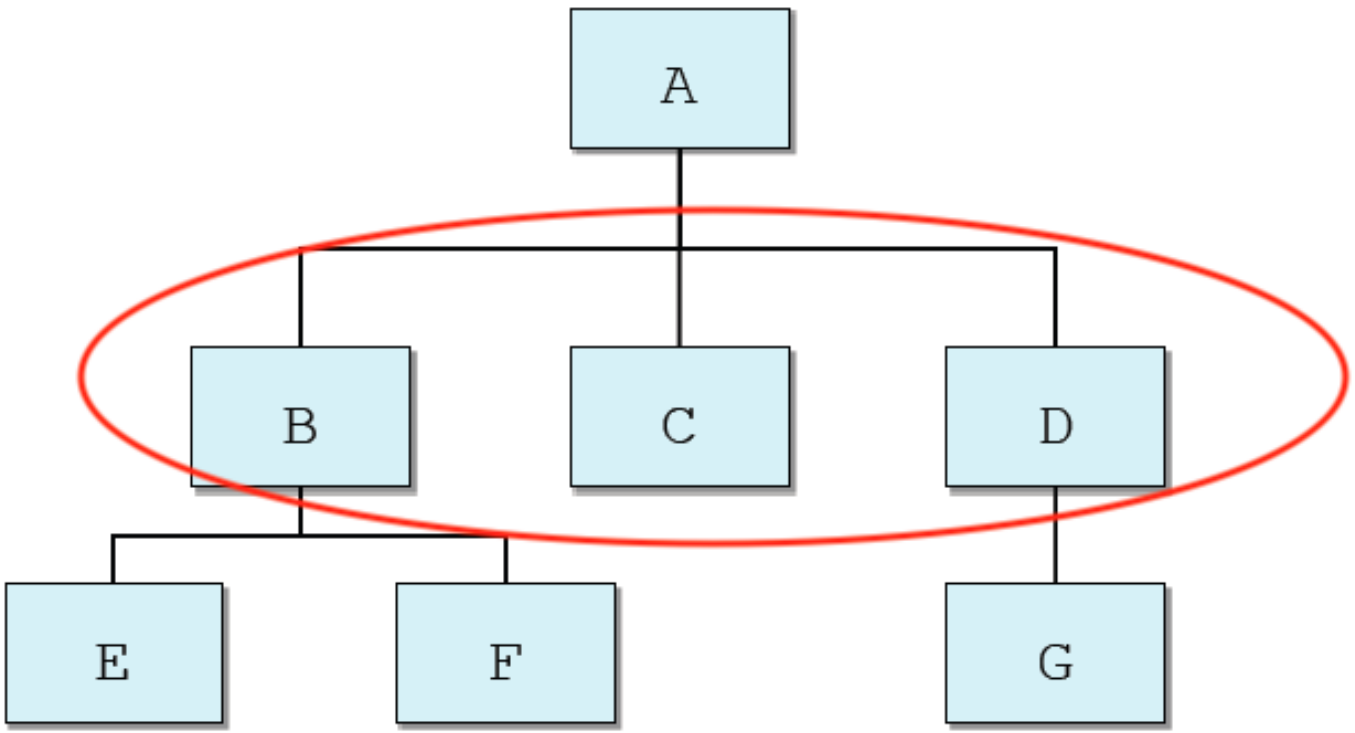
- Pas de stub à mettre en place.
- Les jeux de tests sont définit facilement.
- La démarche est naturelle.

Les inconvénients :

- Les tests dépendent de la disponibilités des composants.

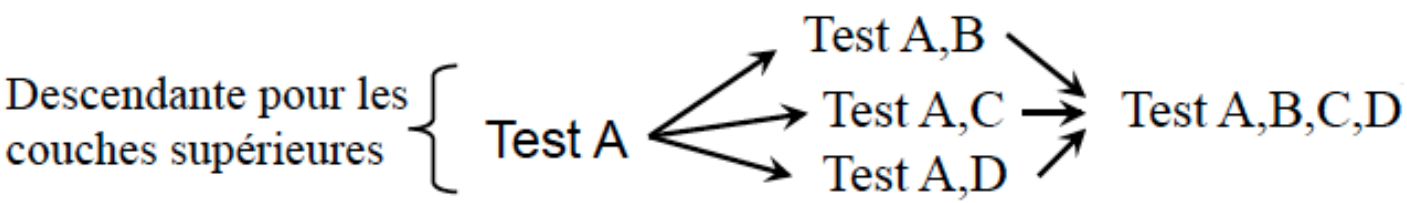
La stratégie sandwich

- Choisir une couche cible.



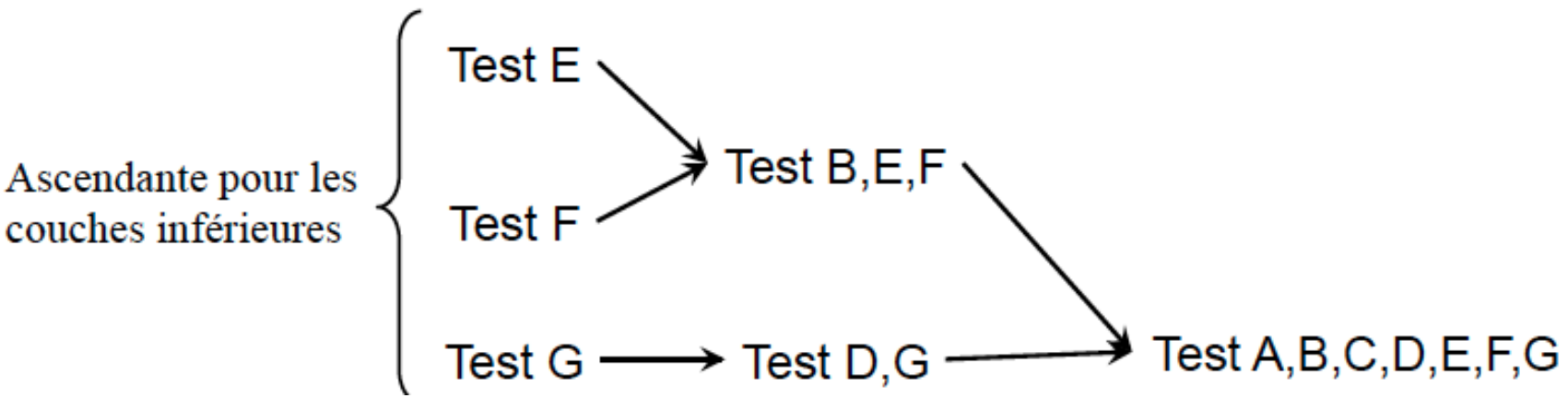
La stratégie sandwich

- Choisir une couche cible.
- La stratégie descendante est choisie pour les couches supérieures.

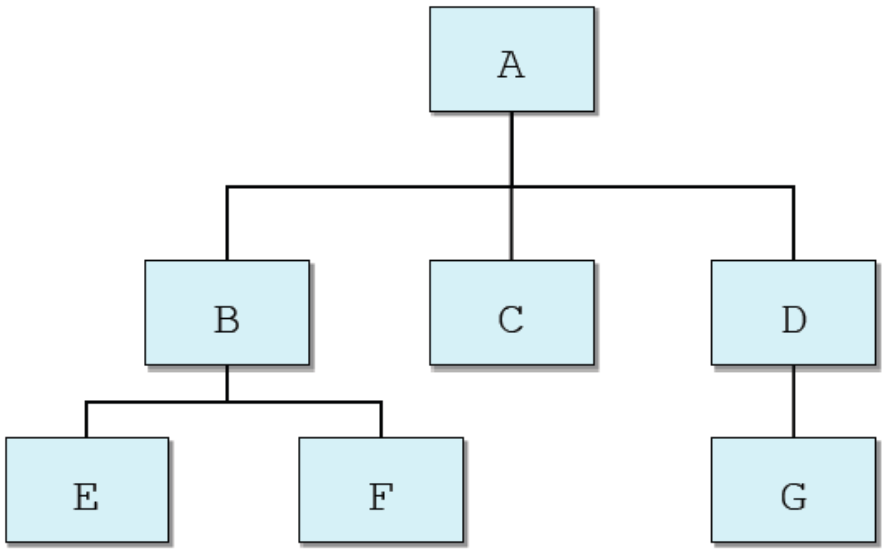


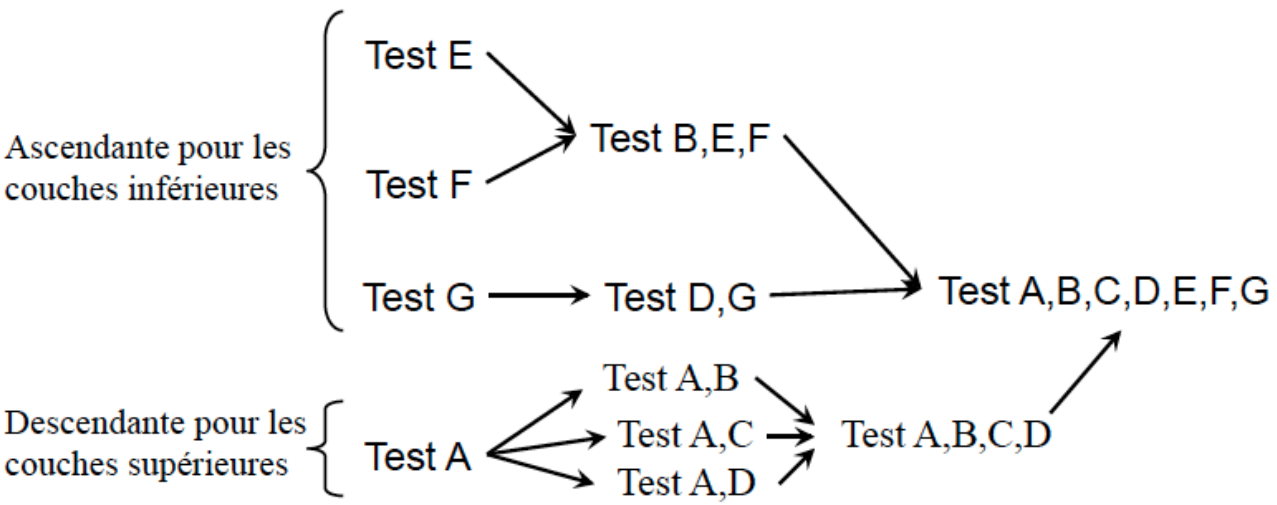
La stratégie sandwich

- Choisir une couche cible.
- La stratégie descendante est choisie pour les couches supérieures.
- La stratégie ascendante pour les couches inférieures.



La stratégie sandwich



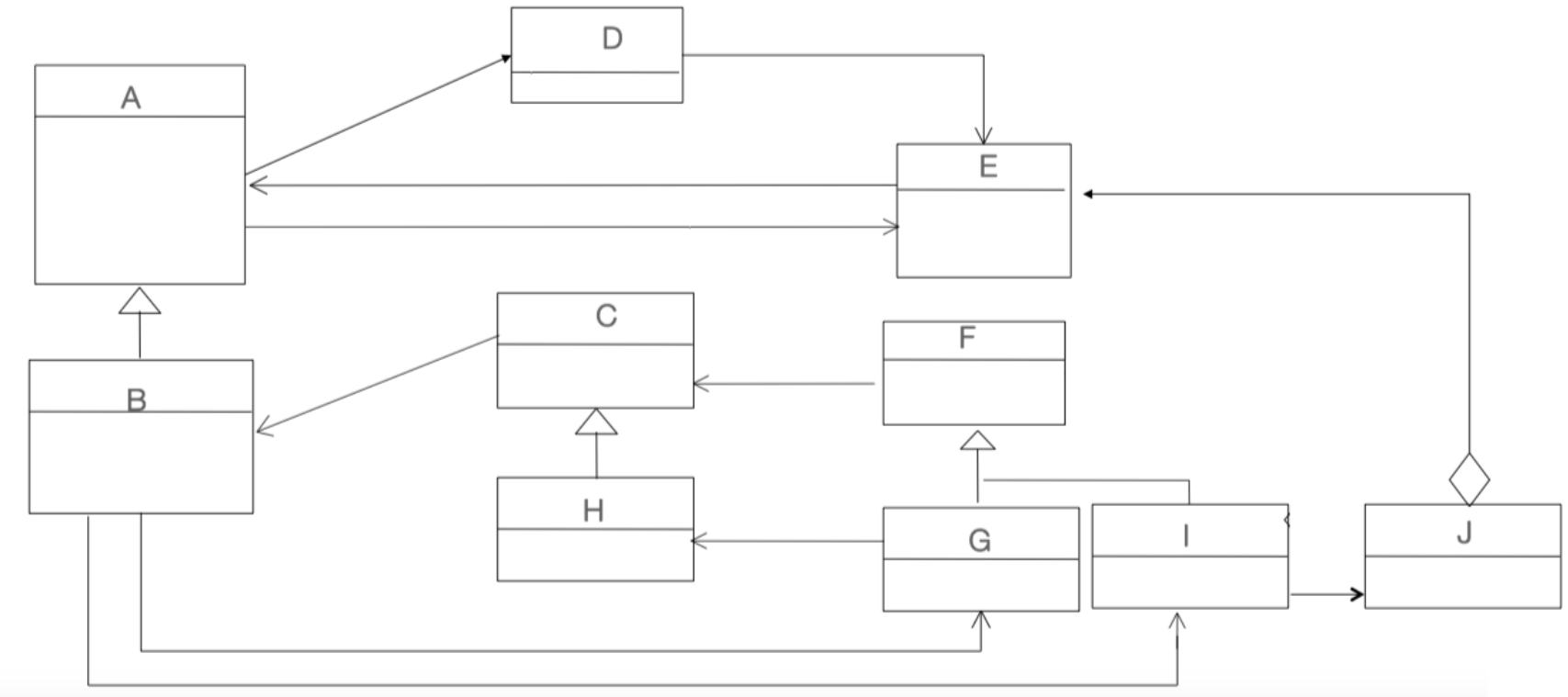


La stratégie sandwich

- Avantage :
  - Les premiers composants terminés sont intégrés en premier.
- Inconvénient :
  - Ne teste pas tous les composants de la couche cible.
  - Le composant C de notre exemple n'est pas testé.

La stratégie d'intégration : exercice

Proposer une stratégie d'intégration :



Mock et stub

Mock et stub

- Un mock est un type de stub.
- Un mock est un objet qui simule le comportement d'objet réel comme : une BDD, un web service ..
- Un mock est utilisé pour simuler le comportement d'une classe ou interface.

Mock et stub

- Un mock permet de tester :
  - une classe en isolation.
  - les interactions avec l'environnement.

Mock et stub

- Tester une classe en isolation (Stubbing) :
  - casser les cycles de dépendance entre classes.
  - remplacer le comportement d'une des classes dans le cycle par une simulation.

Mock et stub

- Tester les interactions entre classes (interaction testing) :
  - utiliser les diagrammes UML de sequence.
  - vérifier si les échanges de messages entre deux classes qui interagissent sont conformes aux diagrammes de séquences correspondants.

Mock et stub

- Mettre en place un mock :
  - Création du mock.
  - Définir les comportements du mock.
  - Écrire et exécuter les tests.
  - Après l'exécution des tests, vérifier le comportement du mock pour voir si les interactions attendues ont bien eu lieu.

Mock et stub

Les langages orienté objet possèdent leurs propre framework pour utiliser des mocks dans les tests.

- Java : EasyMock, Mockito ....
- C♯ : Moq, NSubstitute, Rhino Mocks ....
- C++ : CppUMock, FakeIt ....
- Python : unittest.mock, Mox, Mocker ....

Mockito pour Java

Afin d'utiliser le framework Mockito, il suffit d'ajouter la librairie Mockito.zip au classpath de votre projet.
Pour créer un mock :

- définir la classe de l'objet mocké :
  Maclasse objetMock = **mock**(MaClasse.class).
- utiliser l'annotation avant la déclaration de l'objet mocké :
  @Mock
  MaClasse objet ;

Mockito pour Java

Pour Définir le comportement du mock :

- Fixer la valeur de retour d'une méthode sur l'objet
  `when(objetMock.operationOp()).thenReturn(someResult);`
- Rétablir le comportement d'une méthode :
  `when(objetMock.operationOp()).thenCallRealMethod();`
- Changer le comportement d'une méthode void :
  `when(objetMock.operationOp(Mockito.anyString)).DoNothing();`
- Lancer une exception lors de l'appel de la méthode :
  `when(objetMock.operationOp()).thenThrow(new IllegalArgumentException);`

Mockito pour Java

- Écrire les tests avec Junit.

Mockito pour Java

- Vérifier le comportement du mock :
  verify(objetMock).methode(valeurArgument) ;
- Vérifier l'ordre des invocations des objets :
  InOrder.verify(objetMock) ;

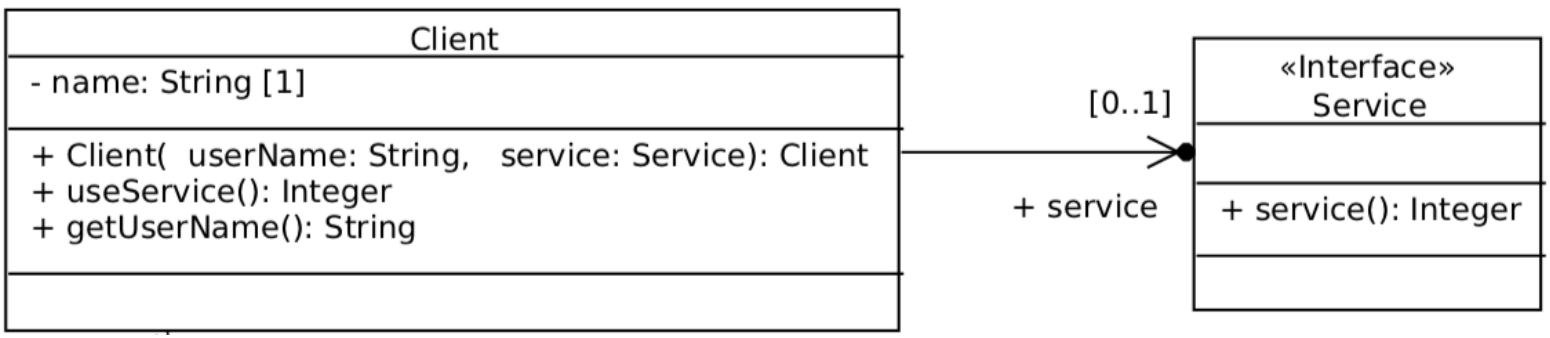
Mockito pour Java

Utiliser l'annotation @Spy

- Instancier l'objet d'une classe et non une interface.
- Pas d'obligation pour redéfinir le comportement des méthodes de l'objet.

Mockito pour Java

Tester une classe en isolation (Stubbing)



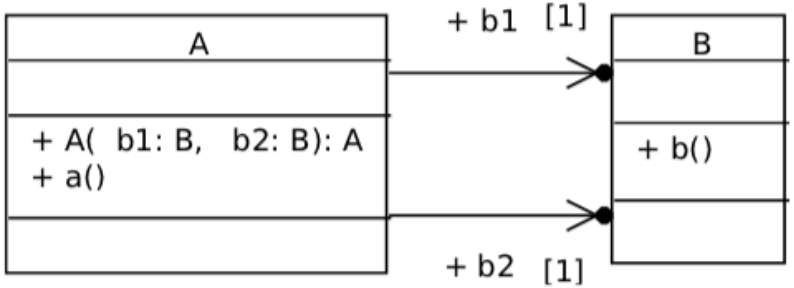
La méthode *useService()* fait appel à *service* et retourne la valeur obtenue plus 10.

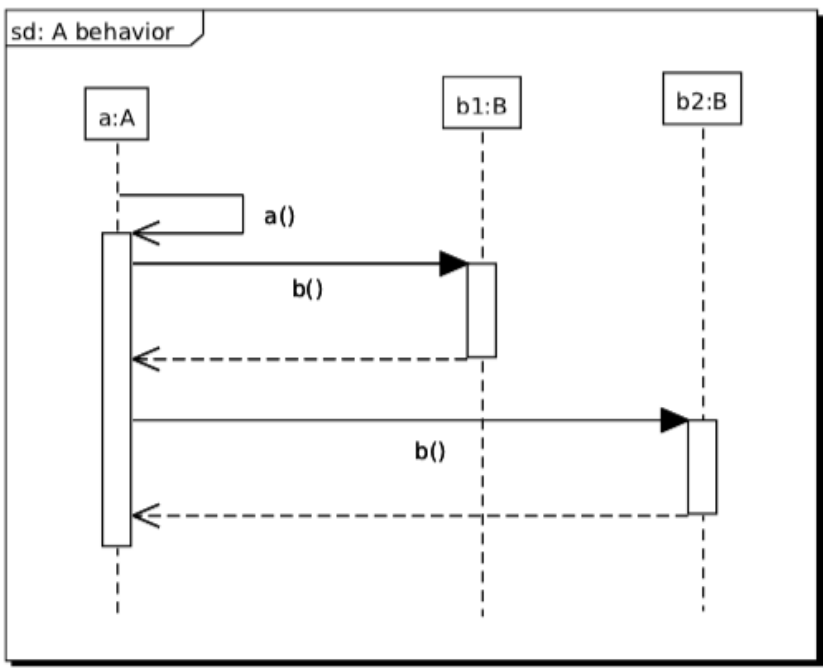
Mockito pour Java

```java
import org.junit.Test;
import static org.mockito.Mockito.*;
import static org.junit.Assert.*;
...

public class ClientTestWithStub {

	@Test
	public void testUseService() {

		// Preparing the context -- we stub the Service instance
		Service mockService = mock(Service.class);
		when(mockService.service()).thenReturn(1327);
		Client client = new Client("John",mockService);

		// Calling the tested operation
		int result = client.useService();

		// Oracle
		assertEquals(result,1337);

	}
}
```

Mockito pour Java

Tester les interactions entre classes (interaction testing)





Mockito pour Java

Tester les interactions entre classes (interaction testing)
Vérifier la méthode appelée par le mock :

- Créer les mocks :
  B b1 = mock(B.class) ;
  B b2 = mock(B.class) ;
  A a = new A(b1,b2) ;

Mockito pour Java

Tester les interactions entre classes (interaction testing)
Vérifier la méthode appelée par le mock :

- Créer les mocks :
  B b1 = mock(B.class) ;
  B b2 = mock(B.class) ;
  A a = new A(b1,b2) ;
- Appeler la méthode à tester : a.a() ;

Mockito pour Java

Tester les interactions entre classes (interaction testing)
Vérifier la méthode appelée par le mock :

- Créer les mocks :
  B b1 = mock(B.class) ;
  B b2 = mock(B.class) ;
  A a = new A(b1,b2) ;
- Appeler la méthode à tester : a.a() ;
- Vérifier si *b1.b()* est appelée durant l'exécution de *a()* pour tester si b1 est utilisé correctement par *a*.
  verify(b1).b() ;

Mockito pour Java

On peut aussi vérifier l'ordre d'appel des méthodes par le mock.
Vérifier si b1.b() est appellée avant b2.b() dans a() :

```
InOrder mockAvecOrdre = inOrder(b1,b2);
a.a();
mockAvecOrdre.verify(b1).b();
mockAvecOrdre.verify(b2).b();
```

Mockito pour Java

Les limites du framework :

- Impossible de mocker une classe ou une interface final.
- impossible de mocker une méthode statique ou privée.
- impossible de mocker les méthodes equals() et hashcode() (Mockito redéfinit et dépend fortement de ces 2 dernières).

Conclusion

- Les Tests d'intégration permettent de tester les interactions entre composants.
- Le choix d'ordre d'intégration est important. Limiter le nombre de Stub à écrire.
- Les mocks sont utilisés pour simuler le comportement d'un objet sans avoir à l'instancier.