Développement dirigé par les tests

Test Driven Development (TDD)

Kent Beck « Une fonctionnalité sans test automatique n'existe tout simplement pas»

Merci à tous ceux qui ont rendu leurs cours et exposés disponibles sur le web & dans les livres, voir Biblio.

M. Blay-Fornarino
blay@unice.fr,
http://users.polytech.unice.fr/~blay/
IUT Département Informatique 2è année

Bibliographie

- Programmation par les tests, ESIREM, Céline ROUDET
- Comment écrire du code testable, Conférence Agile France 2010, Florence CHABANOIS
- Reflexion on Software Quality and Maintenance, Alexandre Bergel, Chili
- An Introduction to Test-Driven Development (TDD), Craig Murphy
- Tests et Validation du logiciel, http://home.nordnet.fr/~ericleleu
- Test à partir de modèles : pistes pour le test unitaire de composant, le test d'intégration et le test système, Yves Letraon
- Les tests en orienté objet, J. Paul Gibson http://www-inf.int-evry.fr/cours/CSC4002/Documents
- Mocks and Stubs, Martin Fowler
- Introduction au test du logiciel, Premiers pas avec JUnit, Mirabelle Nebut
- Écrire du code testable Par Aurélien Bompard

Boite de vitesse ?

Frein ?

Direction assistée?

Pneus défectueux ?

Qu'est-ce qu'un test ?

Définition :

1 expérience d'exécution pour mettre en évidence un défaut, une erreur

➡Vérifier que le résultat de l'expérience est conforme aux intentions

➡*Diagnostic* : quel est le problème, l'erreur

➡*Localisation* : où est la cause du problème ?

Les tests à faire et les résultats dépendent des spécifications

Qui teste ?

- L'utilisateur
- Les collègues en charges du test (s'il y en a)
- Le développeur : il a le devoir de fournir un code le plus clair et le mieux testé possible.

Qu'est-ce qu'on teste et comment ?

Quoi ?

- Fonctionnalité
- Sécurité / intégrité
- Utilisabilité
- Cohérence
- Maintenabilité
- Efficacité
- Robustesse
- Sûreté de fonctionnement

Comment ?

- **Test statique :**
  - relecture / revue de code
  - analyse automatique (vérification de propriétés, règles de codage ...)
- **Test dynamique :**
  - exécution du programme, et observation du comportement en fonction des valeurs en entrée.

Une spécification exprime ce qu'on attend du système, elle prend différentes formes en fonction de ce qui est ciblé : cahier des charges, use cases, données numériques, ...

But d'un test et TDD

- Détecter les **défauts** le plus tôt possible dans le cycle
  - Tester une nouvelle méthode dès qu'on l'écrit
  - Répéter l'ensemble des tests à chaque modification du code

Test Driven Development (TDD)

- Ecrire les cas de test **avant le programme**
- Développer la partie du programme qui **fait passer les cas de test**

« The point of TDD is to drive out the functionnality the software actually needs, rather than what the programmer thinks it probably ought to have »,
Dan North

Rédiger les tests avant le code

Spécifier chaque fonction :

➡NE PAS regarder comment elle va faire

➡MAIS PRECISER :
- ce qu'elle doit faire (description)
- dans quel contexte (pré-conditions)
- avec quel résultat (post-conditions)

Rédiger les tests avant le code (2)

On peut donc, avant de réfléchir à « comment faire » :

➡ Donner l'ensemble des cas d'utilisation

➡ En déduire la liste des tests à faire et les coder

Tant qu'il reste un test i non validé :

✓ **Rédiger** test i

✓ **Ajouter code** nécessaire pour que la fonction passe avec le test i

✓ **Revoir le code** rédigé pour factoriser des lignes, réorganiser le code interne

✓ **Revalider tests 1 à i** (tests de non-régression)

TDD

Prenons un exemple !

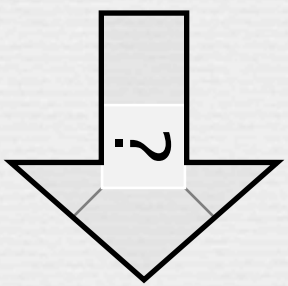
Tests

Quels types de tests ?

Qu'est-ce qu'on teste et comment ?

Quoi ?

- Fonctionnalité
- Sécurité / intégrité
- Utilisabilité
- Cohérence
- Maintenabilité
- Efficacité
- Robustesse
- Sûreté de fonctionnement



Comment ?

- **Test statique :**
  - relecture / revue de code
  - analyse automatique (vérification de propriétés, règles de codage ...)
- **Test dynamique :**
  - exécution du programme, et observation du comportement en fonction des valeurs en entrée.

Une spécification exprime ce qu'on attend du système, elle prend différentes formes en fonction de ce qui est ciblé : cahier des charges, use cases, données numériques, ...

Tests Dynamiques

Si déjà vu, aller ici

Exemple à tester

Type enumere Level = {Low, Mid, High}

Classe LevelManagement :

- int getLowLevel()
- int getMidLevel()
- int getHighLevel()
- void setLevel(int low, int mid, int high)
- Level getLevel(int x)

Exemple à tester

Level getLevel(int x)

- ✓lève une OutOfLevelException si x $\notin$ [0; getHighLevel()]
- ✓retourne Level.Low si x $\in$ [0; getLowLevel()]
- ✓retourne Level.Mid si x $\in$]getLowLevel(); getMidLevel()]
- ✓retourne Level.High si x $\in$]getMidLevel(); getHighLevel()]

Critère de test

D'abord on choisit un critère de test :

- tester une fonctionnalité donnée ;
- couvrir toutes les instructions d'un programme ;
- etc.

Par exemple, je décide de tester les méthodes de la classe *LevelManagement* dans une approche :

- "boîte noire" (on ne regarde pas le source) ;
- fonctionnelle : on vérifie que les sorties sont correctes pour des entrées données

Objectif de test

Ensuite on choisit une propriété ou caractéristique à tester en accord avec le critère.

C'est l'objectif de test.

➡Ex : tester le comportement de la méthode getLevel() quand x ∈]getLowLevel(); getMidLevel()].

Donnée de test

Ensuite on choisit une donnée de test.

Pour notre exemple, il faut choisir :

- une valeur pour chacun des trois seuils ;
- une valeur pour le paramètre.

La donnée de test sera par exemple :

(lowLevel = 2,
midLevel = 10,
highLevel = 20,
x = 7)

Donnée de test

Ensuite on infère de la spécification, le résultat attendu.

Souvent on se rend compte que la spécification est des plus informelles.

On en conclut que getLevel(7) doit retourner Level.Mid.

Test

Dans notre exemple,

si *lm* est un objet de type *LevelManagement* tel que :

lowLevel = 2, midLevel = 10, highLevel = 20

alors on effectue lm.getLevel(7);

Oracle

On compare le résultat obtenu au résultat attendu :

c'est **l'oracle**.

Dans notre exemple, l'oracle est :

lm.getLevel(7) = Level.Mid

L'oracle est toujours une assertion.

En objet le langage des assertions peut être très riche :

- ✓ expression booleenne
- ✓ levée d'une exception
- ✓ appel d'une méthode sur tel objet avec tel paramètre (interactions)
- ✓ etc.

Verdict

On déduit de l'exécution de l'oracle si le test a réussi ou échoué.

C'est le **verdict**.

Verdict classique pour une assertion "booleenne" :

- ✓ elle vaut true : le test passe ;
- ✓ elle vaut false : le test échoue ;
- ✓ il y a une levée d'exception pas prévue : erreur, le test est inconclusif.

Associer verdict et oracle

On aimerait déduire automatiquement le verdict de l'exécution du test (via l'oracle).

En première approche, avec les moyens du bord :

```
...
try {
if (lm.getLevel(7) == Level.Mid)
        S.O.P("test passe");
else
        S.O.P("test echoue");
catch (Exception e) {
        S.O.P("erreur, test inconclusif");
}
```

Fixture

Avant d'exécuter l'oracle, il faut amener l'objet dans un contexte/état favorable.

Cette préparation du contexte = fixture, installation, setUp ;

Dans notre exemple : création de l'objet et positionnement de son état

```
LevelManagement lm = new LevelManagement();
lm.setLevel(2, 10, 20);
try { ... // oracle ...}
```

Cas de test

L'ensemble du test executable s'appelle un **cas de test**.

```
LevelManagement lm = new LevelManagement();
lm.setLevel(2, 10, 20);
try {
if (lm.getLevel(7) == Level.Mid)
S.O.P("test passe");
else
S.O.P("test echoue");
catch (Exception e) {
S.O.P("erreur, test inconclusif");
}
```

Suite de tests

On regroupe les cas de test par suite de tests :

✓ collection de cas de tests (abstraits ou exécutables) ;

✓ ou collection de suite de tests (abstraits ou exécutables).

Permet de :

- grouper des tests par concepts
- lancer des tests par paquets

Tests Statiques

Tests statiques

Lire, faire lire ou traiter par un outil d'analyse

- Traitement du code source sans exécution
  - Lectures croisées (en groupe), inspection, règles et protocoles précis

- Analyse d'anomalies
  - Incohérence des interfaces, de modules, pointeurs, défauts de portabilité …
  - Utilisation de métriques

  => la présence d'anomalies révèle souvent des erreurs de logique

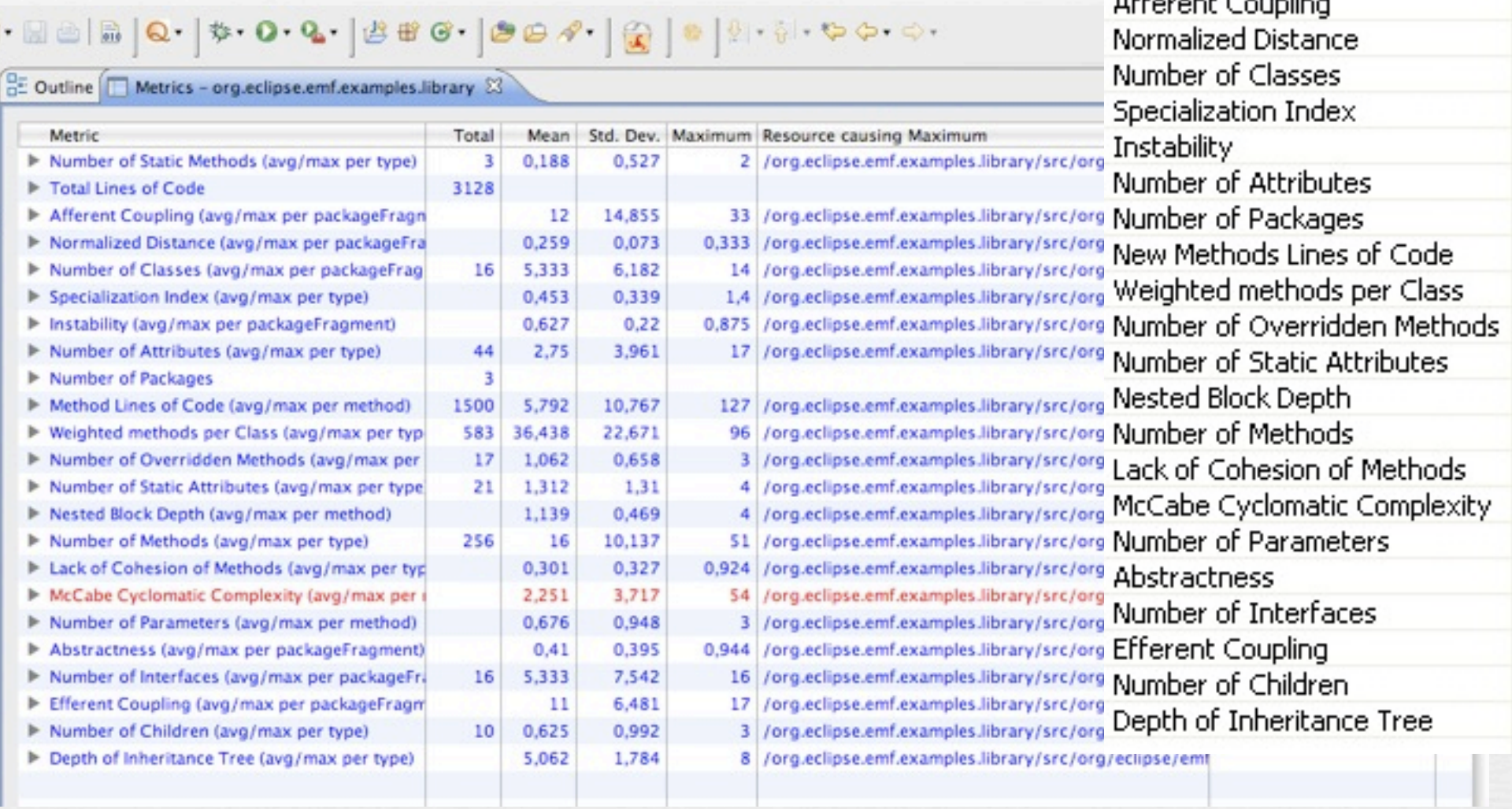
Mesures pour une bonne lisibilité du code

- Nombre d'instructions des opérations/méthodes
- Nombre d'opérations/méthodes par module/classes
- Nombre de variables/attributs par module/classe
- Couplage entre classes/modules
  - fan-out : nombre de modules appelés
  - fan-in : nombre de modules appelant le module à considérer.
  - Couplage fort entre modules/classes
- Taux de commentaires

Pas de panique, il existe des outils pour ce travail

Tests statiques

(plugin Metrics : http://metrics.sourcefor



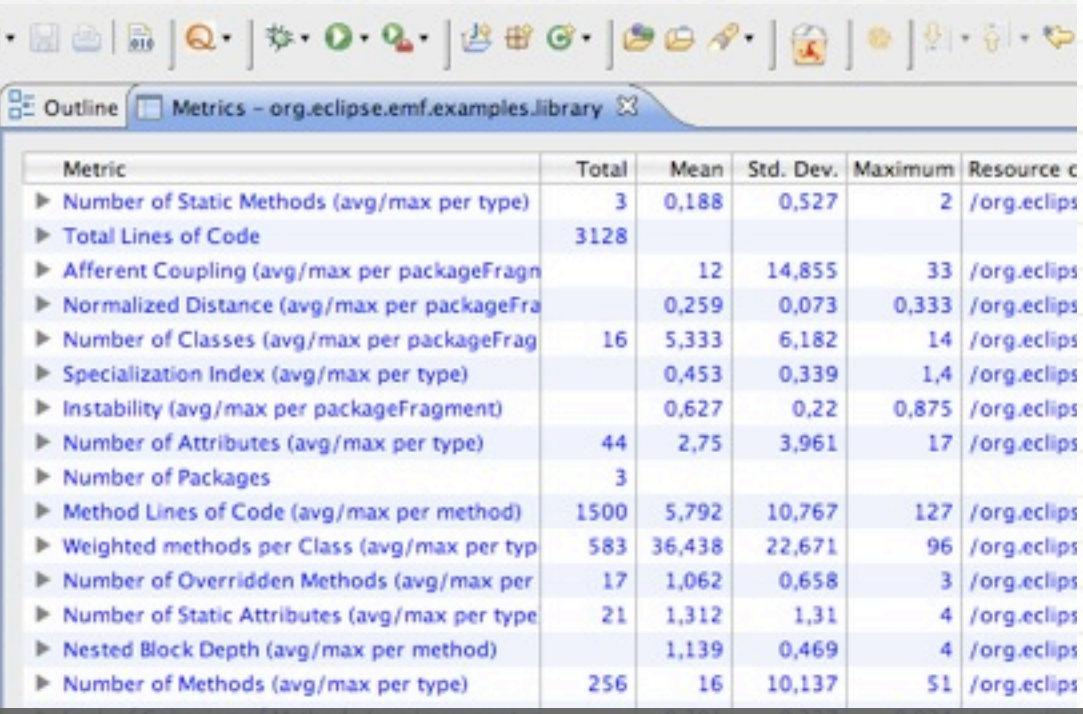
Outline | Metrics - org.eclipse.emf.examples.library

Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maximum
Number of Static Methods (avg/max per type)	3	0,188	0,527	2	/org.eclipse.emf.examples.library/src/org
Total Lines of Code	3128				
Afferent Coupling (avg/max per packageFragn		12	14,855	33	/org.eclipse.emf.examples.library/src/org
Normalized Distance (avg/max per packageFra		0,259	0,073	0,333	/org.eclipse.emf.examples.library/src/org
Number of Classes (avg/max per packageFrag	16	5,333	6,182	14	/org.eclipse.emf.examples.library/src/org
Specialization Index (avg/max per type)		0,453	0,339	1,4	/org.eclipse.emf.examples.library/src/org
Instability (avg/max per packageFragment)		0,627	0,22	0,875	/org.eclipse.emf.examples.library/src/org
Number of Attributes (avg/max per type)	44	2,75	3,961	17	/org.eclipse.emf.examples.library/src/org
Number of Packages	3				
Method Lines of Code (avg/max per method)	1500	5,792	10,767	127	/org.eclipse.emf.examples.library/src/org
Weighted methods per Class (avg/max per typ	583	36,438	22,671	96	/org.eclipse.emf.examples.library/src/org
Number of Overridden Methods (avg/max per	17	1,062	0,658	3	/org.eclipse.emf.examples.library/src/org
Number of Static Attributes (avg/max per type	21	1,312	1,31	4	/org.eclipse.emf.examples.library/src/org
Nested Block Depth (avg/max per method)		1,139	0,469	4	/org.eclipse.emf.examples.library/src/org
Number of Methods (avg/max per type)	256	16	10,137	51	/org.eclipse.emf.examples.library/src/org
Lack of Cohesion of Methods (avg/max per typ		0,301	0,327	0,924	/org.eclipse.emf.examples.library/src/org
McCabe Cyclomatic Complexity (avg/max per		2,251	3,717	54	/org.eclipse.emf.examples.library/src/org
Number of Parameters (avg/max per method)		0,676	0,948	3	/org.eclipse.emf.examples.library/src/org
Abstractness (avg/max per packageFragment)		0,41	0,395	0,944	/org.eclipse.emf.examples.library/src/org
Number of Interfaces (avg/max per packageFr	16	5,333	7,542	16	/org.eclipse.emf.examples.library/src/org
Efferent Coupling (avg/max per packageFragm		11	6,481	17	/org.eclipse.emf.examples.library/src/org
Number of Children (avg/max per type)	10	0,625	0,992	3	/org.eclipse.emf.examples.library/src/org
Depth of Inheritance Tree (avg/max per type)		5,062	1,784	8	/org.eclipse.emf.examples.library/src/org/eclipse/emf

Number of Static Methods
Total Lines of Code
Afferent Coupling
Normalized Distance
Number of Classes
Specialization Index
Instability
Number of Attributes
Number of Packages
New Methods Lines of Code
Weighted methods per Class
Number of Overridden Methods
Number of Static Attributes
Nested Block Depth
Number of Methods
Lack of Cohesion of Methods
McCabe Cyclomatic Complexity
Number of Parameters
Abstractness
Number of Interfaces
Efferent Coupling
Number of Children
Depth of Inheritance Tree

Tests stati

(plugin Metrics : http://metr



Metric	Total	Mean	Std. Dev.	Maximum	Resource c
Number of Static Methods (avg/max per type)	3	0,188	0,527	2	/org.eclips
Total Lines of Code	3128				
Afferent Coupling (avg/max per packageFragn		12	14,855	33	/org.eclips
Normalized Distance (avg/max per packageFra		0,259	0,073	0,333	/org.eclips
Number of Classes (avg/max per packageFrag	16	5,333	6,182	14	/org.eclips
Specialization Index (avg/max per type)		0,453	0,339	1,4	/org.eclips
Instability (avg/max per packageFragment)		0,627	0,22	0,875	/org.eclips
Number of Attributes (avg/max per type)	44	2,75	3,961	17	/org.eclips
Number of Packages	3				
Method Lines of Code (avg/max per method)	1500	5,792	10,767	127	/org.eclips
Weighted methods per Class (avg/max per typ	583	36,438	22,671	96	/org.eclips
Number of Overridden Methods (avg/max per	17	1,062	0,658	3	/org.eclips
Number of Static Attributes (avg/max per type	21	1,312	1,31	4	/org.eclips
Nested Block Depth (avg/max per method)		1,139	0,469	4	/org.eclips
Number of Methods (avg/max per type)	256	16	10,137	51	/org.eclips

Number of Static Methods	type
Total Lines of Code	compilationUnit
Afferent Coupling	packageFragment
Normalized Distance	packageFragment
Number of Classes	compilationUnit
Specialization Index	type
Instability	packageFragment
Number of Attributes	type
Number of Packages	packageFragmentR
New Methods Lines of Code	method
Weighted methods per Class	type
Number of Overridden Methods	type
Number of Static Attributes	type
Nested Block Depth	method
Number of Methods	type
Lack of Cohesion of Methods	type
McCabe Cyclomatic Complexity	method
Number of Parameters	method
Abstractness	packageFragment
Number of Interfaces	compilationUnit

Number of Classes : Total number of classes in the selected scope

Number of Children : Total number of direct subclasses of a class. ...

Number of Interfaces : Total number of interfaces in the selected scope

Depth of Inheritance Tree (DIT) : Distance from class Object in the inheritance hierarchy.

Number of Overridden Methods (NORM) : Total number of methods in the selected ...

McCabe Cyclomatic Complexity : Counts the number of flows through a piece of code.

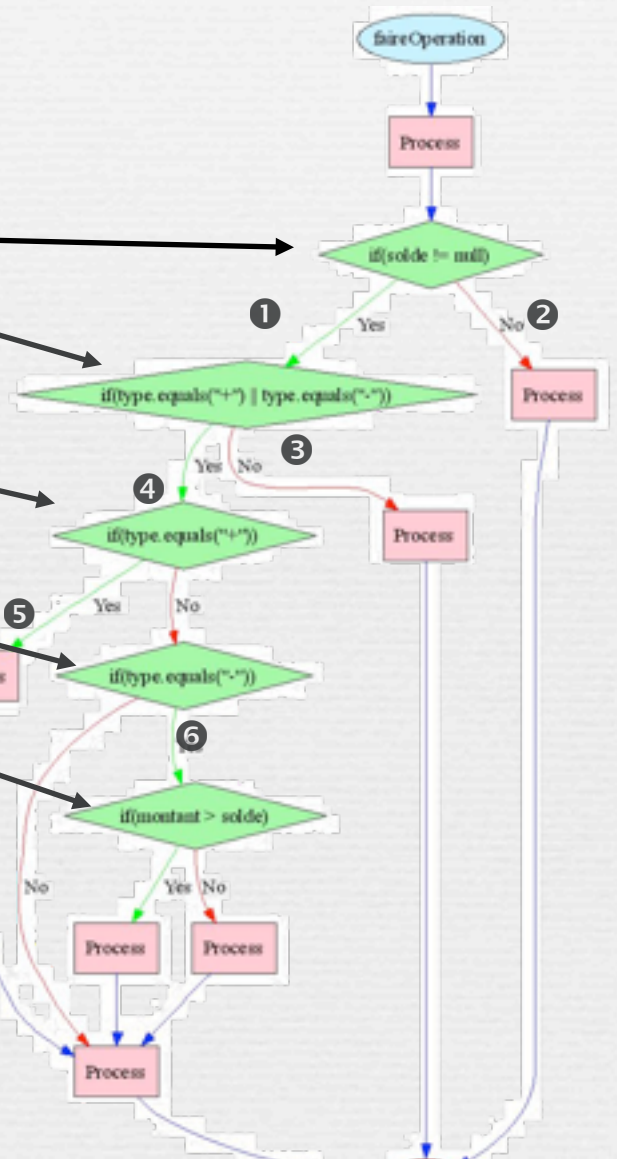
Weighted Methods per Class (WMC) : Sum of the McCabe Cyclomatic Complexity for all methods in a class

Assurer une bonne lisibilité du code

- **Complexité cyclomatique d'une méthode**
  - Définition
    - Nombre de chemins linéairement indépendants qu'il est possible d'emprunter dans cette méthode
      - Nombre de points de décision de la méthode : if, case, while,… + 1 (le chemin principal)
  - Interprétation
    - Une méthode avec une haute complexité cyclomatique est plus difficile à comprendre et à maintenir
      - > à 30 : refactoriser la méthode
      - < à 30 : acceptable si suffisamment de tests
    - La complexité cyclomatique est liée à la notion de « couverture de code"
      - Nombre de tests unitaires = sa complexité cyclomatique
        - **Un chemin = un test**

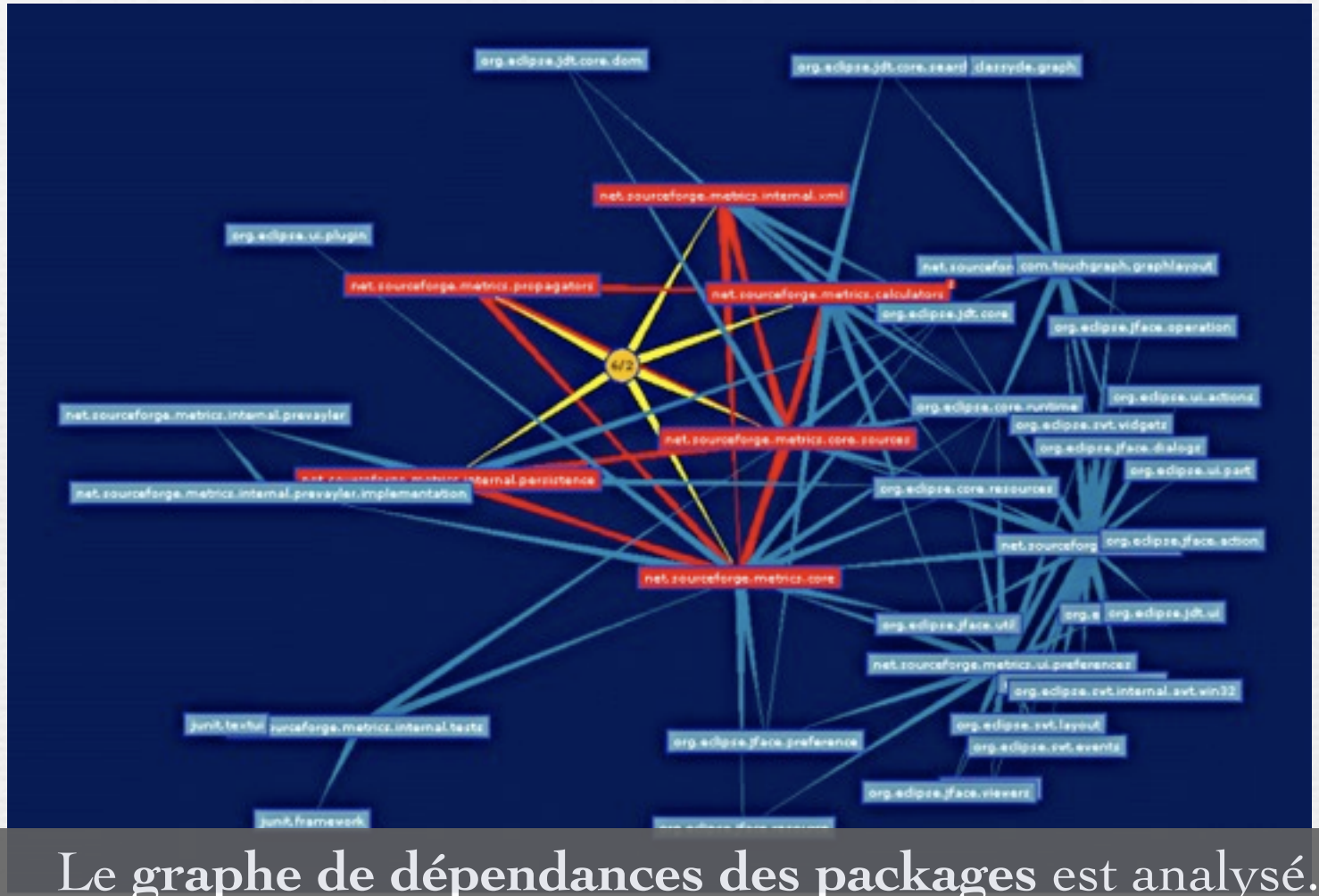
Complexité cyclomatique d'une méthode

```
package banque;


public class Banque {
   private Double solde;

   public void faireOperation(
        String type, double montant) {
      System.out.println("Début d'opération.");
      if(solde != null) {
       if(type.equals("+") || type.equals("-"))
       {
        if(type.equals("+")) {
          solde += montant;
        }
        if(type.equals("-")) {
          if(montant > solde) {
            System.err.println("!!!");
          }
          else {
            solde -= montant;
          }
        }
       }
       else {
        System.err.println("...");
       }
      }
      else {
       System.err.println(".");
      }
    System.out.println("Fin d'opération.");
   }
}
```



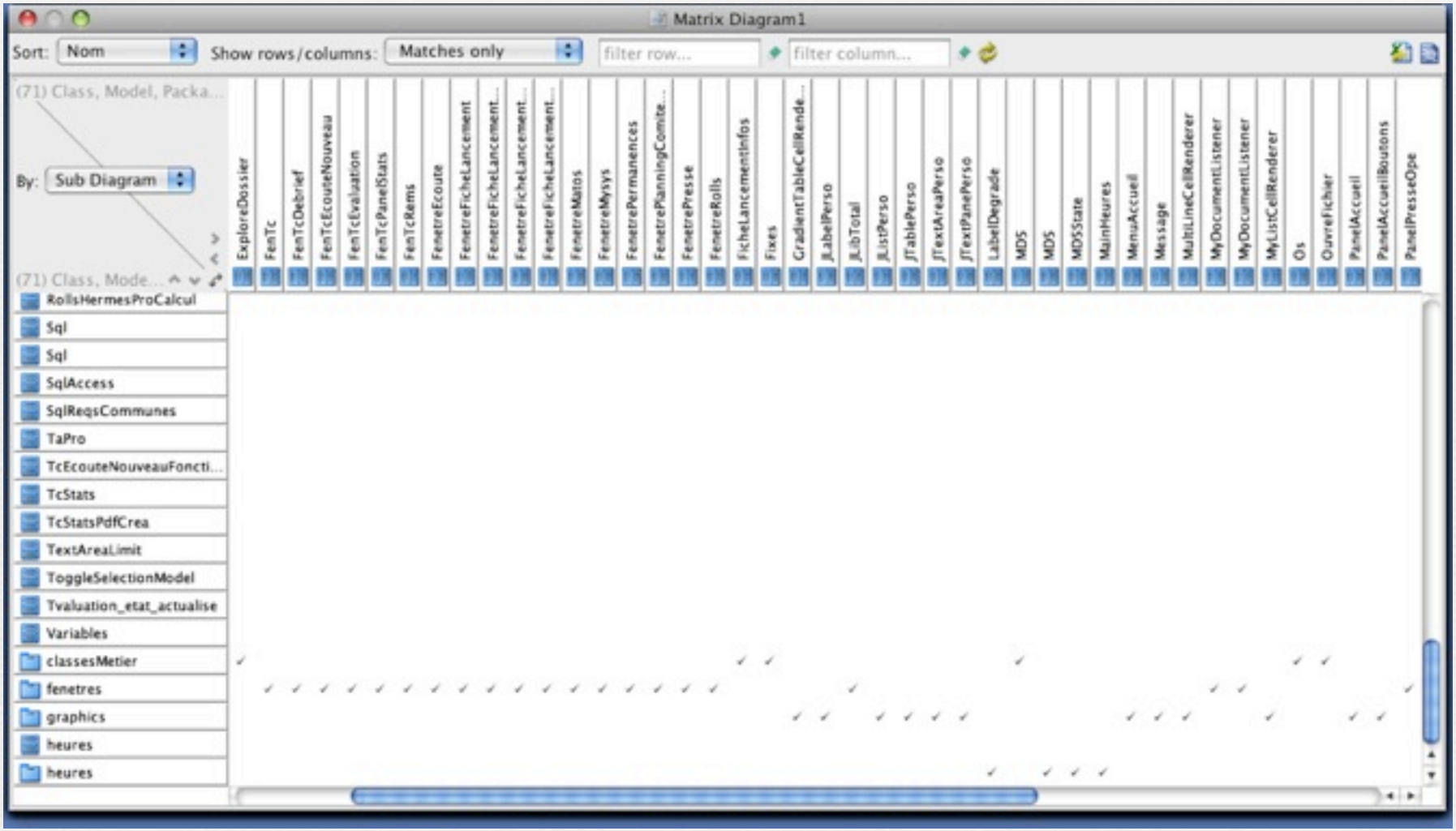
Tests statiques

(plugin Metrics : http://metrics.sourceforge.net/)



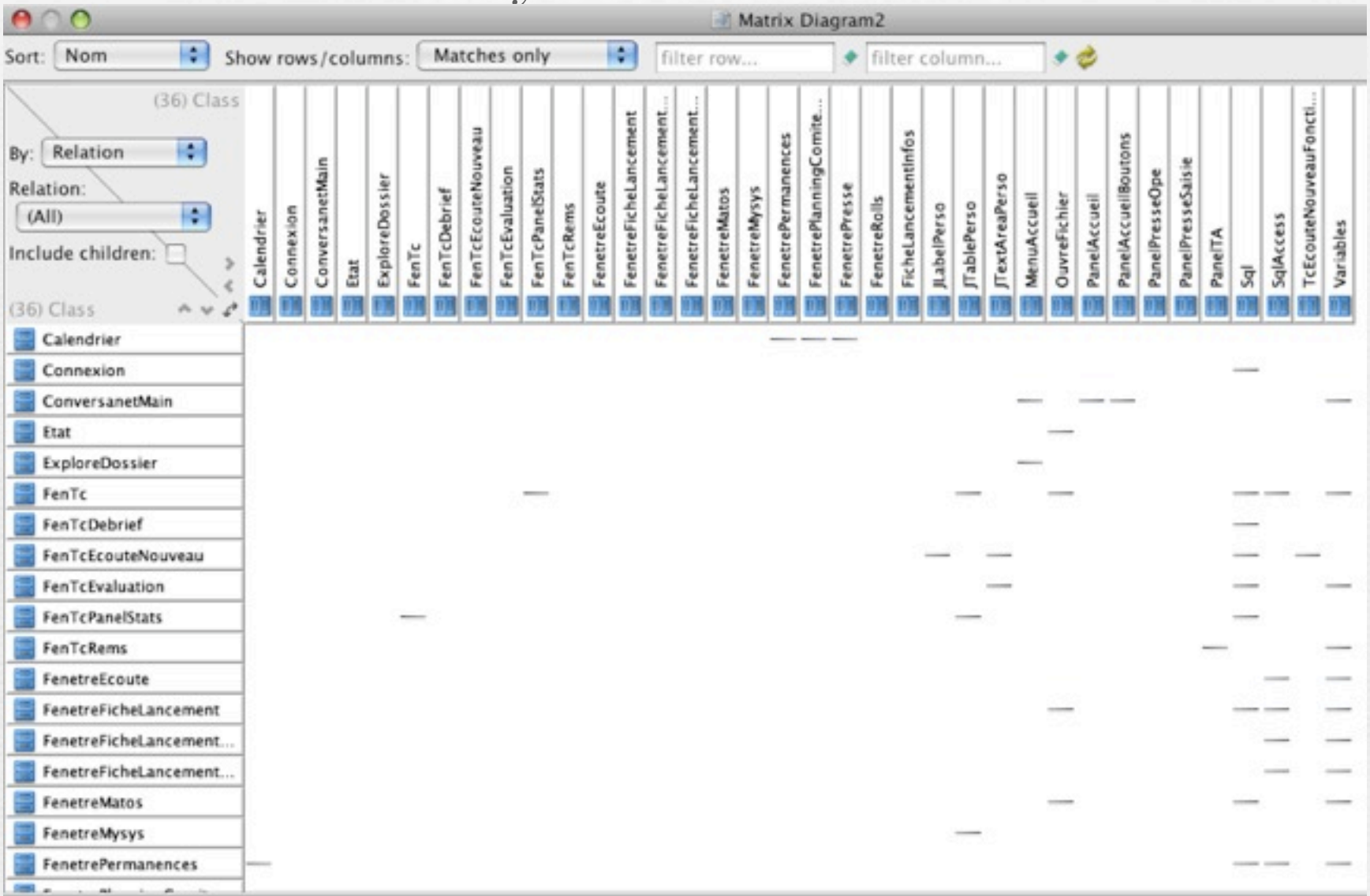
Le **graphe de dépendances des packages** est analysé.
Les packages fortement connectés sont colorés en rouge. Les autres packages ne participant pas à des cycles sont de couleur bleue.

Tests statiques
Analyse de modèles avec VP

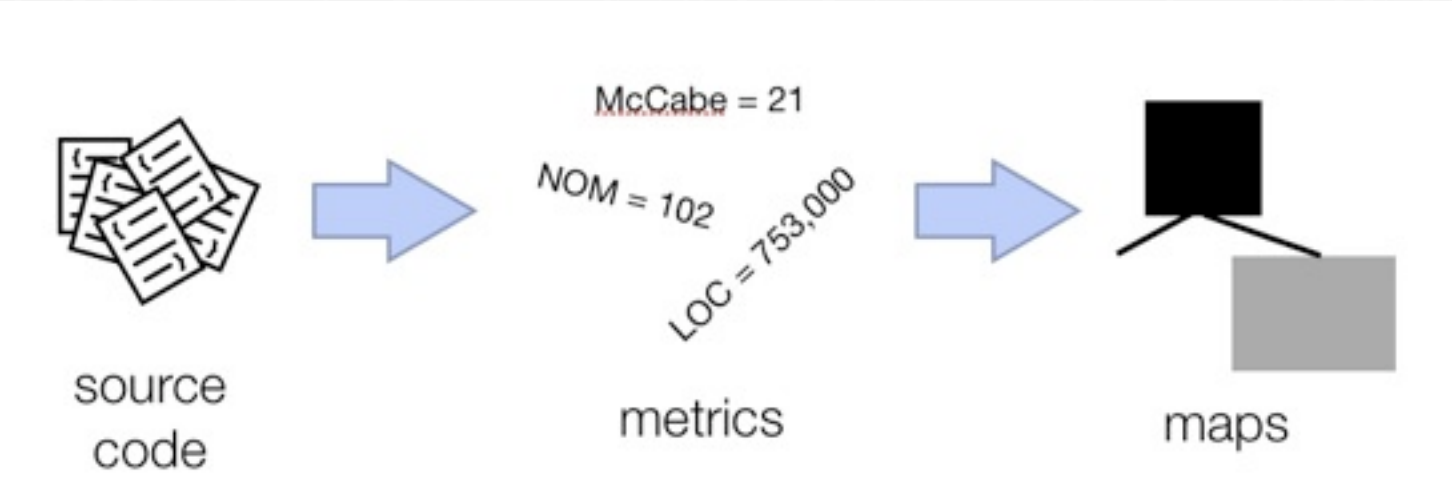


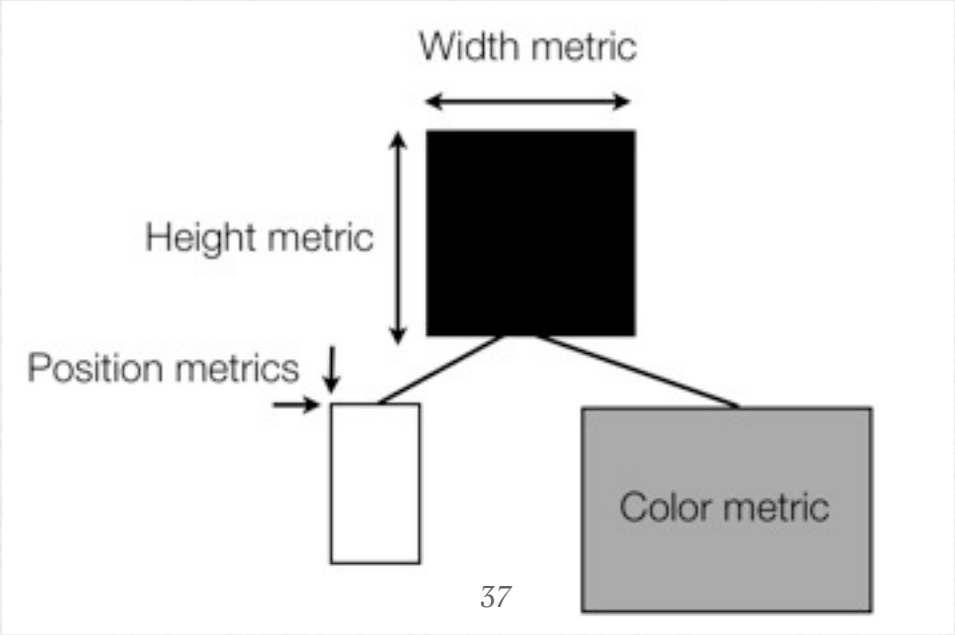
Tests statiques

Analyse de modèles avec VP

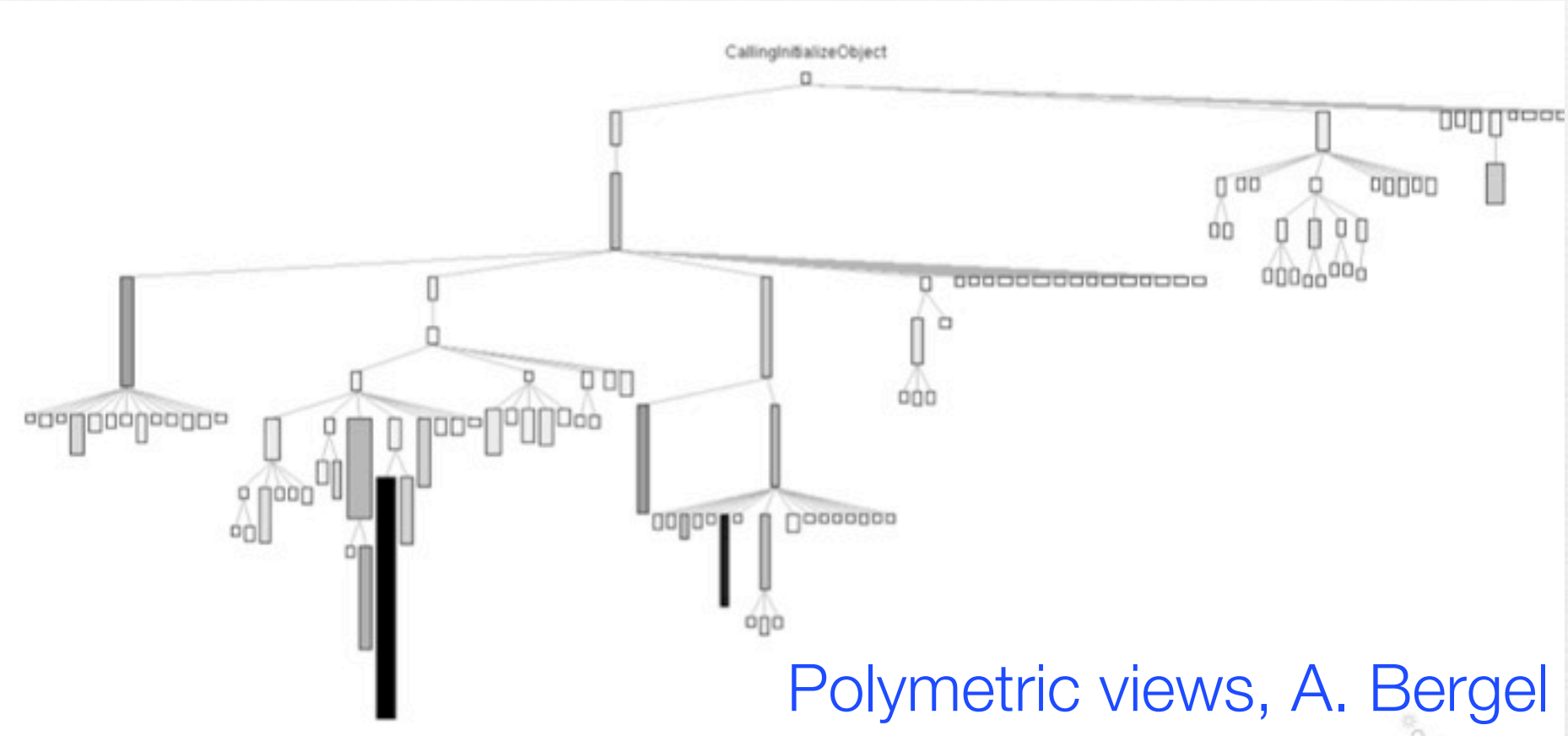


Tests statiques
Autres Visualisations



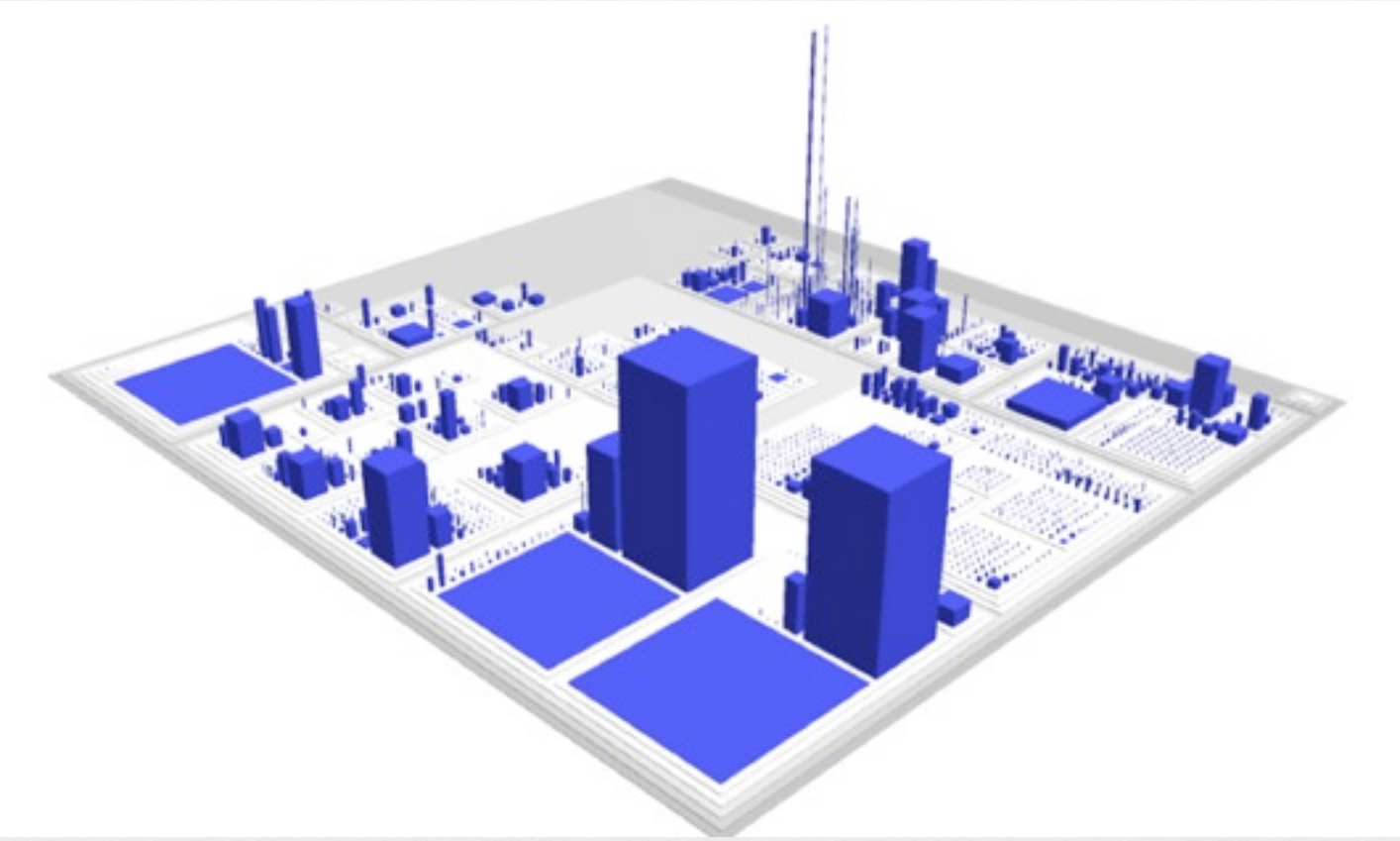


Tests statiques
Autres Visualisations



Polymetric views, A. Bergel

Tests statiques
Autres Visualisations



CodeCity (Wettel, Univ Lugano)

Limites du test

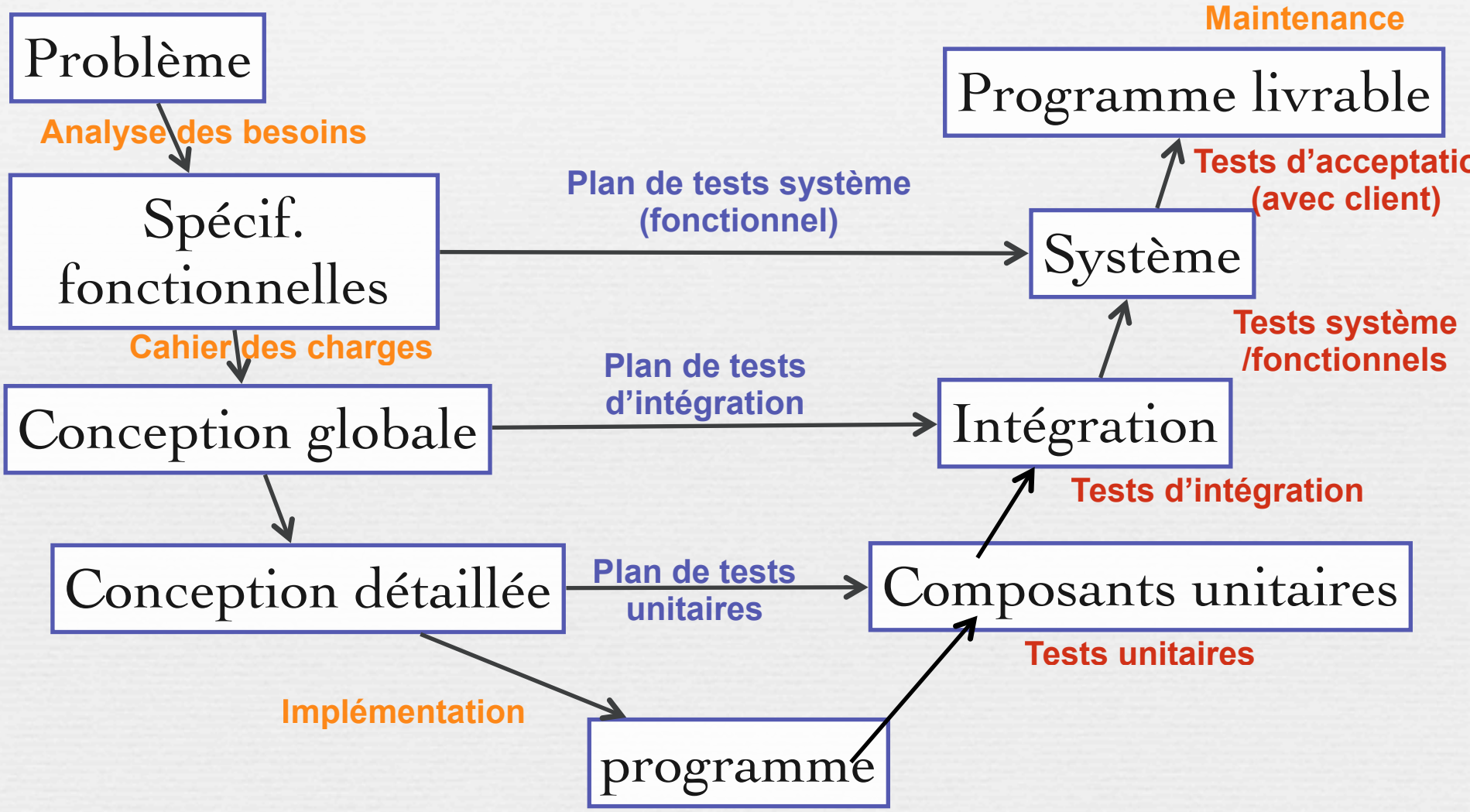
- L'espace des entrées -La *complexité*
- Les séquences d'exécution -La *complexité*
- Sensibilité aux fautes –Transformation entre le *fini* et *l'infini*
- Tester un programme permet de montrer la présence de fautes mais en aucun cas leur absence.
- Les tests basés sur une *implémentation* ne peuvent révéler des omissions car le code manquant ne peut pas être testé
- Comment être sûr qu'un système de test est correct … il faut le tester?

Hiérarchisation et Types de Tests

Types de tests

- Tests système/fonctionnels (test boîte noire)
  - Utilise la description des fonctionnalités du programme
  - Provenant des spécifications (scenarii, uses cases)
- Tests structurels (test boîte blanche)
  - Utilise la structure interne du programme
- Tests de (non) régression (après modifications)
  - Correction et évolution ne créent pas d'anomalies nouvelles
- Tests de robustesse
  - Cas de tests correspondant à des entrées non valides
- Tests de performance (application intégrée dans son environnement)
  - load testing : résistance à la montée en charge
  - stress testing : résistance aux demandes de ressources anormales

Hiérarchisation des tests dans le cycle en V

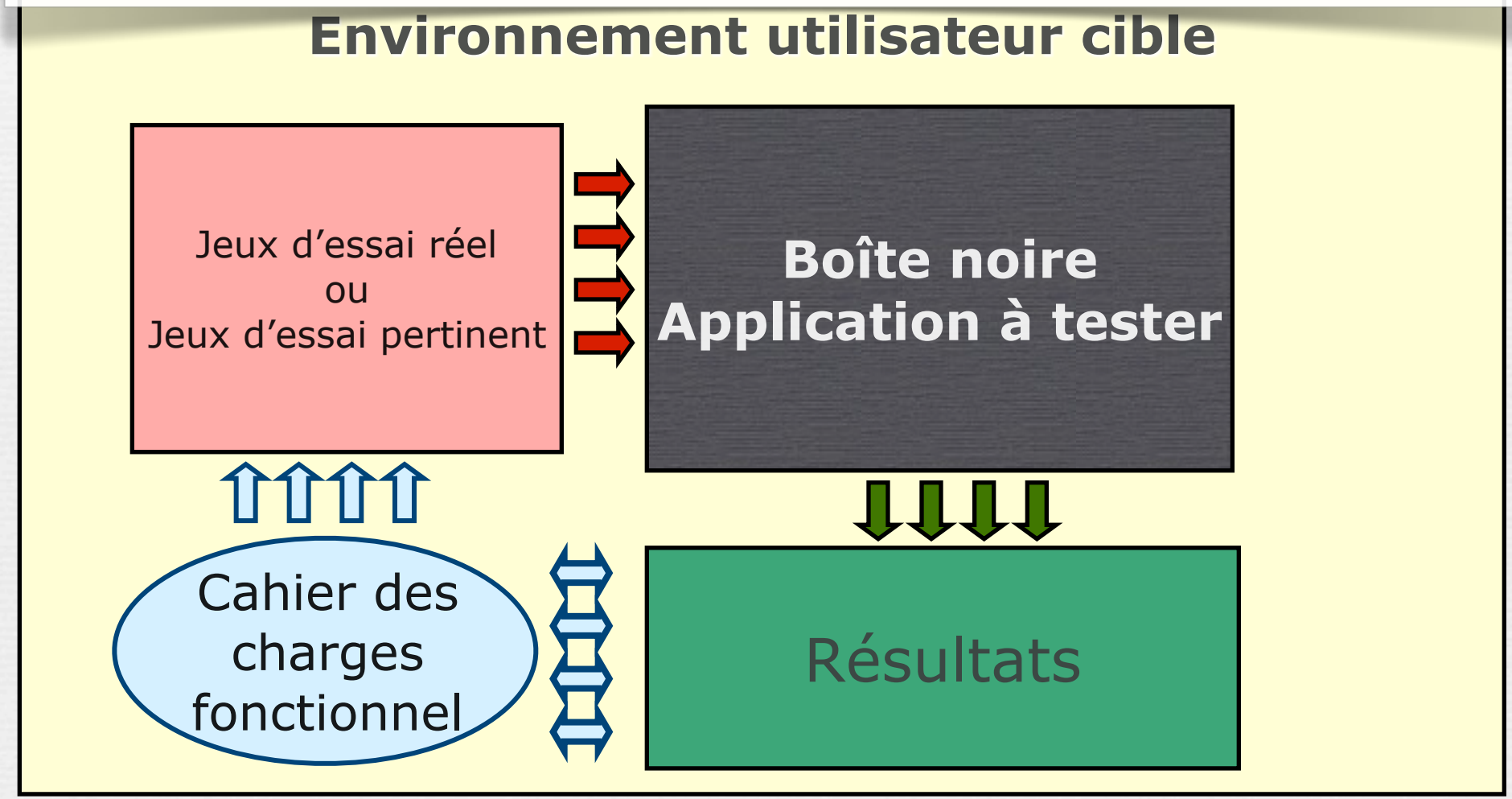


Diagrammes UML et Tests

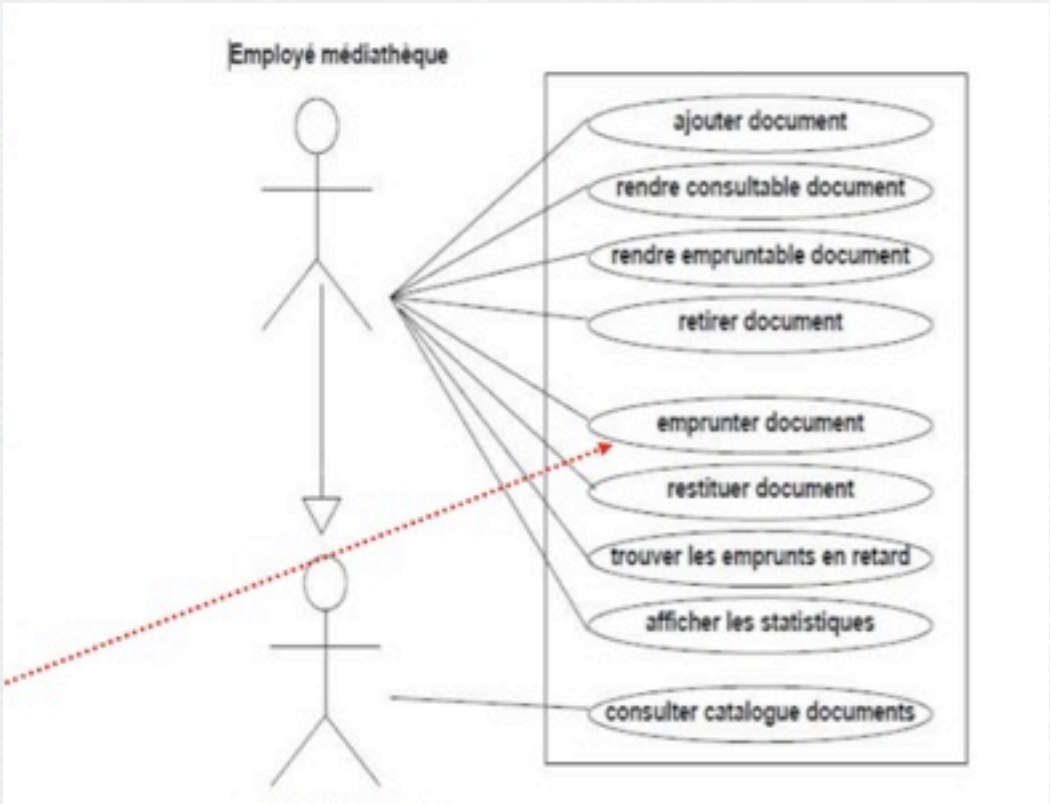
- Niveau Application (spécification)
  - Diagramme des cas d'utilisation — Tests Système/Fonctionnel
  - Diagramme de classes : test des associations, des agrégations
    - multiplicité,
    - création, destruction — Tests d'intégration
  - Diagramme de séquence : test de séquences
    - construction d'un graphe de flot
- Niveau Classes (conception détaillée)
  - Classes détaillées — Tests d'intégration
  - Diagrammes de machine à états — Tests unitaire

Test Système (fonctionnel)

vérifier que les fonctions correspondant aux attentes sont *bien atteintes*



Tests fonctionnels & UML



Tests fonctionnels & UML

Données d'entrée

client: peut être inscrit ou non;

emprunts: déja effectués par le client

- existe-t-il un emprunt en retard ?
- le nombre d'emprunts déja effectués correspond-il au nombre maximum de ce client ?

document:

- existe?
- empruntable ou consultable?,
- déjà emprunté ou disponible?

Tests fonctionnels & UML

Données de sortie
`Emprunt` accepté ou refusé.

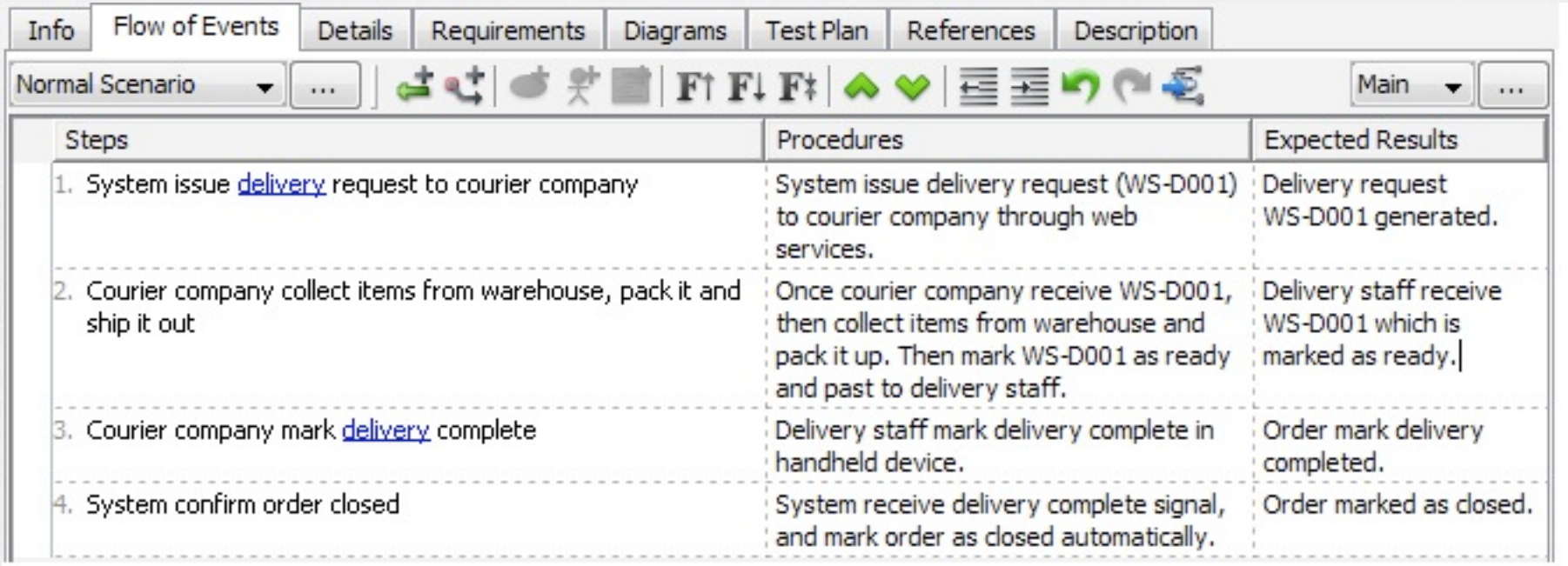
Remarque: la définition des jeux de tests de validation pour le cas d'utilisation `emprunter document` permet de soulever au moins les questions suivantes (à poser au client):

- un abonné qui n'est pas à jour de sa cotisation peut-il tout de même emprunter un document?
- doit-il être considéré comme un client au tarif normal tant qu'il n'a pas renouvelé son abonnement?
- ou doit-il se réabonner avant de pouvoir emprunter un document?

D'une manière générale, la préparation des jeux de tests permet de lever les ambiguïtés et réparer des oublis.

Use cases et Tests

- Pour chaque d'utilisation, sélectionner les scénarios et définir les tests correspondants.



http://www.visual-paradigm.com/product/vpuml/tutorials/testingprocedure.jsp

Tests fonctionnels...

http://seleniumhq.org/docs/

http://watir.com

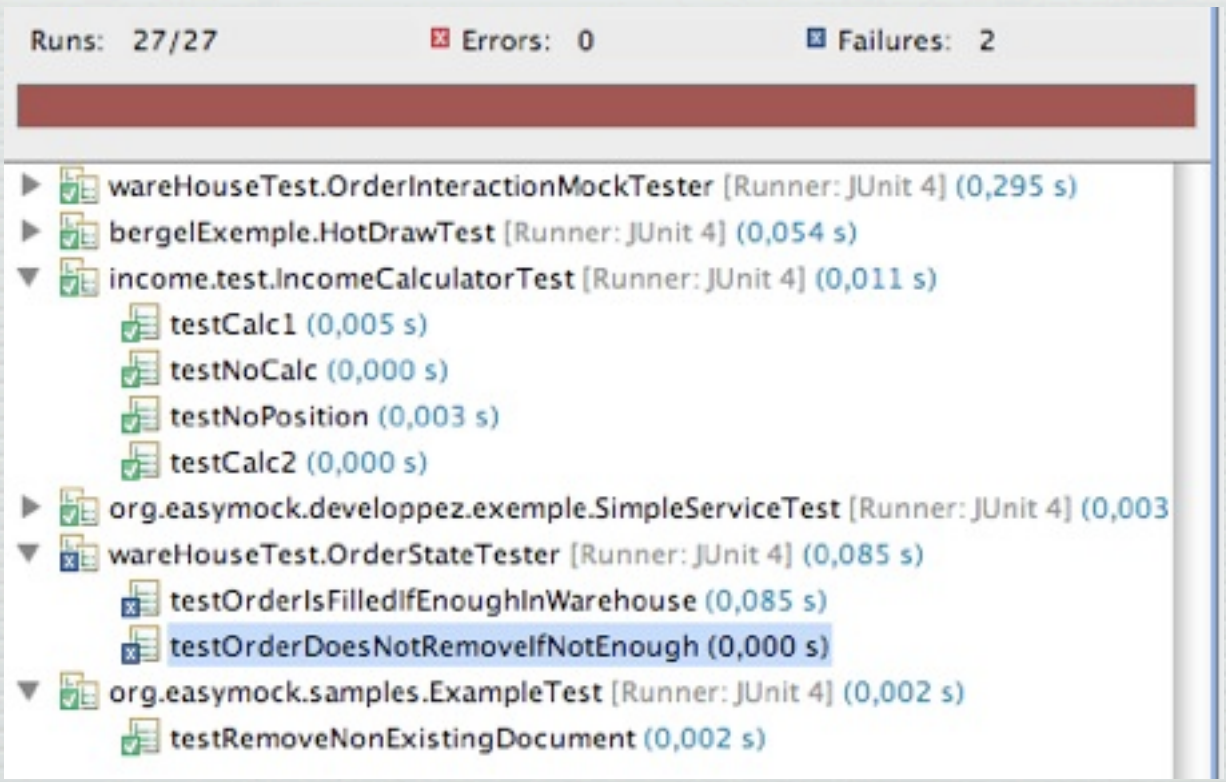
Une démonstration...

http://www.opensourcetesting.org

Types de Tests

Tests Unitaires

exemple JUnit



Tests Unitaires (déjà étudiés en AP)

- Tester une unité logicielle isolée du reste du système
  - Plus petit composant compilable
  - Pour un langage procédural : unité de test = procédure
  - Pour un langage objet : unité de test = classe
  - Test de **l'interface**
  - Les unités sont-elles suffisamment spécifiées ?
  - Le code est-il lisible, maintenable ...?

- Importance de la notion de contrats (pré/post)

Types de Tests

Tests d'intégration

Tests d'intégration

✓Différents modules d'une application peuvent fonctionner unitairement, leur intégration, entre eux ou avec des services tiers, peut engendrer des dysfonctionnements.

✓Il est souvent impossible de réaliser les tests unitaires dans l'environnement cible avec la totalité des modules à disposition.

➡Les tests d'intégration ont pour objectif de créer une version complète et cohérente du logiciel (avec l'intégralité des modules testés unitairement) et de garantir sa bonne exécution dans l'environnement cible.

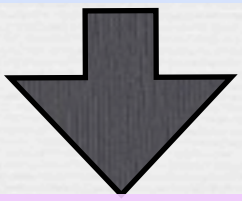
Tests d'intégration

Objectif

Vérifier les interactions entre composants unitaires

Difficultés principales de l'intégration

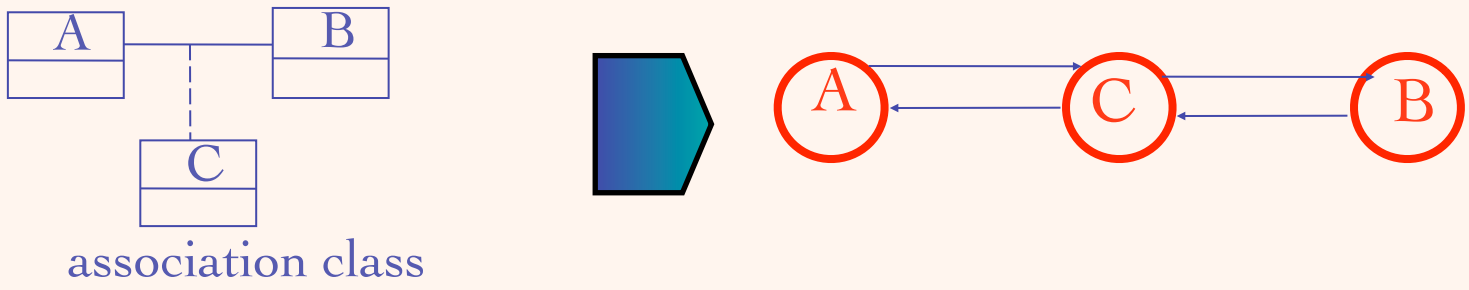
- Interfaces floues
- Implantation non conforme à la spécification
- Réutilisation de composants

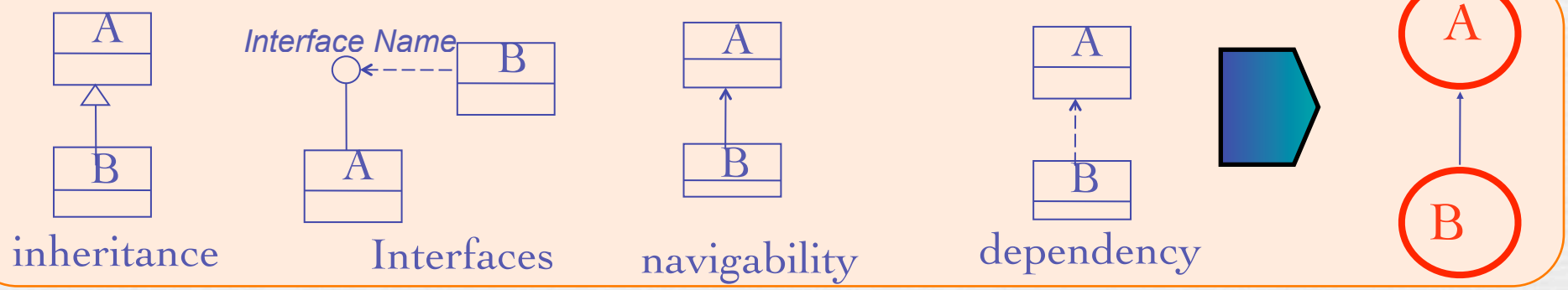


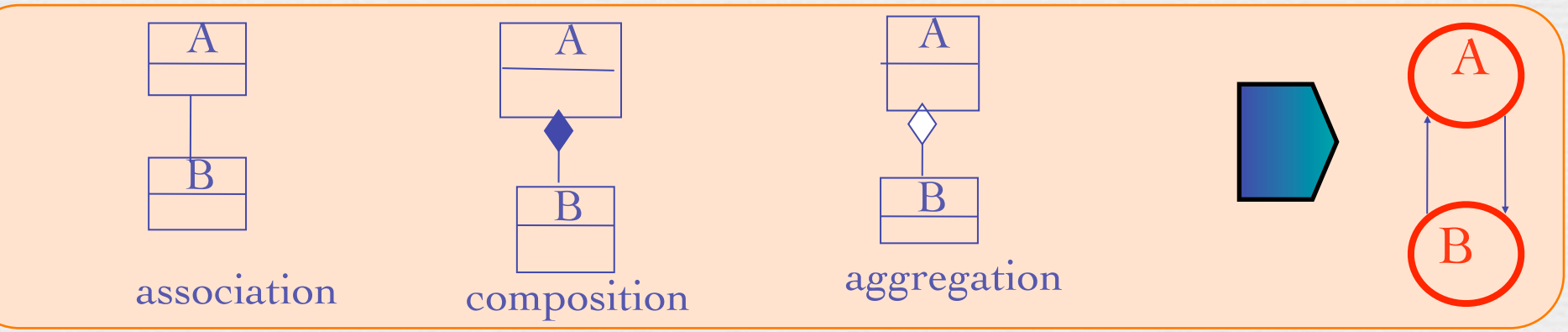
1) modéliser la structure de dépendances entre chaque composant et son environnement (graphe de dépendance des tests)

2) Choisir un ordre pour intégrer (assembler)

Graphe de dépendance : construction







Compilateur GNU pour Eiffel :

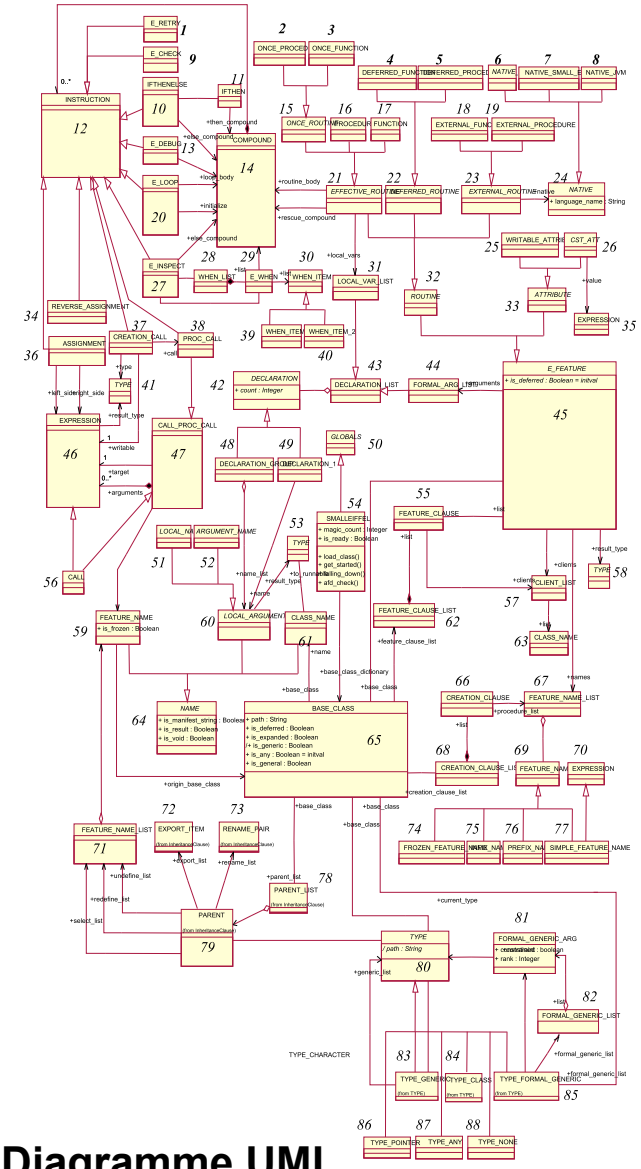
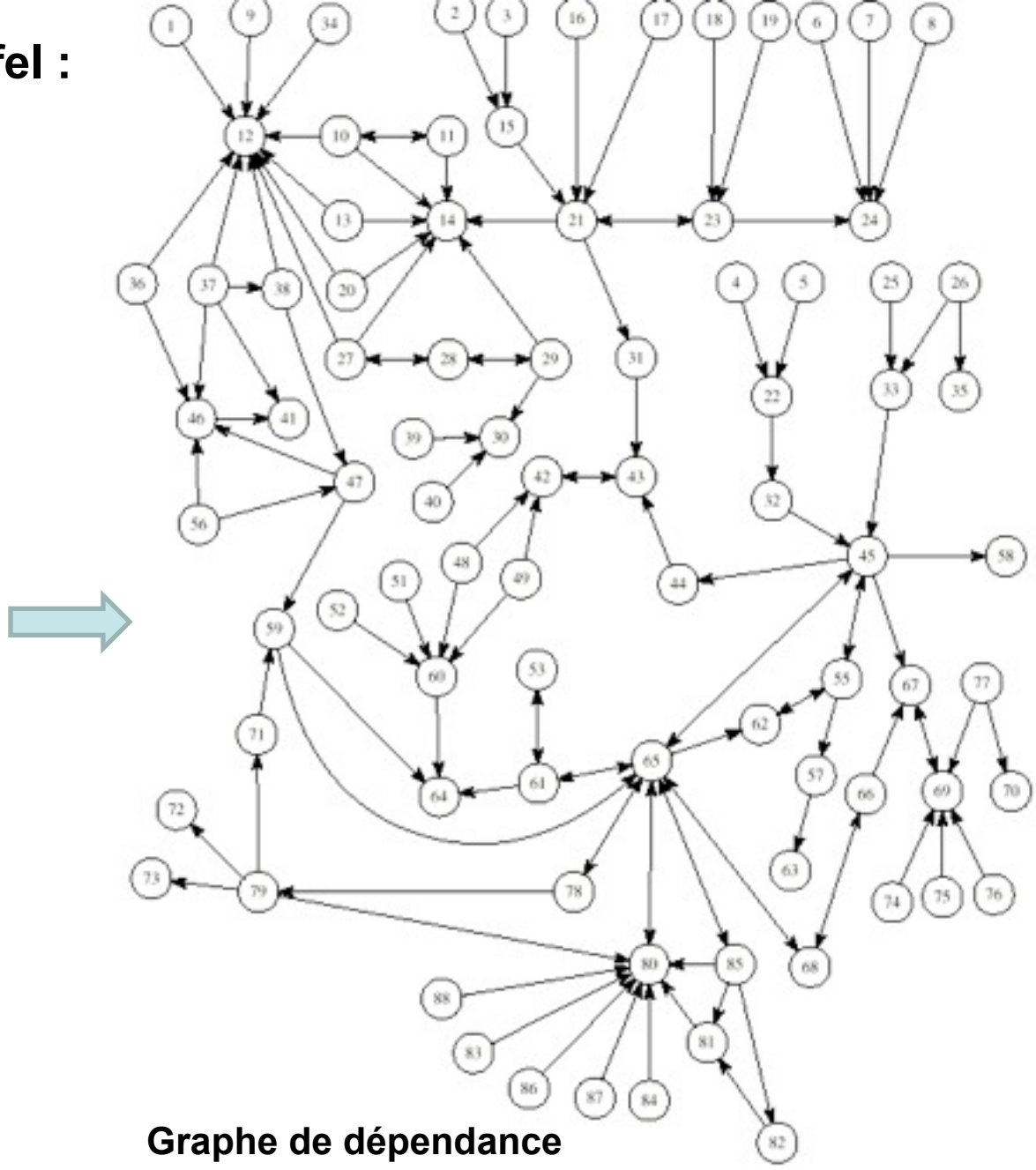


Diagramme UML



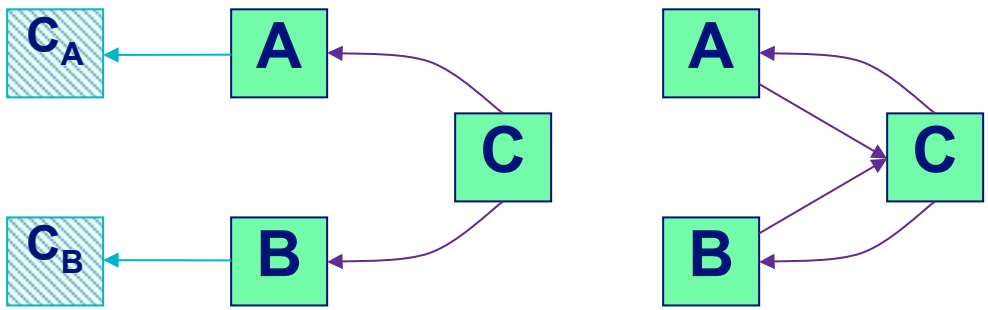
Graphe de dépendance

Test d'intégration : les interdépendances

- Une solution simple consiste à contraindre le concepteur
  - pas de boucle dans l'architecture
    - c'est souvent possible
    - **mais** les optimisations locales ne sont pas toujours optimales globalement
    - **mais** concevoir des composants interdépendants est souvent naturel

Bouchon de test

- Bouchon : une unité qui
  - osimule le comportement d'une unité

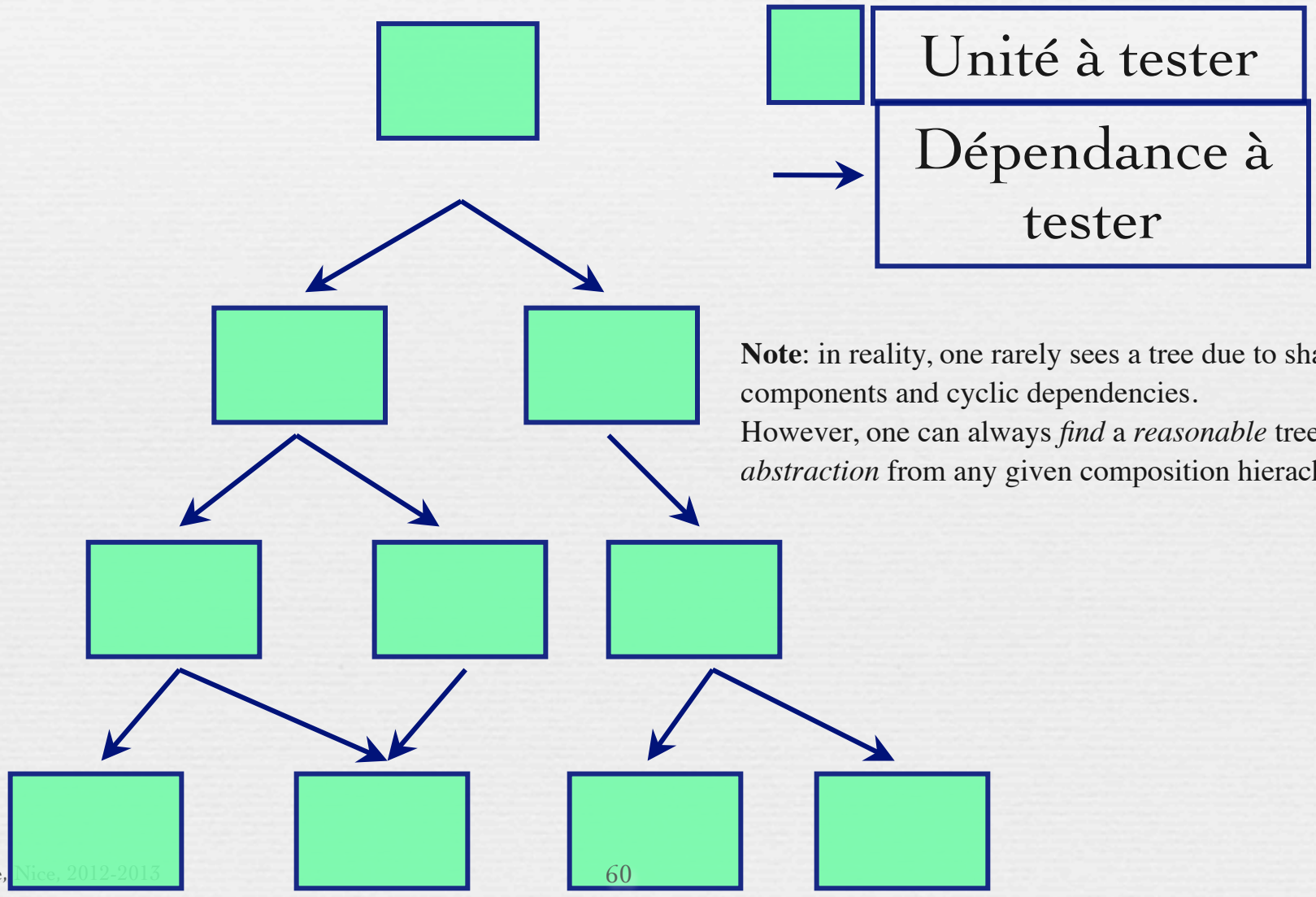


Bouchon spécifique CFC

Diffusion Libre

Tests d'intégration

➡Architecture des dépendances



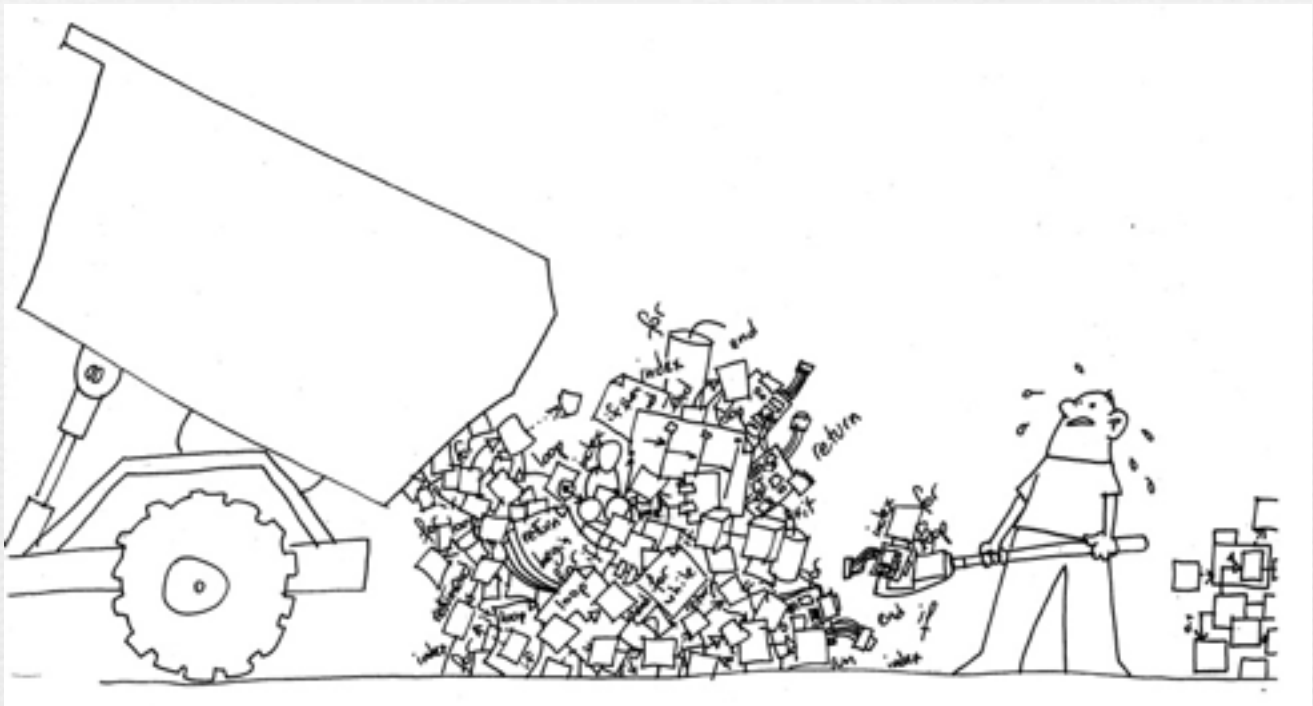
Note: in reality, one rarely sees a tree due to shared components and cyclic dependencies.
However, one can always *find* a *reasonable* tree *abstraction* from any given composition hierachy.

Stratégies

- Big-bang : tout est testé ensemble (peu recommandé)
- Top-down (peu courant)
- Bottom-up (la plus classique)

Intégration - Big-Bang

- ***Big Bang*** – Validation du système –

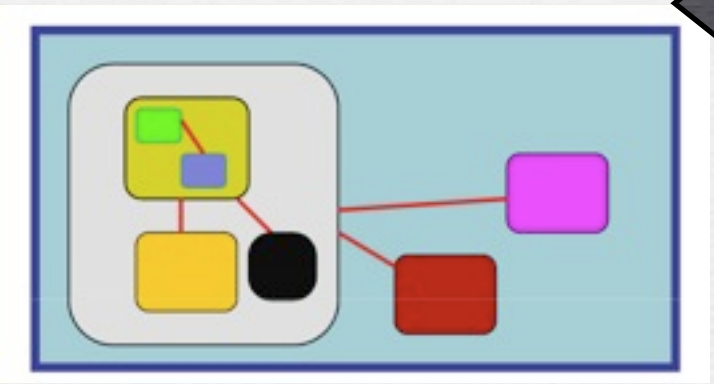


http://emmanuelchenu.blogspot.com/

Intégration - Big-Bang

Intégration de tous les composants à tester en une seule étape. (intégration massive)

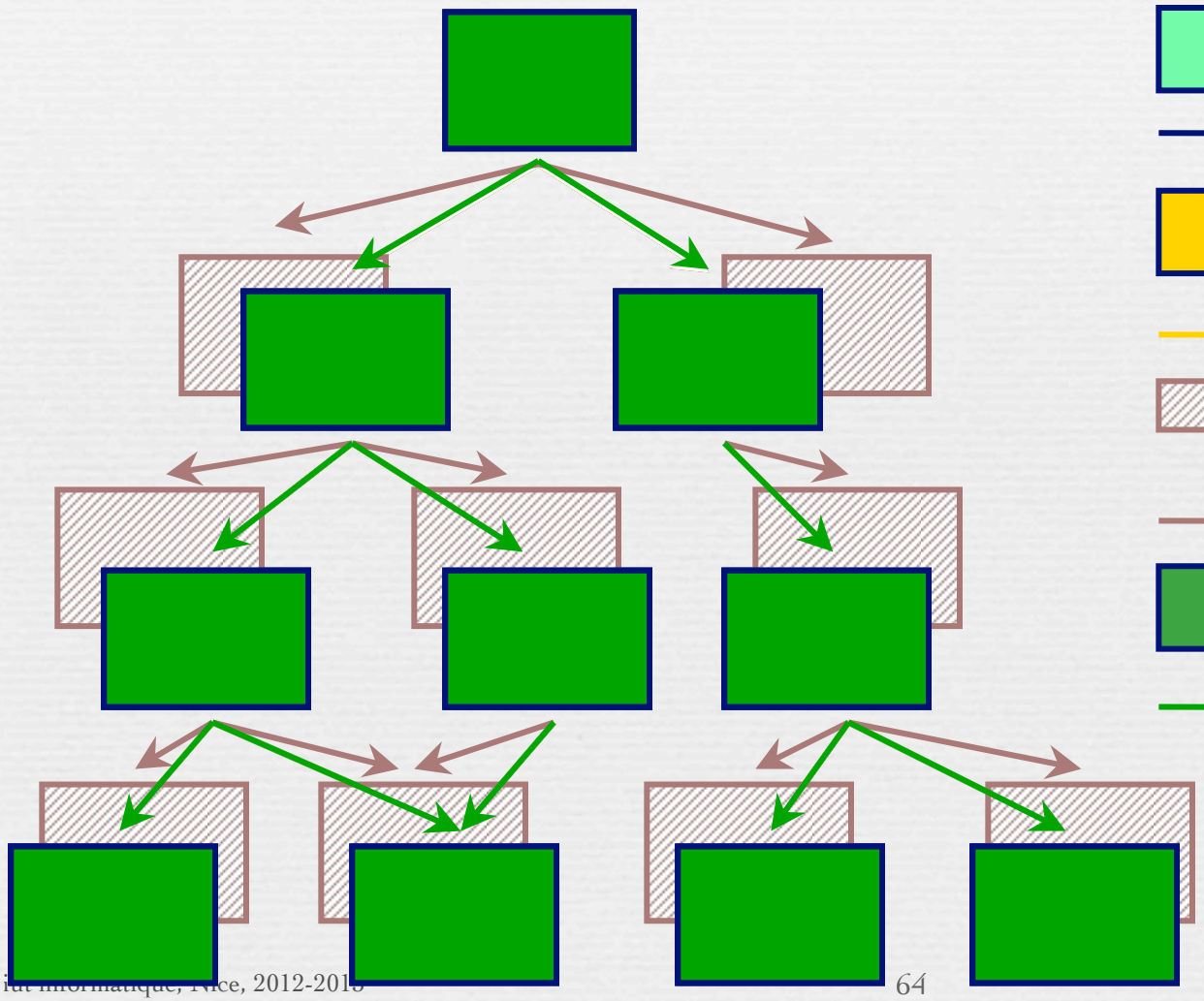
Tests à l'interface du système

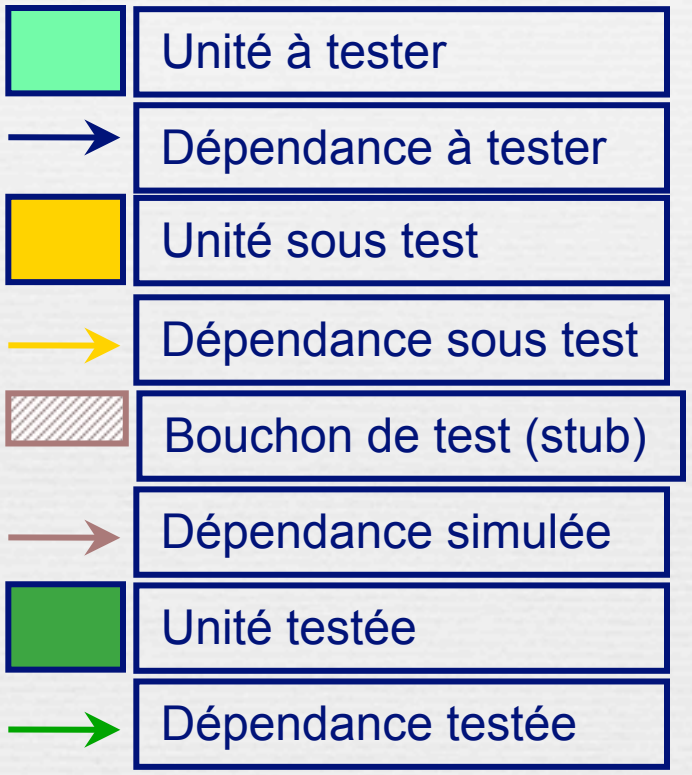


Principaux Problèmes:

- Les tests produisent des erreurs : Quelle en est la cause?
- La complexité induit des tests manquants
- Les tests ne commencent que lorsque tous les composants ont été «codés».

Approche descendante



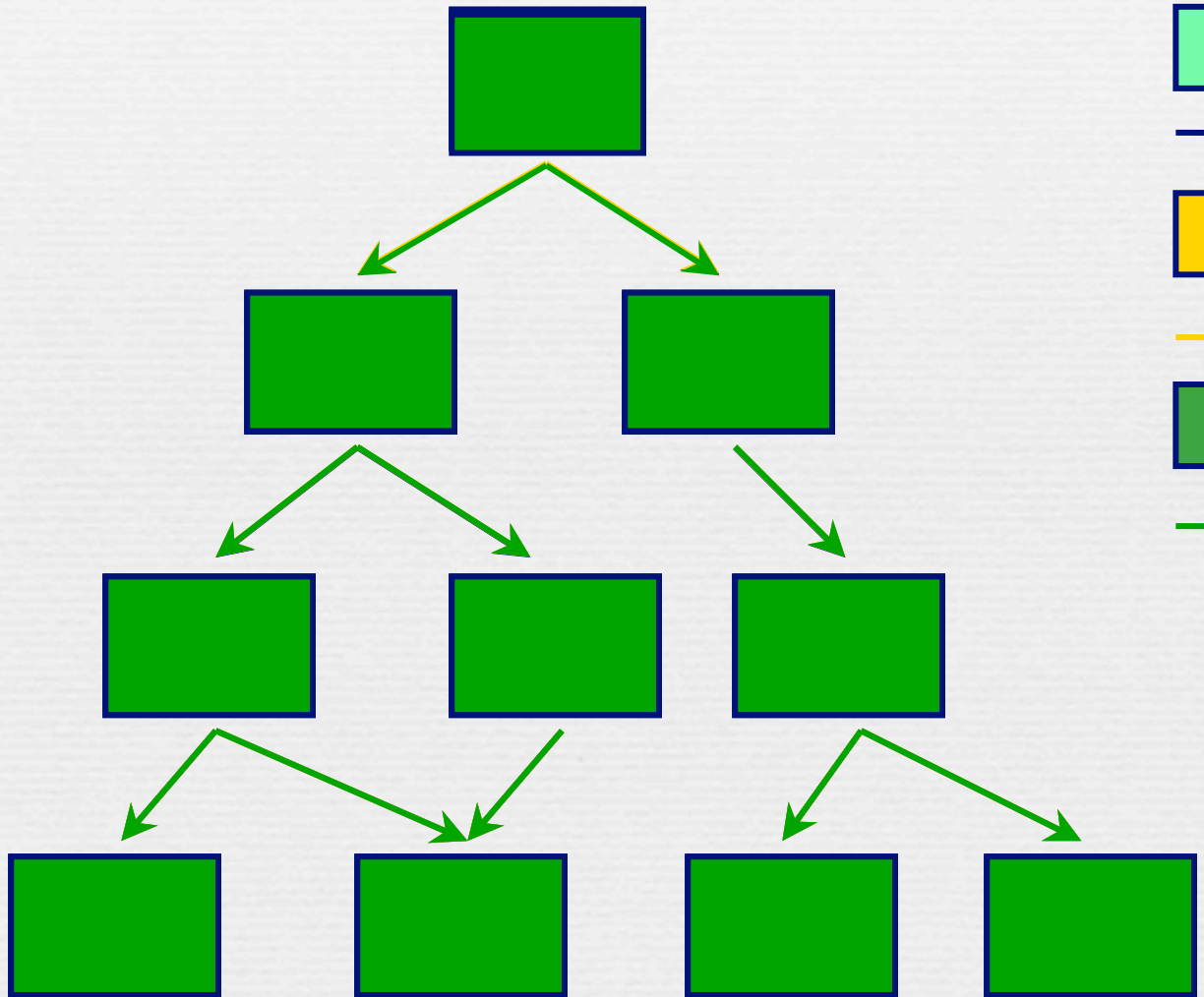


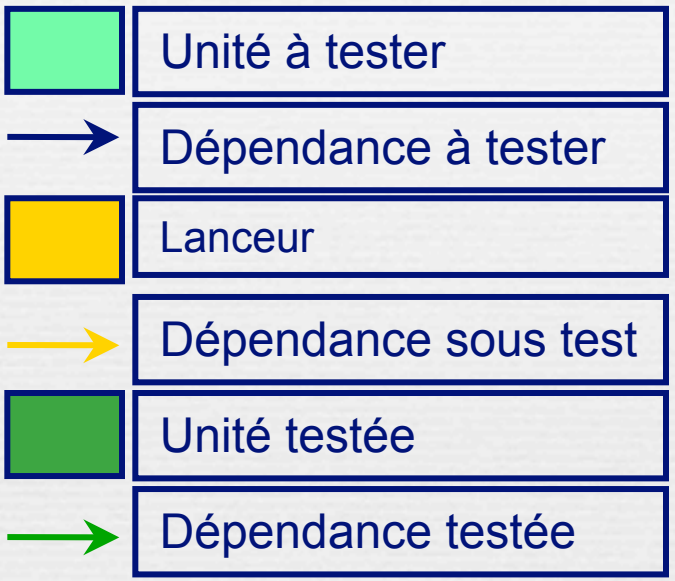
Approche descendante

- Création de **bouchons**
- Test tardif des couches basses
- Détection précoce des défauts d'architecture

- Effort important de simulation des composants absents et multiplie le risque d'erreurs lors du remplacement des bouchons.
- La simulation par « couches » n'est pas obligatoire

Approche Ascendante





Approche ascendante

- **Avantages**
  - Faible effort de simulation
  - Construction progressive de l'application s'appuie sur les modules réels. Pas de version provisoire du logiciel
  - Les composants de bas niveau sont les plus testés,
  - Définition des jeux d'essais plus aisée
  - Démarche est naturelle.
- **Inconvénients**
  - Détection tardive des erreurs majeures
  - Planification dépendante de la disponibilité des composants

Approche Mixte

- Combinaison des approches descendante et ascendante.
- **Avantages** :
  - Suivre le planning de développement de sorte que les premiers composants terminés soient intégrés en premier ,
  - Prise en compte du risque lié à un composant de sorte que les composants les plus critiques puissent être intégrés en premier.

- La principale difficulté d'une intégration mixte réside dans sa complexité car il faut alors gérer intelligemment sa stratégie de test afin de concilier les deux modes d'intégration : ascendante et descendante.

Test d'intégration

voir Stub/Mocks

En conclusion, pourquoi un développement dirigé par les tests?(1)

Les programmeurs n'aiment pas tester.

➡ Ils testeront relativement bien la première fois.

- La deuxième fois, cependant, les tests sont généralement moins approfondis
- La troisième fois, ..

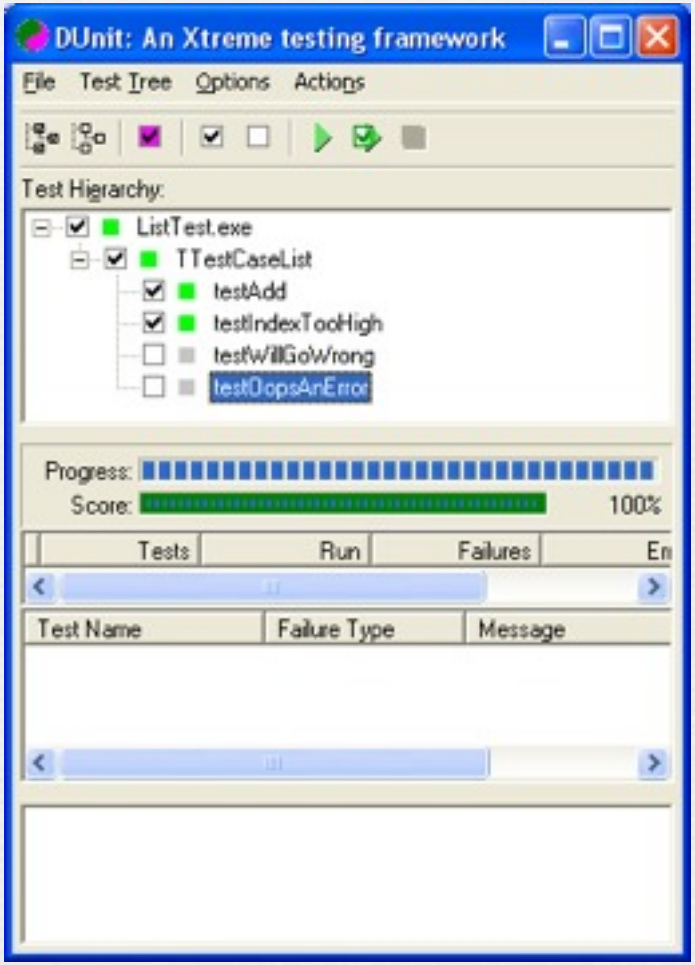
➡Le test est considéré comme une tâche ennuyeuse qui pourrait être le travail de quelqu'un d'autre!

✓DDT (TDD) encourage les programmeurs à maintenir un ensemble exhaustif de tests reproductibles

- ✓ Ils sont supportés par des outils qui permettent à la fois une exécution sélective et automatisée des tests.
- ✓Les tests peuvent être exécutés après chaque changement

En conclusion, pourquoi un développement dirigé par les tests? (2)

- Bob Martin:
  - "L'acte d'écrire un test unitaire est plus un acte de conception que de vérification"
- Renforcer la confiance
  - En pratiquant TDD, les développeurs vont s'efforcer d'améliorer leur code - sans la peur qui est normalement associée à des modifications du code
- Supprimer / réduire le recours au débogueur
  - Plus de "debug plus tard" attitudes



S'organiser pour tester

- Module contenant les tests indépendant du code « réel »
- Faire des tests indépendants les uns des autres
  - Pas de tests imbriqués
  - Un test qui échoue ne fait pas échouer les tests suivants
- Automatiser les tests
  - Ant ou Makefile : compiler et lancer automatiquement tous les tests
  - Utiliser un Environnement de tests : framework JUnit
  - Tests Unitaires/Tests de non régression
- Réutiliser les tests et leurs résultats comme documentation

TDD conclusion (1)

- **Modéliser est support aux tests :**
  - ✓ Tests fonctionnels => use cases et scenarios, Tests d'intégration et structure, ...
- **Coder / tester, coder / tester…**
  - ✓ Lancer les tests aussi souvent que possible (aussi souvent que le compilateur !)
- Commencer par écrire **les tests au moins sur les parties les plus critiques**
  - ✓ Ecrire les tests qui ont le meilleur retour sur investissement !
- Quand on **ajoute des fonctionnalités**, on écrit **d'abord les tests**
- Quand on trouve **un bug**, écrire **un test qui le caractérise**
  - ✓ Principe d'**intégration continue**

> Pendant le développement, le programme marche toujours, peut être ne fait-il pas tout ce qui est requis, mais ce qu'il fait, il le fait bien !

"Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead." -- Martin Fowler

Les Tests sont un tuteur

http://emmanuelchenu.blogspot.com/

Encore plus en amont, le «preprototype» ;-)

Pretotyping	Prototyping
• **Investment:** hours, days	• **Investment:** days, weeks
• **Main Q:** Would we use it?	• **Main Q:** Can we build it?
• **Deliverable:** [Working] pretotype	• **Deliverable:** Working prototype

Palm «pre»

- ➡now beats later
- ➡doing beats talking
- ➡simple beats complex
- ➡commitment beats committees

http://www.pretotyping.org/the-pretotyping-manifesto-1