IUT Nantes
Pôle Sciences et technologie

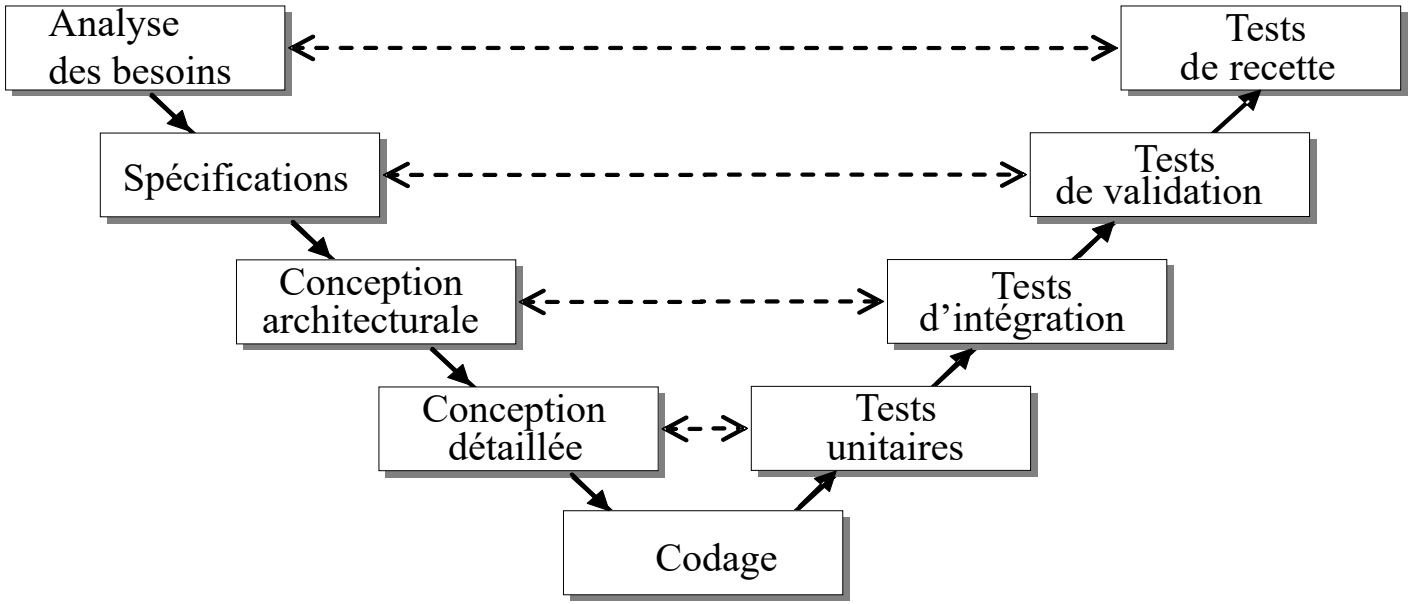
Nantes Université

R4.02 - Qualité de développement 3
CM4 – Test d'intégration

Jean-Marie Mottu - BUT info 2 – IUT Nantes
Le Traon – Baudry

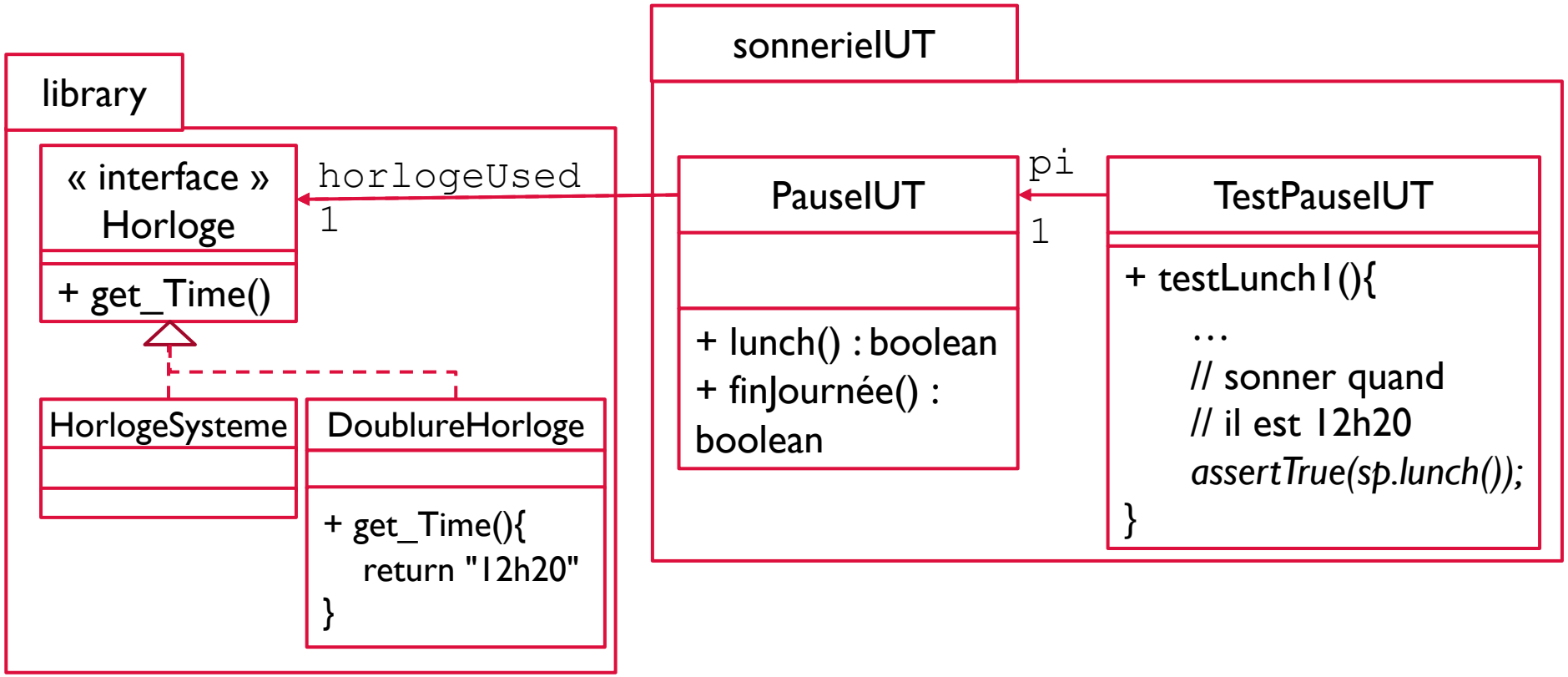
Etapes de test

- Le test d'intégration pour tester l'intégration des unités les unes avec les autres



L'intégration au-delà des dépendances non contrôlables

- On a déjà vu comment permettre le test unitaire en gérant les dépendances (injection de doublures) :



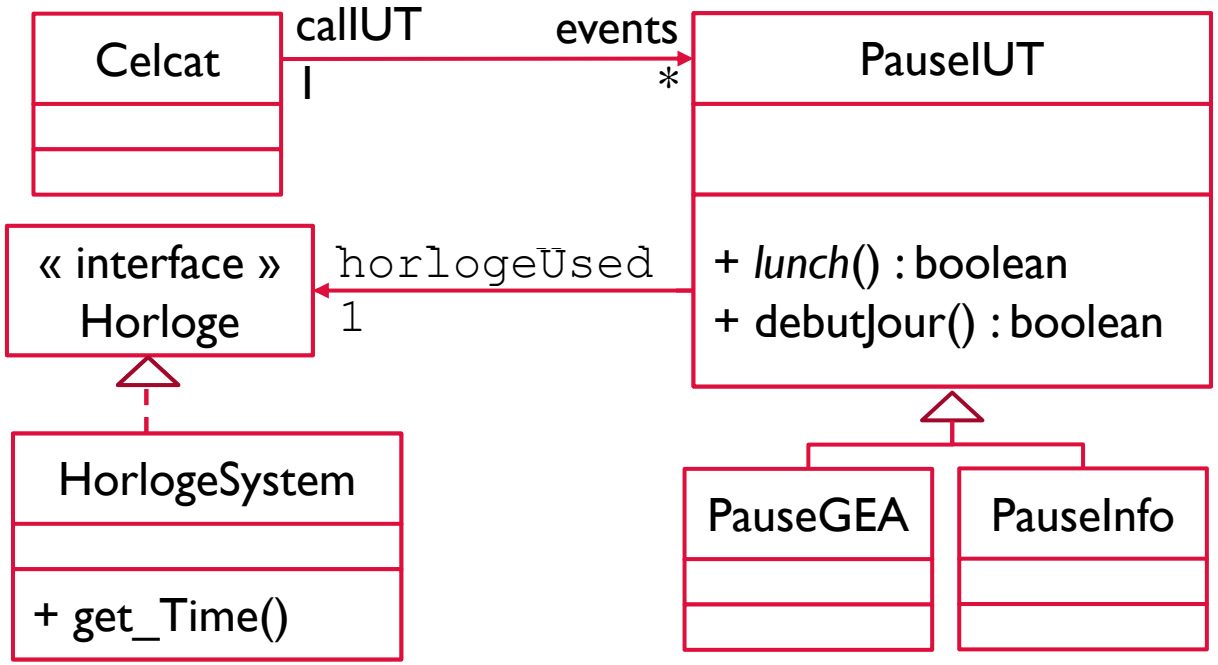
L'intégration au-delà des dépendances non contrôlables

- Le test d'intégration sert à vérifier des assemblages d'unités qui ont nature à travailler ensemble,
  - au-delà de dépendances non contrôlables : sous-traitance, dépendances externes, non maîtrisables (temps, aléatoire, rapidité, coût), etc.,
  - des dépendances font partie intégrante de l'assemblage du système développé

- Il faudra tester les unités avec leurs dépendances et pas seulement avec les doublures de ces dépendances
  - quand elles auront été
    - développées
    - Testées elles-mêmes

L'intégration au-delà des dépendances non contrôlables

- Exemple



Approches pour le test d'intégration

- Approche Big Bang
- Approches progressives incrémentales
  - Bottom up
  - Top down
  - Mixte
- Approche d'ordonnancement de graphe de dépendance de test

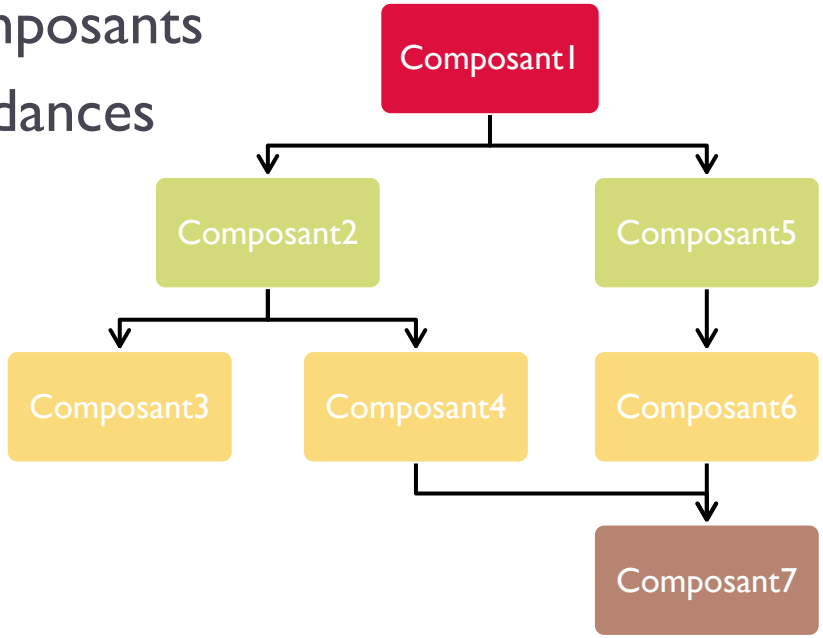
Approche Big Bang

- On considère l'assemblage en tant qu'unité
  - Seules les dépendances non contrôlables sont doublées

- Réaliste uniquement avec de petits systèmes pour lesquels il s'agit presque de passer directement à l'étape de test de validation du système.
- Inconvénients
  - Difficulté pour le diagnostic
  - Nécessitent que toutes les unités de l'assemblage soient prêtes
  - On se positionne à l'interface de l'assemblage et pas d'une unité avec ses dépendances : perte d'intensité

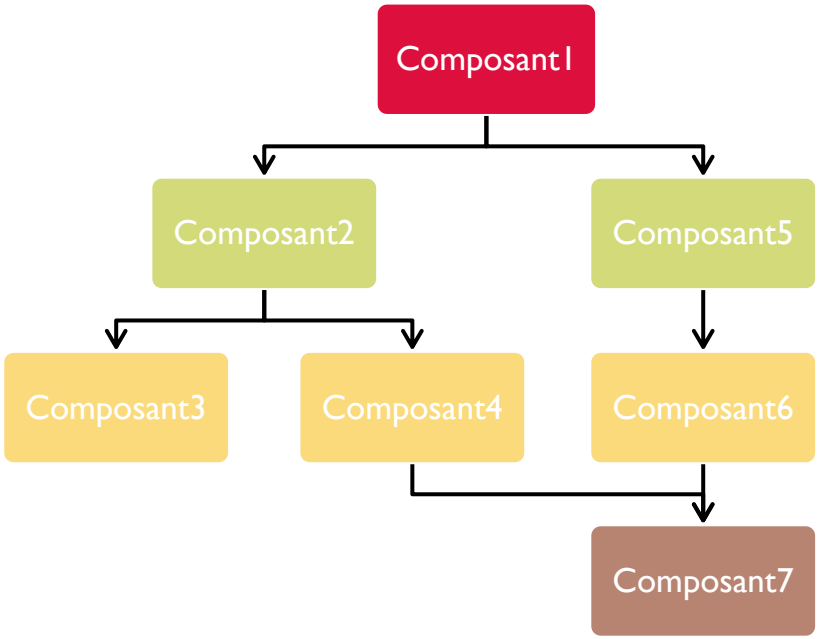
Approches progressives incrémentales

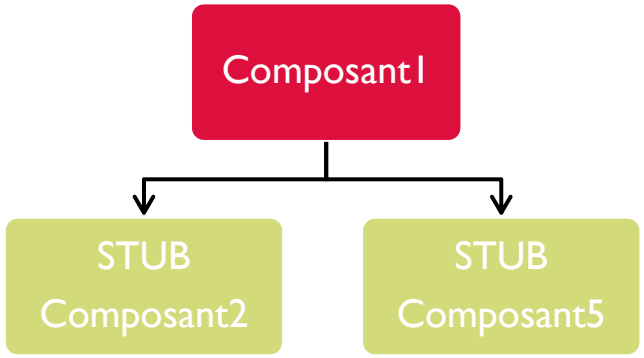
- On teste l'assemblage formé par l'intégration progressive des composants
- Basées sur un arbre de dépendances
  - Modélise les dépendances entre les composants en se basant sur l'architecture et sur le flot d'exécution
  - Nœuds : composants
  - Arcs : dépendances



Approche Top-down

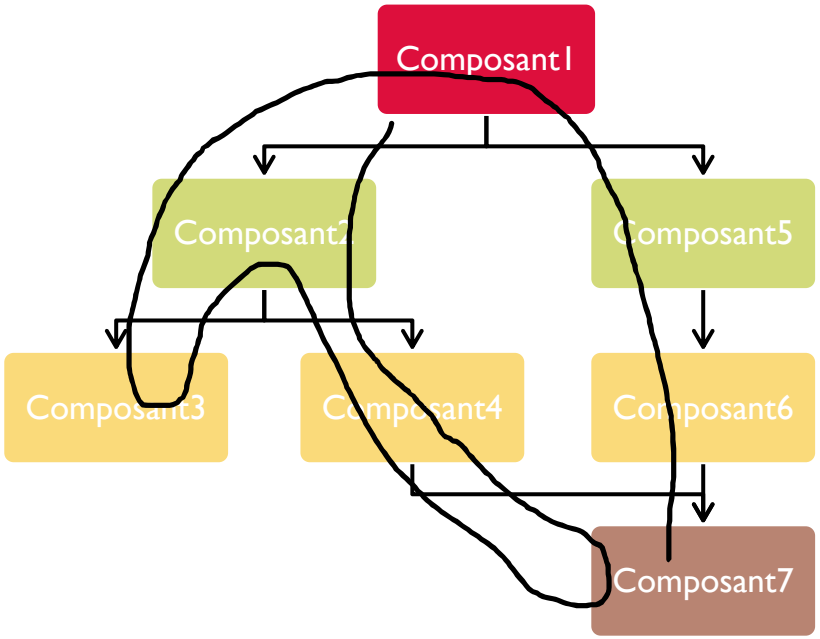
- En commençant par le début…
  - Depuis le sommet/racine de l'arbre jusqu'aux branches
  - Nécessitent des stubs

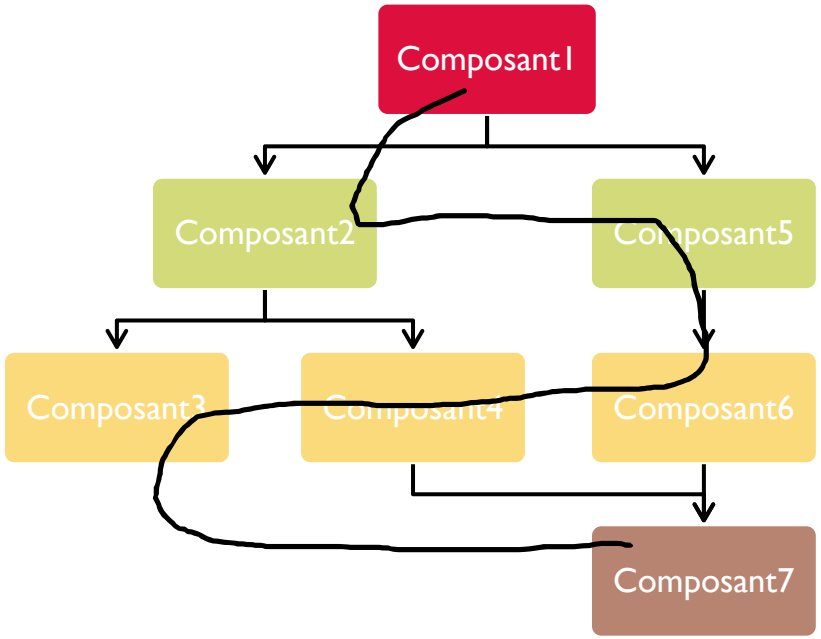




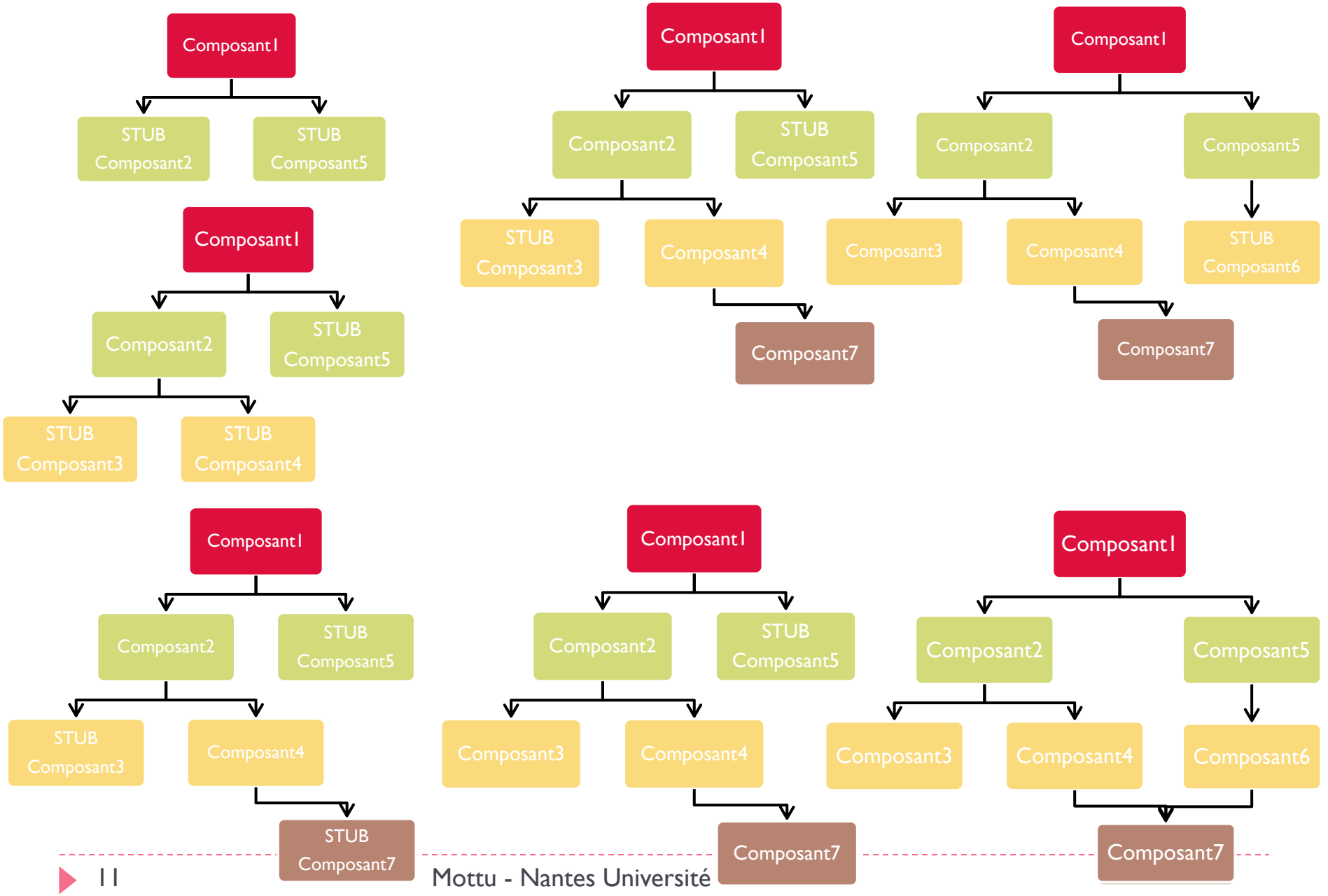
Approche Top-down

- Différents parcours incrémentaux
  - En profondeur ou en largeur

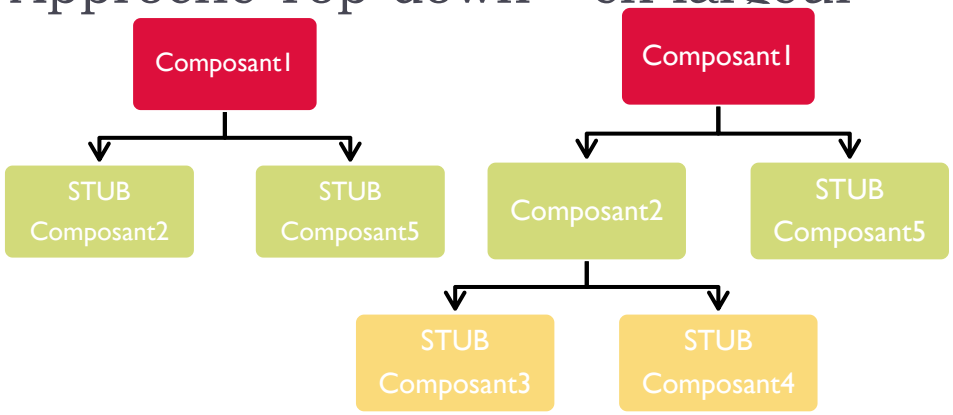


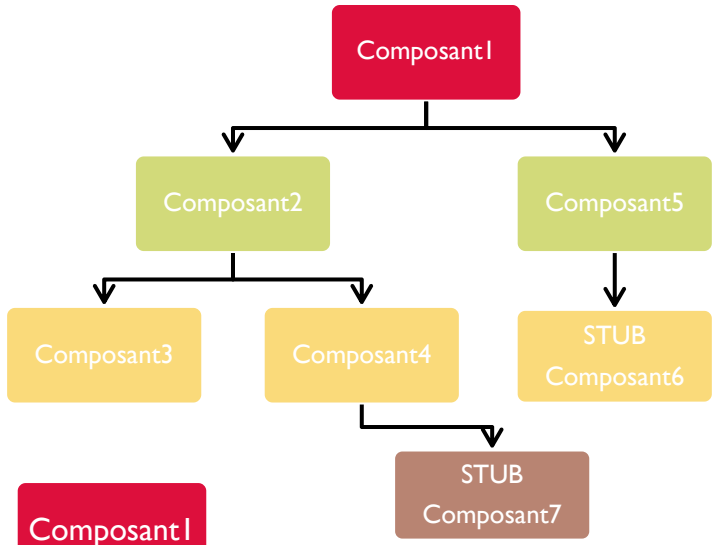


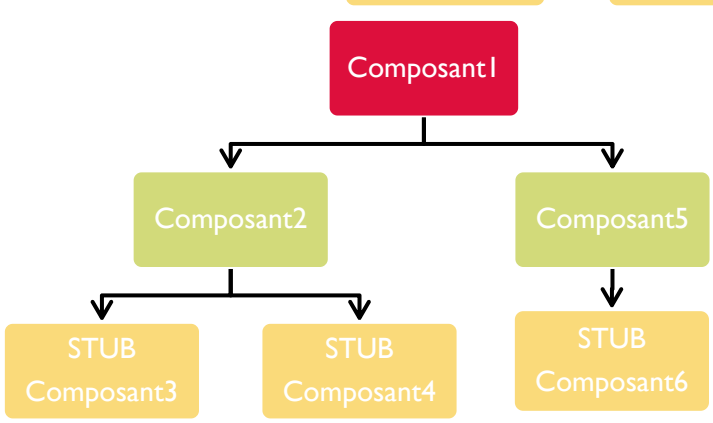
Approche Top-down – en profondeur

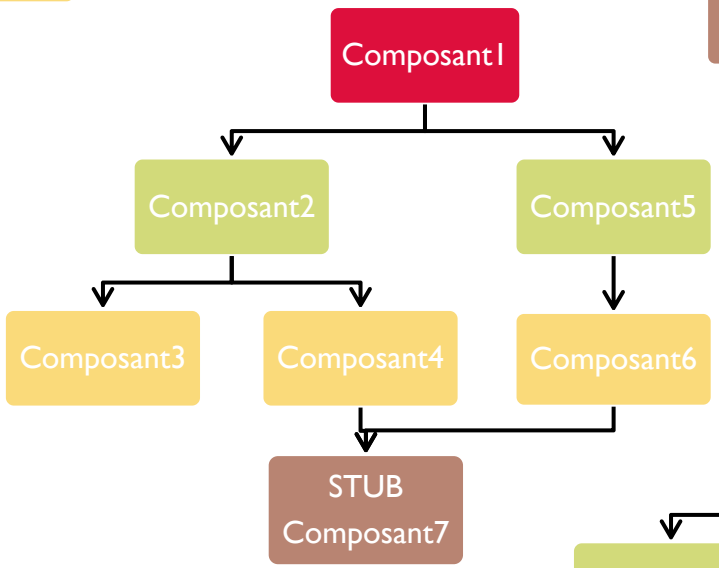


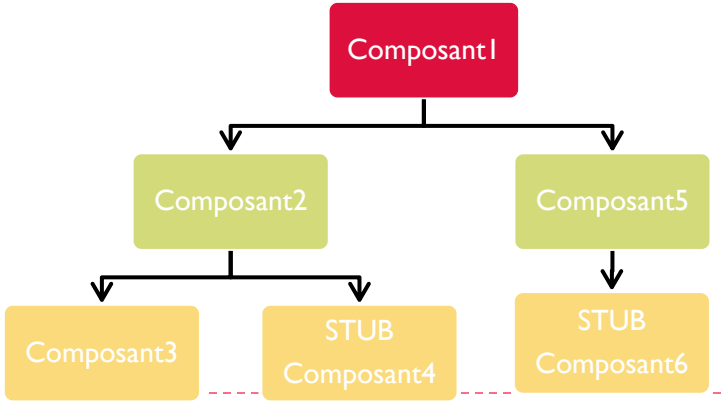
Approche Top-down – en largeur

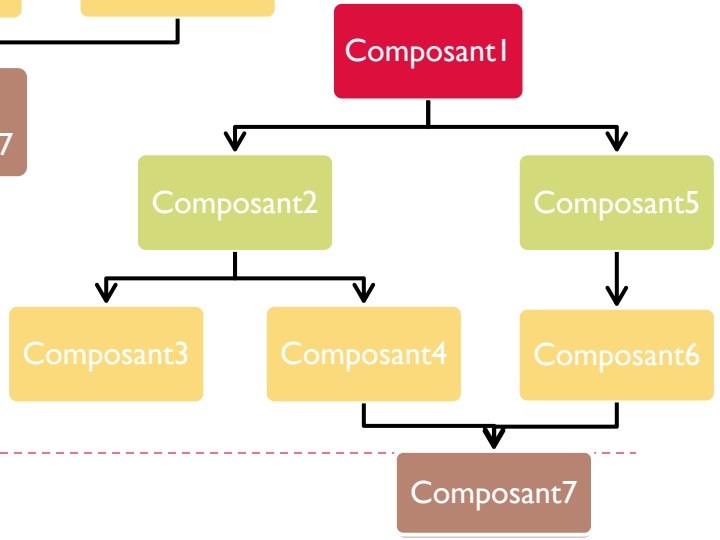






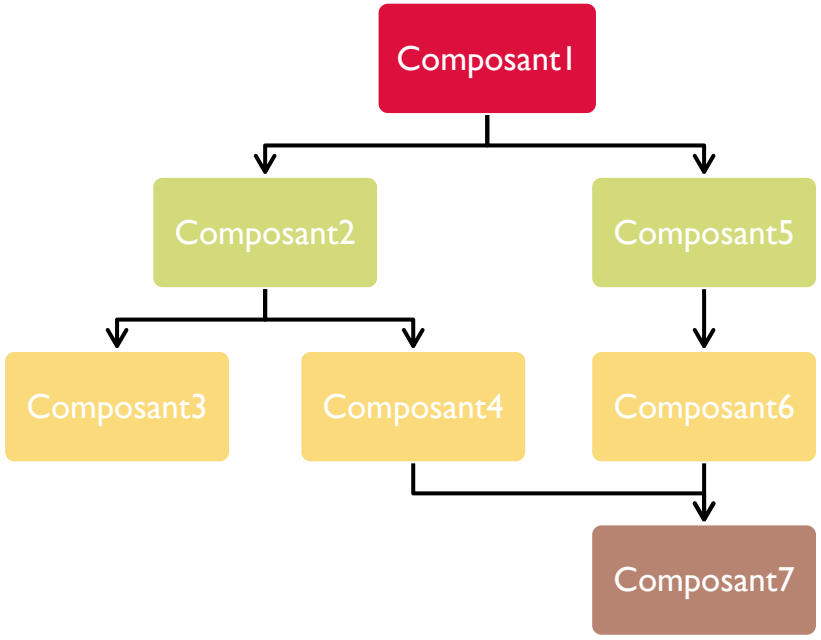


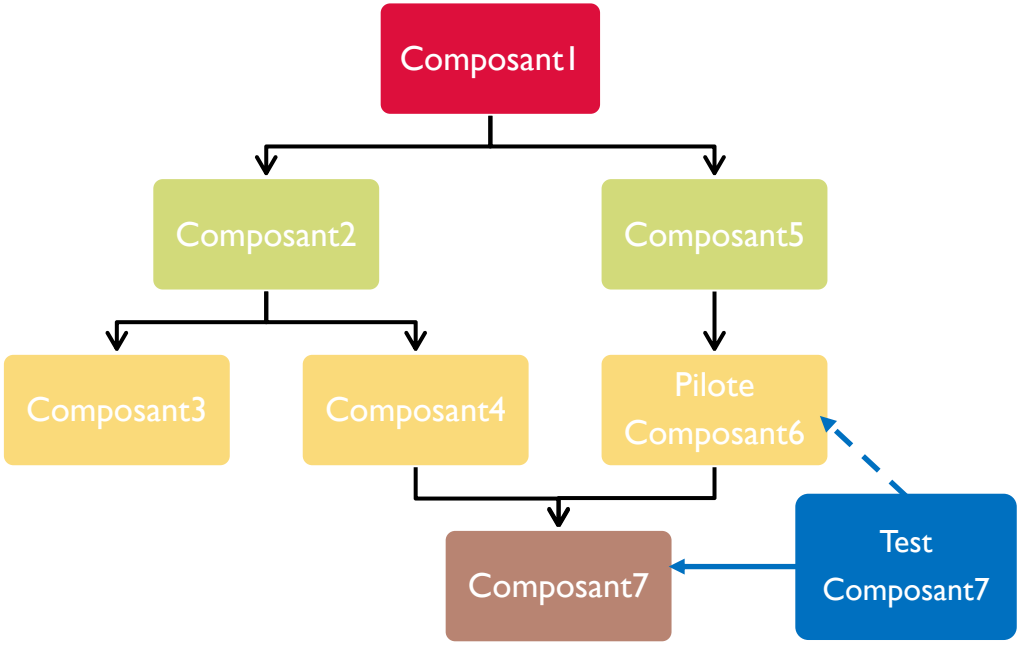




Approche Bottom-up

- En commençant par les moins dépendants
  - Depuis les feuilles jusqu'à la racine
  - Peut nécessiter des doublures Drivers
    - Un composant même sans dépendance peut ne pas pouvoir travailler seul, il a besoin d'être piloté
    - En particulier, les (oracles des) tests peuvent en avoir besoin





Bottom-up et Top-down avantages

- Débuter le test sans avoir tout le système
  - Grâce aux doublures (stubs et drivers)
- Progressivité des tests et de l'intégration facilite le diagnostic
- Hybridable : dans les 2 directions
  - approche Mixte, plus au fil du développement

Bottom-up VS Top-down

- Selon le cycle de développement:
  - Top-Down si on commence par les composants principaux
    - Permet de montrer un système (partiel) exécutable (précepte agile)
  - Bottom-up si on commence par les composants mineurs
- Selon le paradigme de programmation
  - Top-down si le développement est fait dans l'ordre des scénarios
  - Bottom-up si le développement est fait des classes les moins dépendantes
- Selon la difficulté
  - Top-down : relative à la création des stubs
  - Bottom-up : relative à la nécessité de drivers
- Selon l'efficacité du test et du diagnostic
  - Bottom-up, moins dépendant des stubs, peut être plus facilement diagnosticable
  - Top-down, commençant par les composants principaux permet de détecter des fautes (potentiellement majeures) tôt.

Bottom-up et Top down VS interdépendances

- Les deux approches
  - se focalisent sur l'intégration cumulative de composants
  - négligent les échanges entre composants s'intégrant mutuellement.
- Passage trop rapide du test unitaire au test d'intégration
  - Le test unitaire limité par les (inter)dépendance
  - Le test d'intégration trop cumulatif

- Les deux approches considèrent surtout des composants avec peu d'interdépendances
- En considérant l'intégration de systèmes à fortes interdépendances (en particulier l'intégration de classes en programmation orienté objet), ces approches sont confrontées à la difficulté de créer de multiples stubs :
  - ils deviennent nécessaire aussi en bottom-up,
  - mais finalement, l'arbre de dépendance n'est plus réaliste (même avec des arcs entre branches) :
    - graphe de dépendances

Graphe de dépendances de test (GDT)

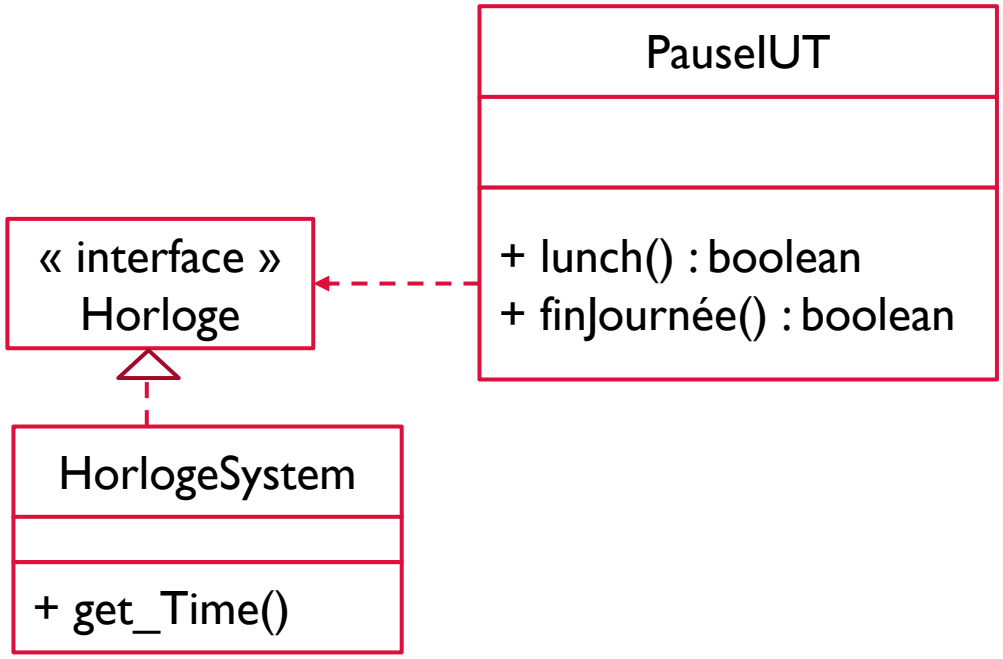
- Modélise les dépendances entre unités devant être intégrées
  - Typiquement les classes en POO
- Exploitable par le test d'intégration
  - pour minimiser le nombre de doublures nécessaire,
  - paralléliser le test d'intégration.

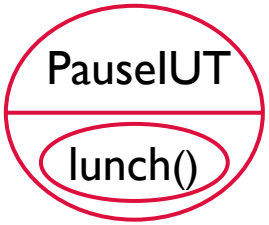
- Modélise
  - Deux types de dépendances
    - La spécialisation (héritage/polymorphisme)
    - L'association client/serveur
  - Deux niveaux de dépendances
    - Entre classes
    - Entre méthode et classe

Création du GDT

- Création des nœuds modélisant
  - Les classes
  - Les méthodes

Noeud classe

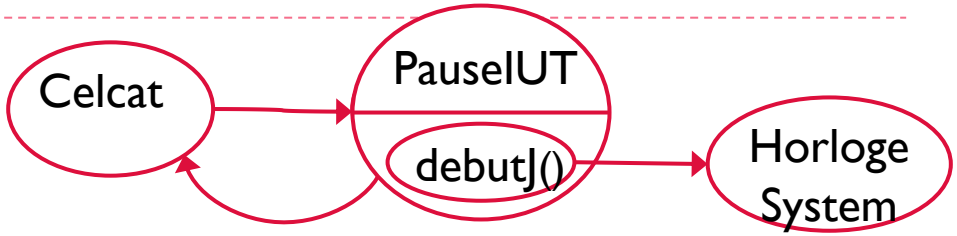


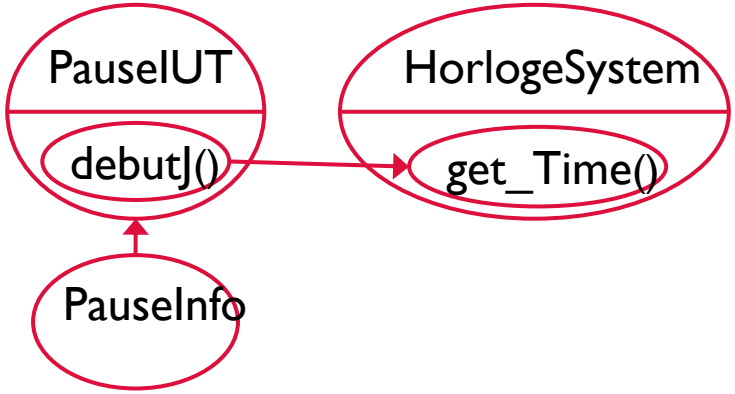


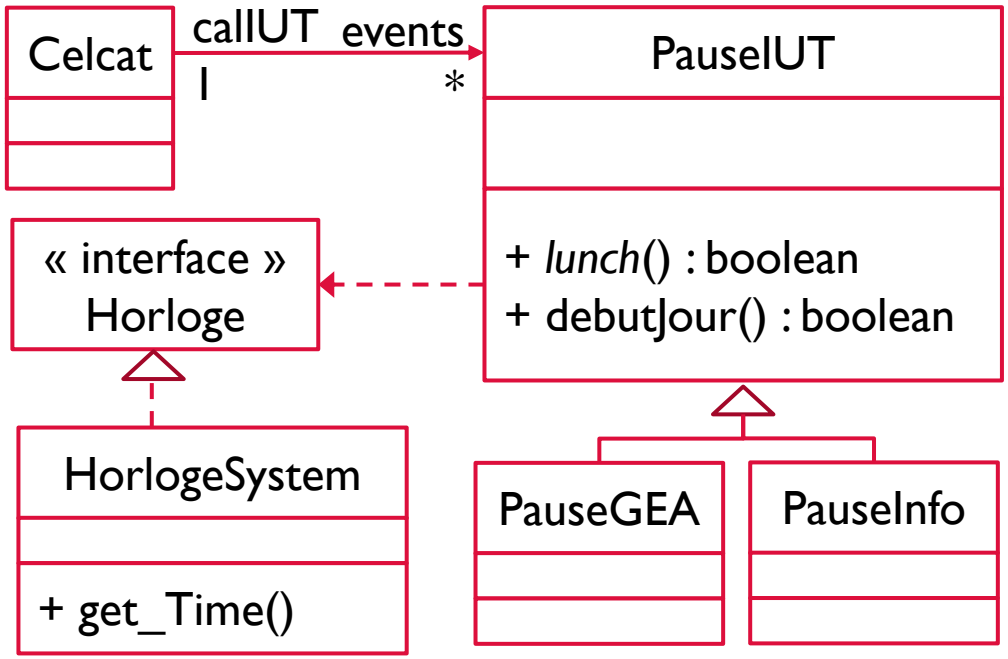
Noeud méthode dans classe

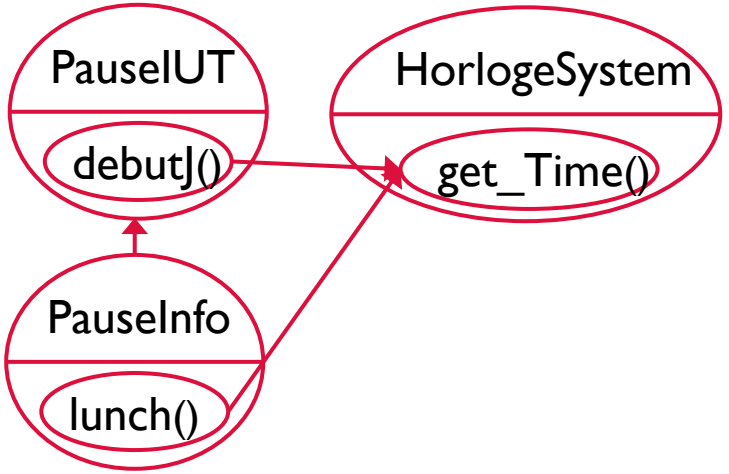
Création du GDT

- Création des arcs modélisant
  - Les relations entre classes
  - Les relations de méthode à classe
  - Les relations entre méthodes



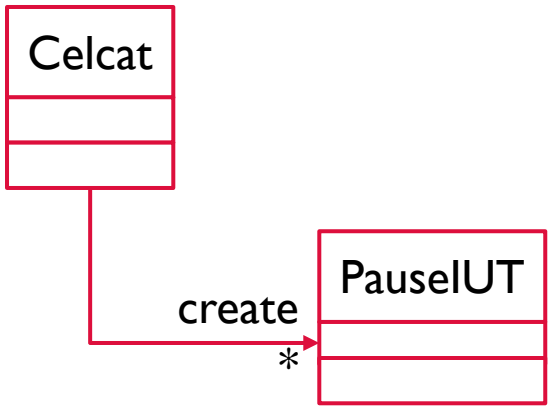


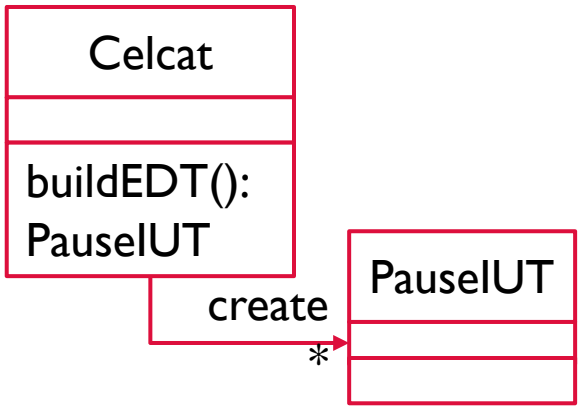


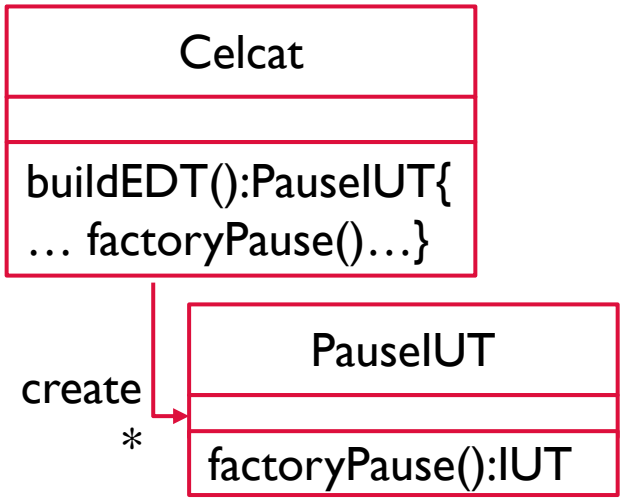


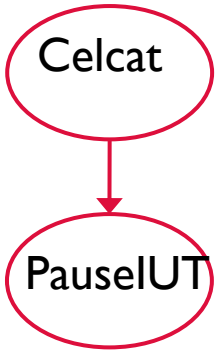
Création du GDT : niveau de détail

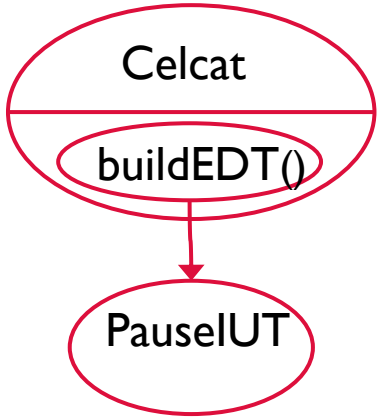
- Classe vers classe
- Méthode vers classe
- Méthode vers méthode

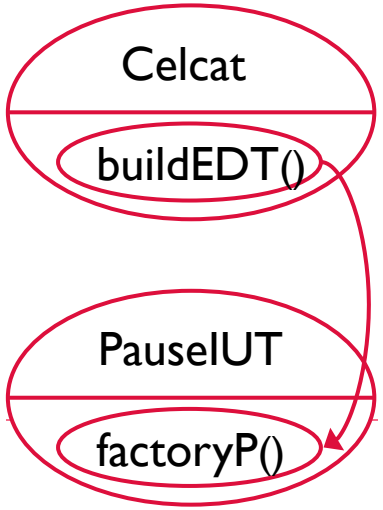






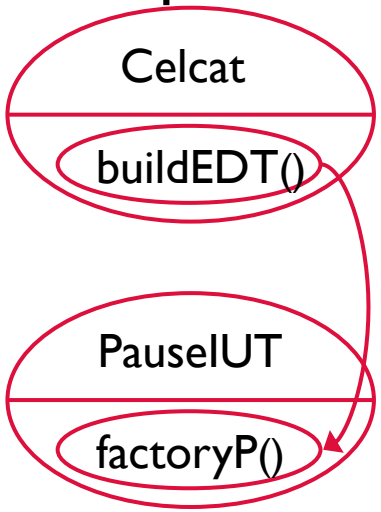




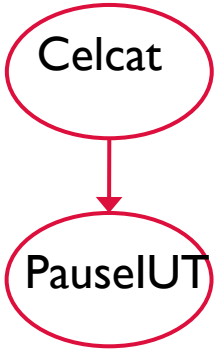


Création du GDT : mise à plat

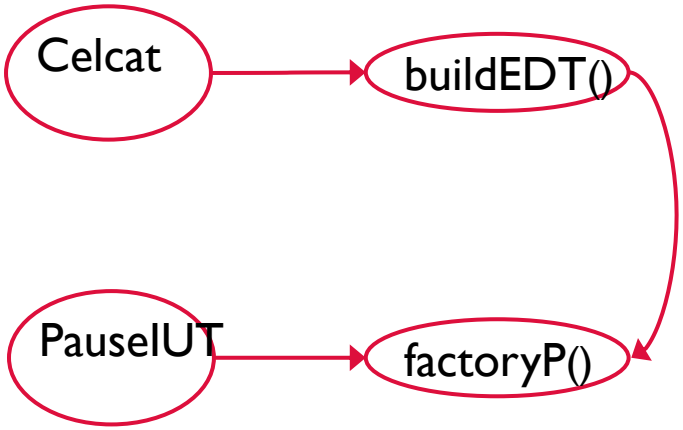
- On doit mettre à plat le GDT s'il a des nœuds imbriqués



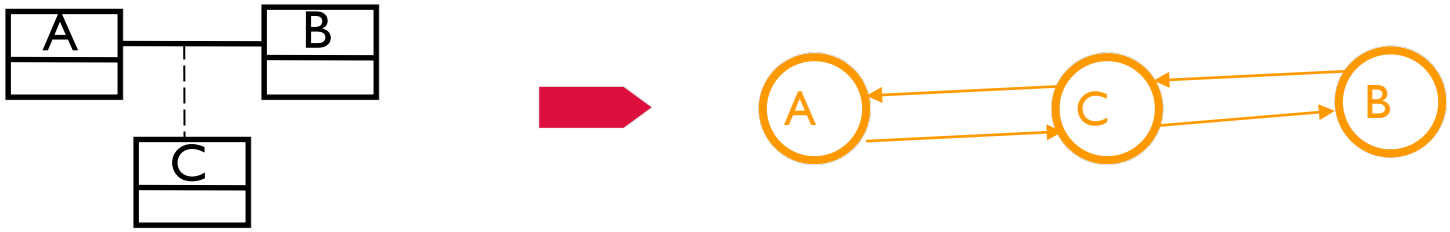
- Avec perte d'information, si on coupe :

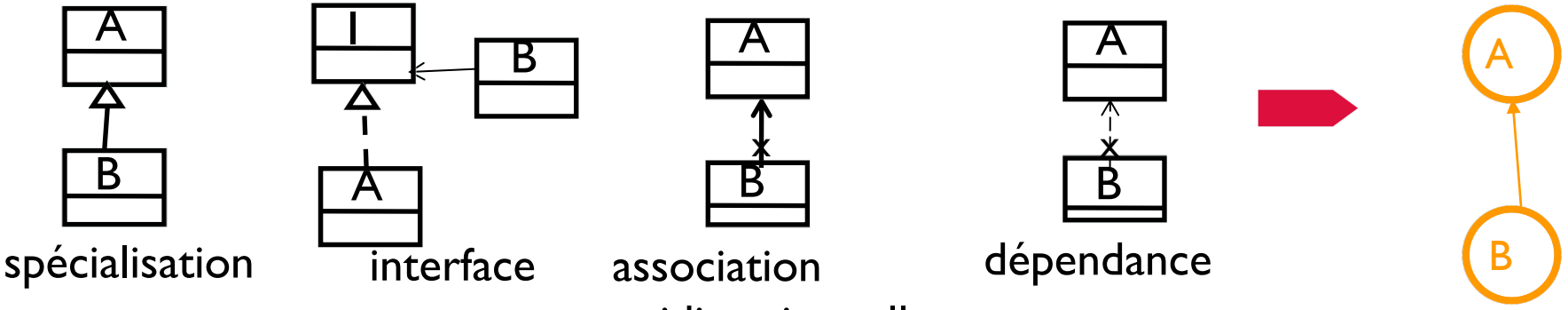


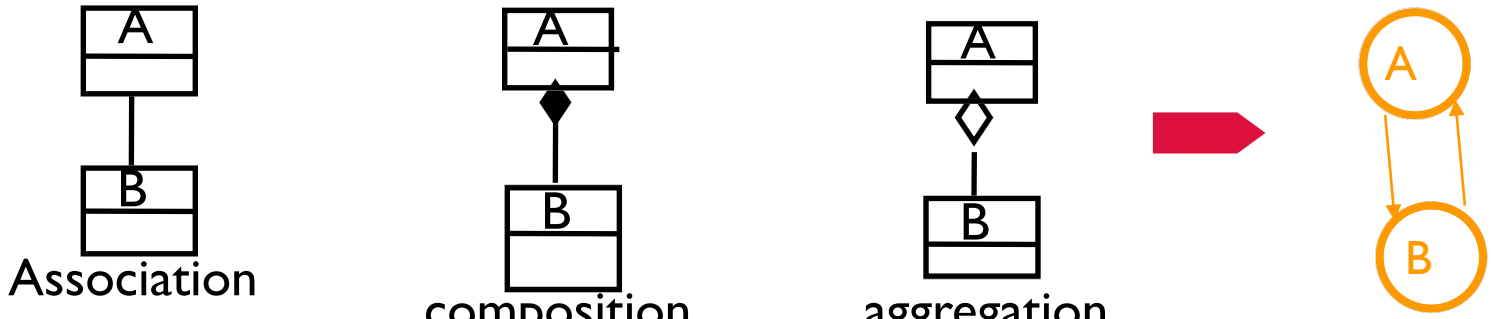
- Sans perte d'information, si on transforme en extrayant les nœuds :



Création du GDT : du diagramme de classes au GDT



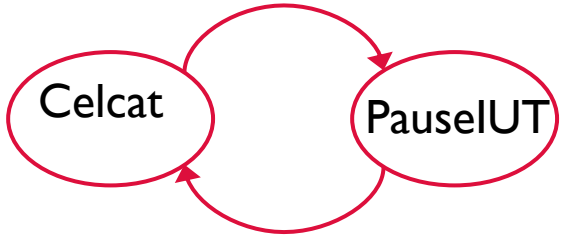




Ordonner le test d'intégration avec interdépendances

- Objectif : obtenir un plan (de test) d'intégration

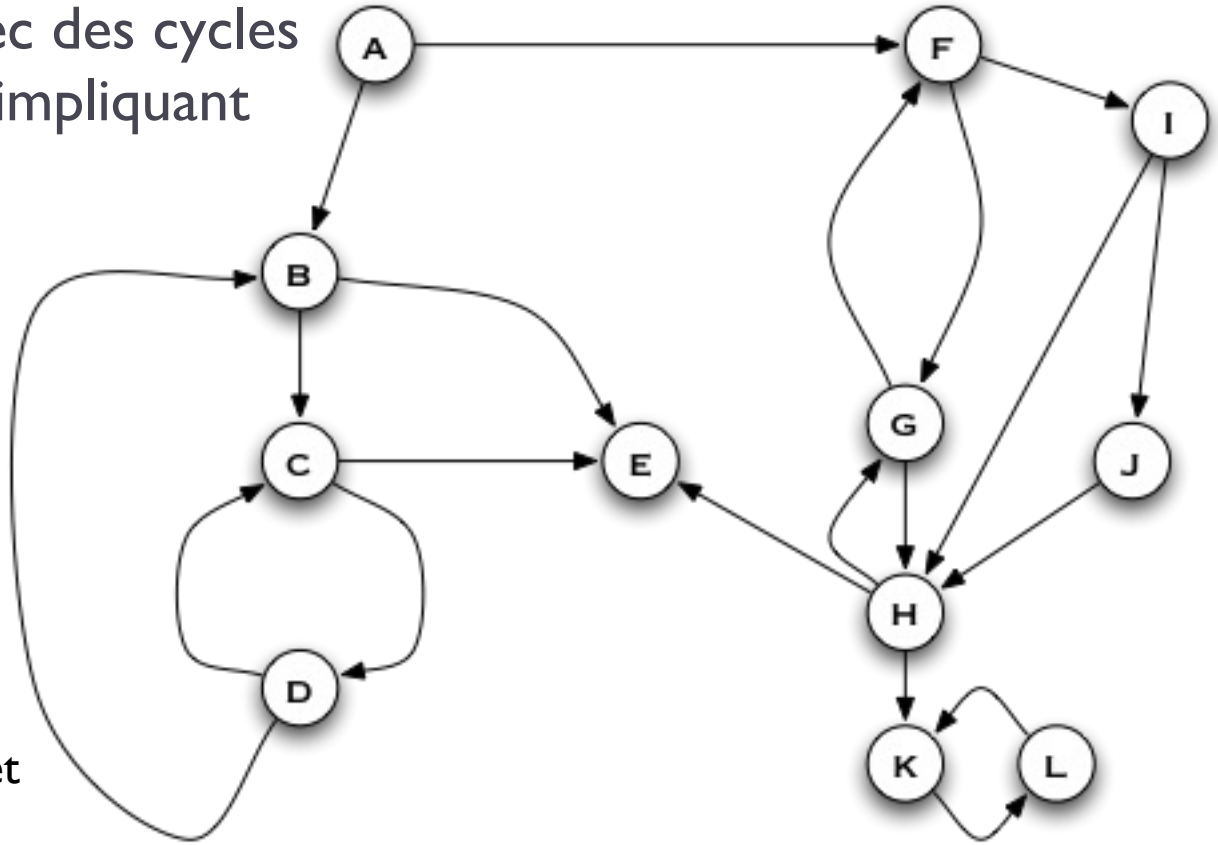
- Interdépendances entre 2 nœuds, plan d'intégration :
  - Etape 1
    - Tester la classe Celcat avec le stub de PauseIUT
  - Etape 2
    - Tester la classe PauseIUT
  - Etape 3
    - Retester la classe Celcat avec la classe PauseIUT



Plan d'intégration
Celcat *avec stub*(**PauseIUT**)
PauseIUT
restest **Celcat**

Ordonner le test d'intégration avec interdépendances : stratégie générale

- Comment minimiser le nombre de stub (et l'effort pour les créer et retester) ?
  - En particulier avec des cycles de dépendances impliquant plusieurs noeuds

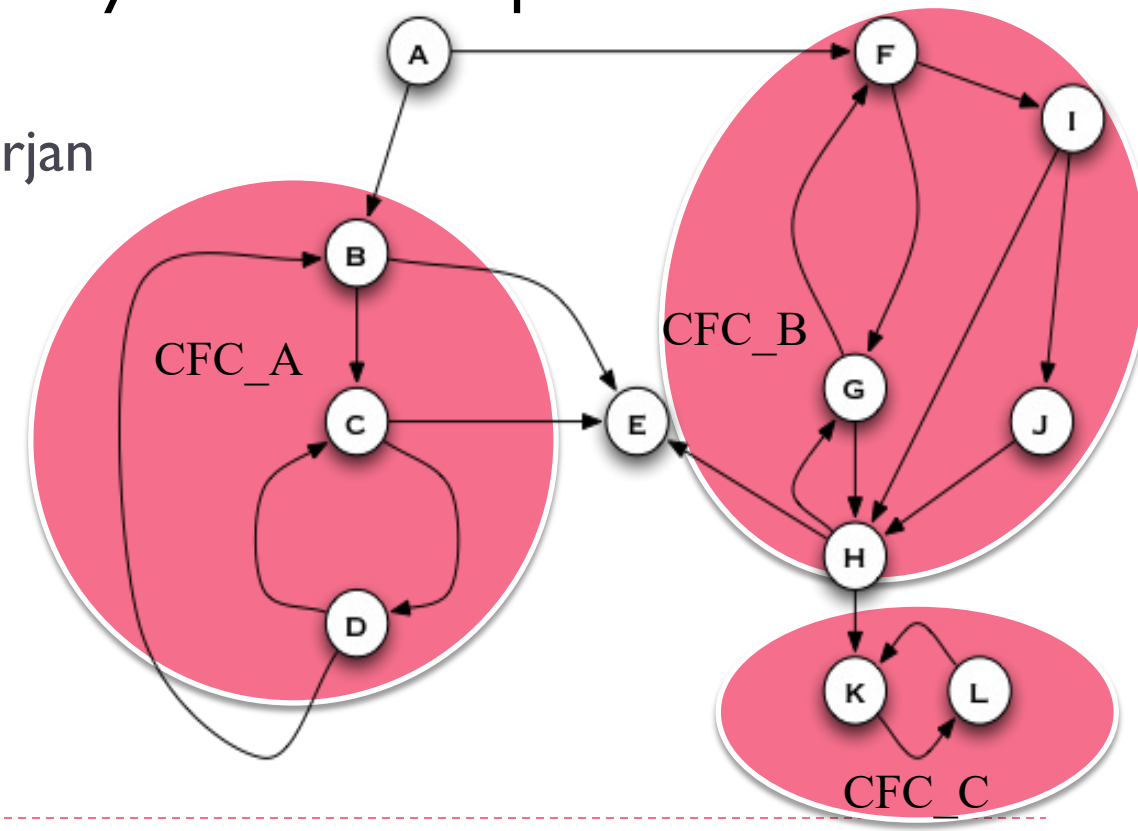


ordre optimal => NP-complet
complexité maximale = n!

Ordonner le test d'intégration avec interdépendances : stratégie générale

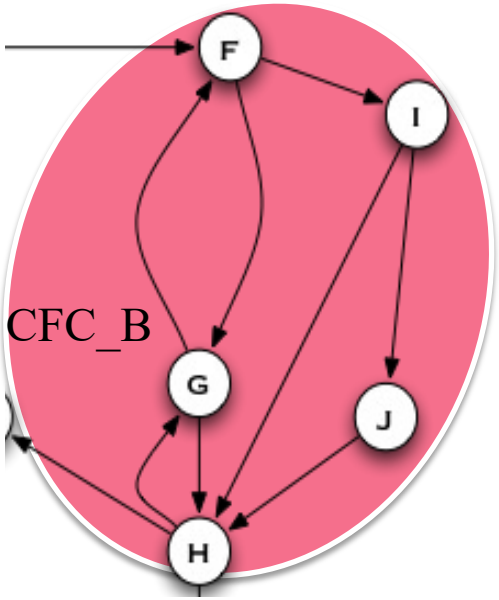
- Etape 1 : produire le GDT
- Etape 2 : identifier les composantes fortement connexes
  Ce sont les zones avec des cycles d'interdépendances et il va falloir casser ces cycles
  - Application de l'algo de Tarjan
  - Cela permet d'obtenir un plan d'intégration partiel

Plan d'intégration partiel
CFC_C
E
CFC_A
CFC_B
A



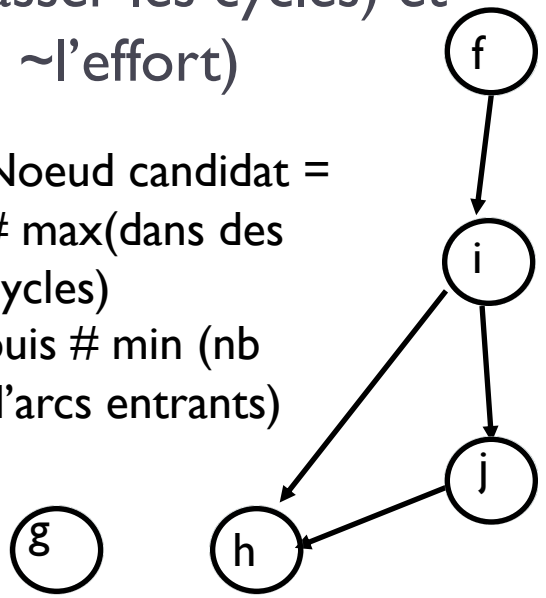
Ordonner le test d'intégration avec interdépendances : stratégie générale

- Etape 3 : casser les CFC
  - Identifier le nœud qu'on va isoler (pour casser les cycles) et stubber (en minimisant le nombre de stub ~l'effort)



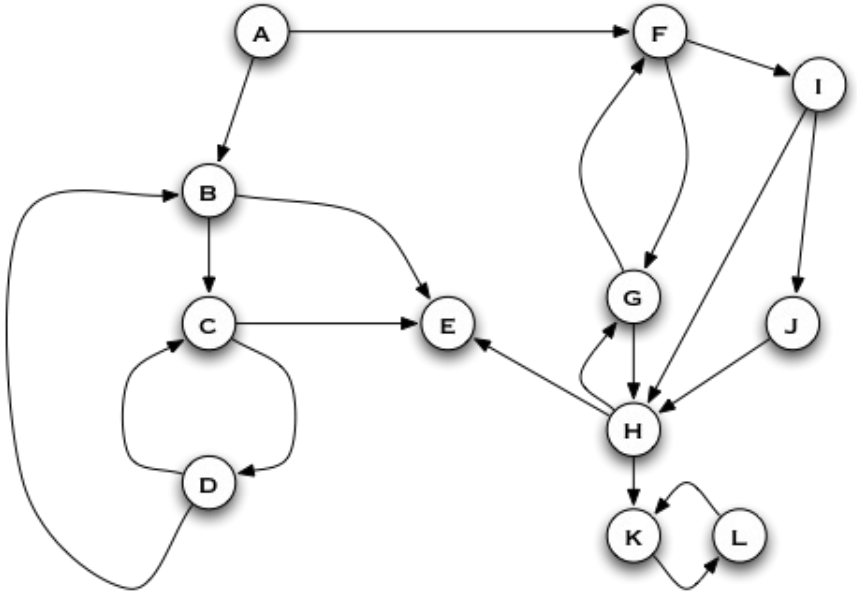
Noeud	#cycle	# min arcs entrants de la CFC
F	3	1
G	4	2
H	3	3
I	2	1
J	1	1

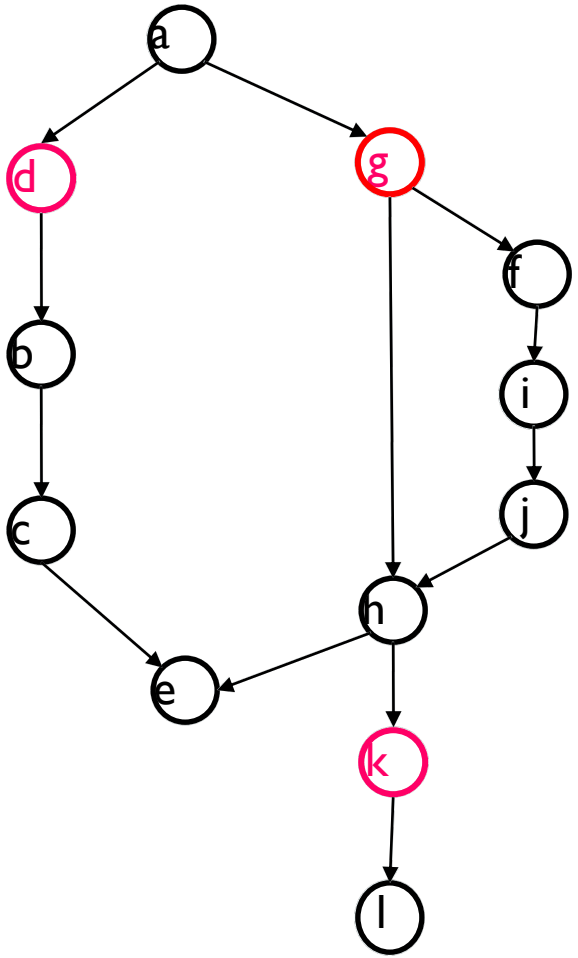
Noeud candidat = # max(dans des cycles) puis # min (nb d'arcs entrants)



Ordonner le test d'interdépendances : stratégie générale

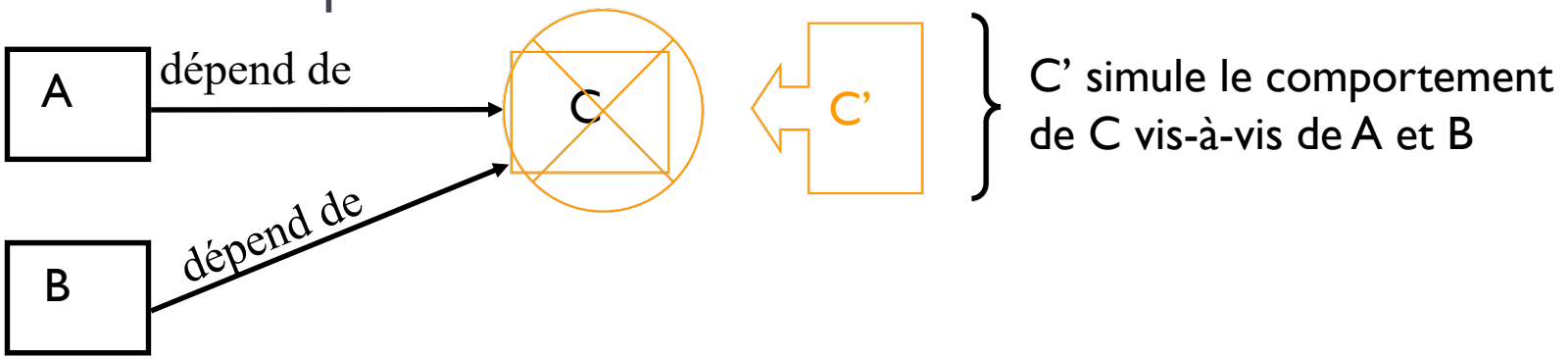
Résultat = un ordre partiel
de toutes les stratégies possibles



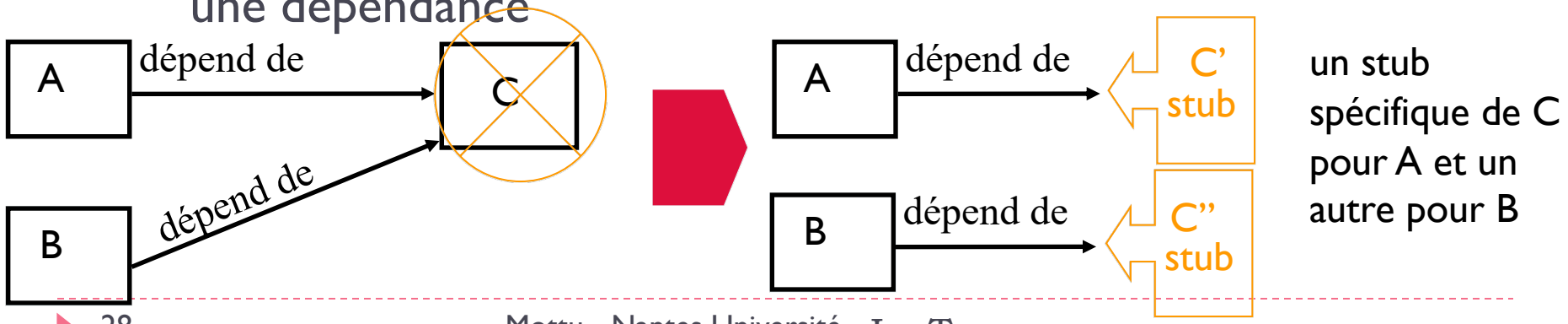


Création de stub ajusté au besoin

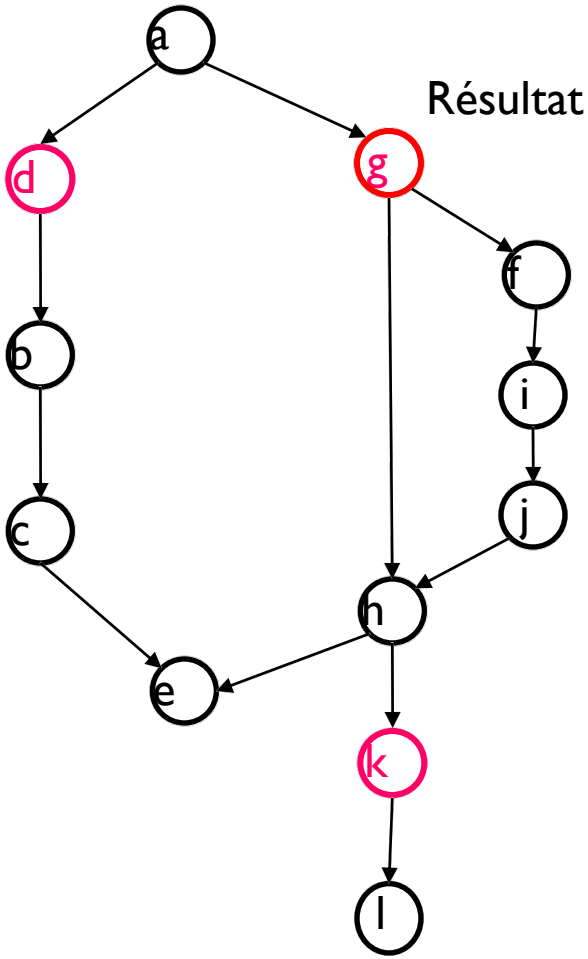
- Une classe peut être stubbé en function du besoin des classes dépendantes :
  - Un stub réaliste simule le comportement necessaire pour toutes ces dépendances



  - Un stub spécifique simule le comportement necessaire pour une dépendance



Ordonner le test d'interdépendances : stratégie générale



Résultat = un ordre partiel de toutes les stratégies possibles

Algorithme optimisé

#stubs spécifiques = 4
#stubs réalistes = 3

Génération aléatoire

#stubs spécifiques = 9.9
#stubs réalistes = 5

Conclusion des étapes de test d'intégration et unitaire

- Illusoire de penser qu'il s'agit de deux étapes successives, disjointes
  - Les (inter)dépendances sont multiples

- Retour sur la testabilité :
  un système sera plus testable s'il permet
  - de tester intensivement ses unités
  - d'intégrer progressivement ses unités

- On devrait porter un effort supplémentaire sur les interfaces entre composants.

To be continued