Test de logiciel dans les méthodes agiles

Appliqué au contexte objet (Java)

1 – Aspects « théoriques »

2 – Aspects pratiques – le développement dirigé par les tests

en partie inspiré d'un cours de Laurie Williams @2004
http://agile.csc.ncsu.edu/SEMaterials/AgileTesting.pdf

Franck Ledoux genie.logiciel.evry@gmail.com

Introduction

- Il est nécessaire d'assurer la fiabilité des logiciels
  - Domaines critiques : atteindre une très haute qualité imposée par les lois/normes/assurances/
  - Autres domaines : atteindre le rapport qualité/prix jugé optimal (attentes du client)
- Assurer la fiabilité du logiciel est une part cruciale du développement
  - Plus de 50% du développement d'un logiciel critique
  - Plus de 30% du développement d'un logiciel standard
- Plusieurs études ont été conduites et indiquent que
  - Entre 30 et 85 erreurs sont introduites par portion de 1000 lignes de code produites (logiciel standard).
- Ces chiffres peuvent fortement augmenter si les méthodes permettant d'assurer la fiabilité sont mal gérées au sein du projet
  - Voir la part importante de l'activité de tests dans les différentes méthodes de conception de logiciels

Introduction

- Dans le cycle de vie du logiciel, on assure la fiabilité / qualité du logiciel par l'activité de Vérification et de Validation

  - Vérification
    - le développement est-il correct par rapport à la spécification initiale ?
    - Est-ce que le logiciel fonctionne correctement?

  - Validation
    - a-t-on décrit le « bon » système ?
    - est-ce que le logiciel fait ce que le client veut ?

- Les tests ont un rôle prépondérant dans les méthodes agiles

Règle agile : écrire les tests avant de coder

Introduction

- Les principales méthodes de validation et vérification
  - Test statique
    - examen ou analyse du texte
    - Revue de code, de spécifications, de documents de design
  - Test dynamique
    - Exécuter le code pour s'assurer d'un fonctionnement correct
    - Exécution sur un sous-ensemble fini de données possibles
  - Vérification symbolique
    - Run-time checking
    - Exécution symbolique, …
  - Vérification formelle :
    - Preuve ou model-checking d'un modèle formel
    - Raffinement et génération de code

Actuellement, le test dynamique est la méthode la plus répandue et utilisée.

dans ce cours, **test = test dynamique**

Plan du cours

- Définitions autour du test
  - Métriques de qualité/fiabilité logicielle
  - Positionnement du test dans le cycle de vie du logiciel
  - Test et méthodes agiles
  - Qu'est-ce que le test ?
- Processus de test simplifié
- Comment choisir les scénarios de test à jouer ?
  - Les méthodes de test Boîte Noire
  - Les méthodes de test Boîte Blanche
- Tests unitaires avec JUnit et EclEmma
  - Présentation de JUnit
  - Couverture de test avec EclEmma
- Utilisation de doublures (mocks) pour les tests unitaires avant intégration

Outils utilisés

- *Eclipse*
  - Environnement de développement intégré
  - www.eclipse.org
- *JUnit*
  - Canevas pour les test unitaires
  - www.junit.org
- *Emma*
  - Outil d'analyse de couverture structurel
  - emma.sourceforge.net
  - *EclEmma, plugin eclipse (help→Eclipse Marketplace)*
- *Mockito*
  - Outil pour la gestion de mock/doublures/simulacres
  - http://code.google.com/p/mockito/
  - Récupérer la dernière release mockito-all-1.9.0-rc1.jar

Qu'est-ce que tester ?

- Tester c'est réaliser l'exécution du programme pour y trouver des défauts et non pas pour démontrer que le programme ne contient plus d'erreur
  - Le but d'un testeur est de trouver des défauts à un logiciel
  - Si ce n'est pas son but, il ne trouve pas
    → aspect psychologique du test
  - C'est pourquoi il est bon que
    - Le testeur ne soit pas le développeur
      - souvent difficile
      - Valable que pour certains types de tests
    - Les tests soient écrits avant le développement (règle usuelle des méthodes agiles)
      - cf. le développement dirigé par les tests (test-driven development) proné par l'eXtrem Programming

Qu'est-ce que tester ?

- En pratique : ces idées font leur chemin
  - Cycle en V : même entreprise, mais équipes différentes
  - XP : tests unitaires par le développeur, mais tests de recettes par le client
  - logiciels critiques : tests de recettes validés par une agence externe
  - Microsoft / Google / etc. : beta-testing par les futur usagers

Si vous devez ne retenir qu'une chose de ce cours, ce doit être que ***vous devez croire que votre code contient des erreurs*** avant de le tester

Que recherche-t-on avec le test ?

Défaut : mauvais point dans le logiciel (conception, codage, etc.)

Faute (= bug) : défaut amenant un comportement fautif *observable*

- La différence est qu'il existe toujours des défauts mais qu'ils ne sont pas forcément observables
  - ex : fonction boguée mais jamais appelée

- Veut-on éviter les fautes ou les défauts ?
  - on veut éviter les fautes, mais souvent on cherche les défauts
  - objectif irréaliste : 0 défaut
  - objectif réaliste : pas de faute sévère, pas de faute récurrente (w.r.t. attentes usagers)

Traits caractéristiques d'un test

- Définition IEEE

> Le test est l'exécution ou l'évaluation d'un système ou d'un composant par des moyens automatiques ou manuels, pour vérifier qu'il répond à ses spécifications ou identifier les différences entre les résultats attendus et les résultats obtenus
>
> *(Standard Glossary of Software Engineering Terminology)*

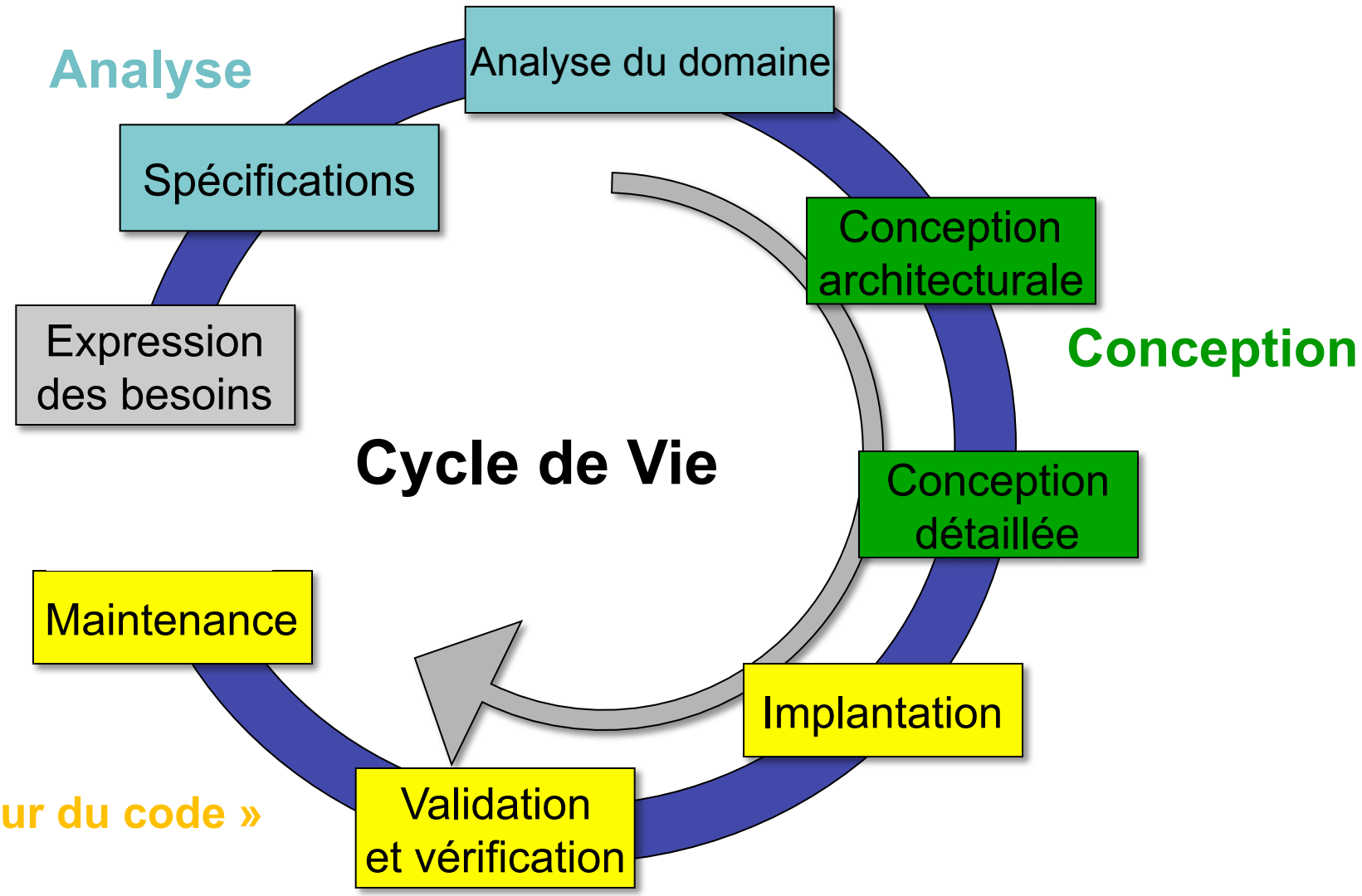
- Le test est une méthode **dynamique** visant à trouver des bugs

> Tester, c'est exécuter le programme dans l'intention d'y trouver des anomalies ou des défauts
>
> *G. J. Myers (The Art of Software Testing, 1979)*

- Le test est une méthode de validation **partielle** des logiciels

> Tester permet seulement de révéler la présence d'erreurs mais jamais leur absence.
>
> *E. W. Dijkstra (Notes on Structured Programming, 1972)*

Le test dans le cycle de vie du logiciel



eXtrem Programming et test

- La méthodologie XP est centrée sur le test et met en avant deux pratiques de test complémentaires
  - Le développement dirigé par les tests (Beck, 2003)
  - Les test d'acceptation par la clientèle

- Dans tous les cas, le test doit être automatisé!!
  - Test = code
  - Les tests doivent pouvoir être rejoués sans cesse et sans effort pour le développeur

    Kaner, Bach, et al, Lessons Learned in Software Testing, New York, John Wiley and Sons, Inc. 2002.
  - Avantages
    - Confiance dans le nouveau code qui ne « casse » pas l'existant et fait ce qu'il doit
    - Permet de comprendre quelles parties des fonctionnalités ont réellement été implantées
    - Forme naturellement un base de non régression

Test Driven Development

- Le développement dirigé par les tests est une pratique basée sur
  - Les tests unitaires
  - L'écriture des tests par les développeurs
  - L'écriture des tests avant celle du code

- L'écriture du code se fait suivant 6 itérations rapides
  1. Conception rapide (optionnelle)
  2. Ecriture d'un petit nombre de test unitaires **automatisés**
  3. Lancement des tests pour vérifier qu'ils échouent (car pas encore de code correspondant)
  4. Implantation du code à tester
  5. Lancement des tests jusqu'à ce qu'ils passent tous
  6. Restructuration du code et des tests si nécessaire pour améliorer les performances ou le design

- En XP, le principe d'itérations rapides « écriture des tests/ écriture du code » souvent en binôme remplace une grande partie des phases de conception

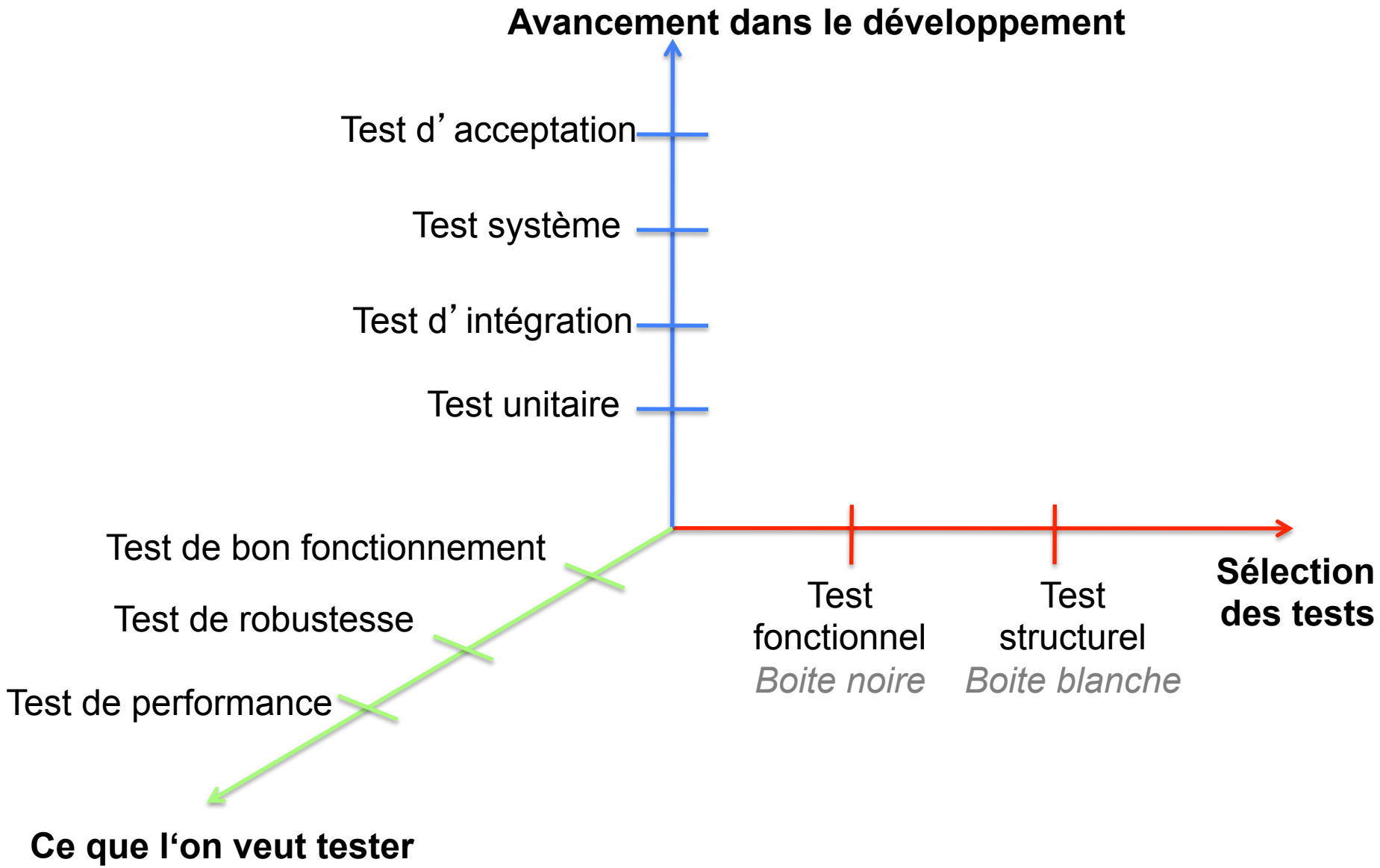
Comment écrire des tests automatiques

- Suivre le « Test Automation Manifesto »

  Meszaros, Smith, et al, The Test Automation Manifesto, Extreme Programming and Agile Methods – XP/Agile Universe 2003, LNCS 2753, Springer.

- Les tests automatisés doivent être
  - Concis – écrit aussi simplement que possible
  - Auto-vérifiable – aucun intervention humaine ne doit être nécessaire
  - Répétable – toujours sans intervention humaine
  - Robuste – un test doit toujours produire le même résultat
  - Suffisants – ils doivent couvrir tous les besoins du système
  - Nécessaires – pas de redondances dans les tests
  - Clairs – faciles à comprendre
  - Spécifiques – en particulier chaque test d'échec pointe un point précis
  - Indépendants – ne dépendent pas de l'ordre d'exécution des tests
  - Maintenables – ils doivent pouvoir être aisément modifiés
  - Traçables

Quels types de tests ?



Le test dans le cycle de vie du logiciel

Selon l'avancement du développement, on distingue plusieurs types de test

- Le ***test unitaire*** consiste à tester des composants isolément
  - Test de fonctions, du comportement d'une classe
  - Doivent répondre à des scénarios de test préalablement établis
  - Généralement effectué dans les environnements de développement (éclipse+Junit, Visual Studio)
  - Facile à mette en œuvre
  - Cible : les méthodes et les classes

- Le ***test d'intégration*** consiste à tester un ensemble de composants qui viennent d'être assemblés. Il comporte 2 types de tests
  - Les *test de validité* – le nouveau module répond à ce qui a été demandé lorsqu'on l'intègre dans le système global
  - Les *tests de non régression* – L'intégration du nouveau module n'induit pas de régressions fonctionnelles (modification du comportement des modules existants) ou non fonctionnelles (instabilité) du système dans sa globalité
  - Cible : les composants et le système global

Le test dans le cycle de vie du logiciel

- Le ***test système*** consiste à tester le système sur son futur site d'exploitation dans des conditions opérationnelles et au-delà (surcharge, défaillance matérielle,…)
  - Cible : le système global sur l'architecture informatique du client

- Le ***test d'acceptation*** est la dernière étape avant l'acceptation du système et sa mise en œuvre opérationnelle
  - On teste le système dans les conditions définies par le futur utilisateur, plutôt que par le développeur.
  - Les tests d'acceptation révèlent souvent des omissions ou des erreurs dans la définition de besoins
    - Donc des erreurs de validation et non de vérification

Dans tous les cas, pour assurer la traçabilité et la reproductibilité des tests (lors des étapes de maintenance), on doit fournir des documents indiquant les modules testés, les scénarios de test (entrées, environnements, etc.) et les résultats des tests

Le test dans le cycle de vie du logiciel

- Résumé

Test unitaire	Tester les modules, classes, opérations, au fur et à mesure de leur développement	Fait par le développeur du module (avant ou après écriture)
Test d'intégration	Tester l'assemblage des modules, des composants et leurs interfaces	Fait par le développeur du module ou un autre développeur
Test de système	Tester les fonctionnalités du système dans son ensemble	Développeurs
Test d'acceptation	Confirmer que le système est prêt à être livré et installé	Client

Un autre type de test important :les tests de non régression!!!

- Que faire quand le programme a été modifié ?
- **Réponse :** rejouer les tests
- **Problèmes :**
  1. maintenance : scripts de tests peuvent ne plus fonctionner
  2. Trop de tests à rejouer
- **Solutions à (1) :** infrastructure de gestion des tests
- **Solutions à (2) :**
  - sélection des tests pertinents,
  - Minimisation du jeu de tests,
  - Prioritisation
- Les solutions à ces problèmes sont bien automatisées

Types de test – ce que l'on veut tester!

- **Tests nominal ou test de bon fonctionnement**
  - Les cas de test correspondent à des données d'entrée valide.

    Test-to-pass

- **Tests de robustesse**
  - Les cas de test correspondent à des données d'entrée invalides

    Test-to-fail

- Règle usuelle : Les tests nominaux sont passés avant les tests de robustesse

- **Tests de performance**
  - *Load testing* (test avec montée en charge)
  - *Stress testing* (soumis à des demandes de ressources anormales)

Difficulté du test

- Limite théorique = Notion d'indécidabilité
  - propriété indécidable = qu'on ne pourra **jamais prouver** dans le cas général (pas de procédé systématique)
- Exemples de propriétés indécidables
  - l'exécution d'un programme termine
  - deux programmes calculent la même chose
  - un programme n'a pas d'erreur
  - un paramètre du programme fait passer l'exécution
    - sur une instruction, une branche, un chemin donné
    - sur toutes les instructions, branches ou chemins
- Une bataille perdue d'avance
  - un programme a un nombre infini (ou immense) d'exécutions possibles
  - un jeu de tests n'examine qu'un nombre fini d'exécutions possibles
- Trouver des heuristiques :
  - approcher l'infini (ou le gigantesque) avec le fini (petit)
  - tester les exécutions les plus « représentatives »

Difficulté du test

- Le test exhaustif est en général impossible à réaliser
  - En test fonctionnel, l'ensemble des données d'entrée est en général infini ou très grande taille

    Exemple : la somme de 2 entiers 32 bits à raison de 1000 tests par seconde prendrait plus de 580 millions d'années
  - En test structurel, le parcours du graphe de flot de contrôle conduit à une forte explosion combinatoire
  - Conséquence
    - le test est une méthode de vérification partielle de logiciels
    - la qualité du test dépend de la pertinence du choix des données de test

Exemple célèbre (G.J. Myers, *The Art of Software Testing*, 1979)

> Un programme prend 3 entiers en entrée, qui sont interprétés comme représentant les longueurs des côtés d'un triangle. Le programme doit retourner un résultat précisant s'il s'agit d'un triangle scalène, isocèle ou équilatéral.

Quels cas test pour ce programme ?

Difficulté du test

14 cas possibles

- Cas scalène valide (1,2,3 et 2,5,10 ne sont pas valides par ex.)
- Cas équilatéral valide
- Cas isocèle valide (2,2,4 n'est pas valide)
- Cas isocèle valide avec les 3 permutations sur l'ordre des entrées (par ex. 3,3,4 – 3,4,3 – 4,3,3)
- Cas avec une valeur nulle
- Cas avec une valeur négative
- Cas où la somme de deux entrées est égale à la troisième entrée
- 3 cas pour le test 7 avec les trois permutations
- Cas où la somme de deux entrées est inférieure à la troisième
- 3 cas pour le test 9 avec les 3 permutations
- Cas avec les 3 entrées nulles
- Cas avec une entrée non entière
- Cas avec un nombre erroné d'entrée (2 ou 4)
- Pour chaque cas test, avez-vous défini le résultat attendu?

Difficulté du test

- Chacun de ces tests correspond à un défaut constaté dans des implantations de cet exemple
- La moyenne des résultats pour des développeurs expérimentés est de 7.8 sur 14
- Il faut donc bien admettre que le test exhaustif n'est pas possible mais aussi que la sélection des tests est une activité complexe (même sur un exemple simple)

Difficulté du test

- Difficultés d'ordre psychologique ou «culturel»
  - Le test est un processus destructif : un bon test est un test qui trouve une erreur alors que l'activité de programmation est un processus constructif - on cherche à établir des résultats corrects
  - Les erreurs peuvent être dues à des incompréhensions de spécifications ou de mauvais choix d'implantation
- L'activité de test s'inscrit dans le contrôle qualité, indépendant du développement

Des règles pratiques (1/2)

- Vos efforts pour les tests devraient toujours être axés pour essayer de faire *planter* votre code
  - Essayez des valeurs hors de la plage que vous avez spécifiée
  - Essayez des quantités d'information qui sont nulles, en trop, et des quantités moyennes.
  - Essayez des valeurs erronées (entrez une chaîne de caractères au lieu d'un nombre)…
- Les tests doivent donc tester les bornes pour lesquelles le programme a été écrit.

Des règles pratiques (2/2)

- Votre programme doit être écrit pour être testable
  - Les modules et les fonctions doivent être petites et cohésives
  - Si vous écrivez des fonctions qui sont complexes, vos tests vont être complexes
  - Vous devez toujours penser à vos tests quand vous faites le design de vos programmes, car tôt ou tard vous allez avoir à tester ce que vous avez produit

- Il y a des parties de votre code qui ne sont pas testables facilement
  - Les tests de défaillance, le traitement des erreurs et certaines conditions anormales
  - Les fuites mémoire ne peuvent pas être testées, vous devez inspecter votre code et vérifier que les blocs de mémoires sont libérés quand vous n'en avez plus de besoin
    - Sinon il faut utiliser des outils tels que *purify* ou *valgrind*

Plan du cours

- Définitions autour du test
  - Métriques de qualité/fiabilité logicielle
  - Positionnement du test dans le cycle de vie du logiciel
  - Test et méthodes agiles
  - Qu'est-ce que le test ?
- Processus de test simplifié
- Comment choisir les scénarios de test à jouer ?
  - Les méthodes de test Boîte Noire
  - Les méthodes de test Boîte Blanche
- Tests unitaires avec JUnit et EclEmma
  - Présentation de JUnit
  - Couverture de test avec EclEmma
- Utilisation de doublures (mocks) pour les tests unitaires avant intégration

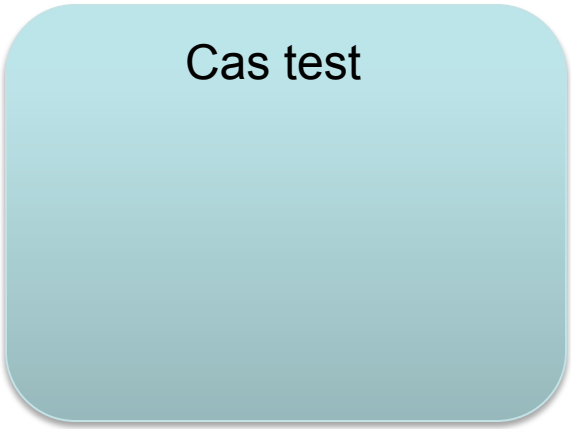
Processus simplifié de test

- **Tester :** réaliser une exécution d'un programme pour y trouver des défauts
- *Exemple*
  - *un programme pour trier un tableau d'entiers en enlevant les redondances*
  - *Interface :* `int[] mon_tri(int[] vec)`

Processus simplifié de test

- **Tester :** réaliser une exécution d'un programme pour y trouver des défauts
- *Exemple*
  - *un programme pour trier un tableau d'entiers en enlevant les redondances*
  - *Interface :* `int[] mon_tri(int[] vec)`

- **Etape 1 :** On définit un cas test (CT) à exécuter, i.e. un "scénario" que l'on veut appliquer
  - *Un tableau de N entiers non redondants*

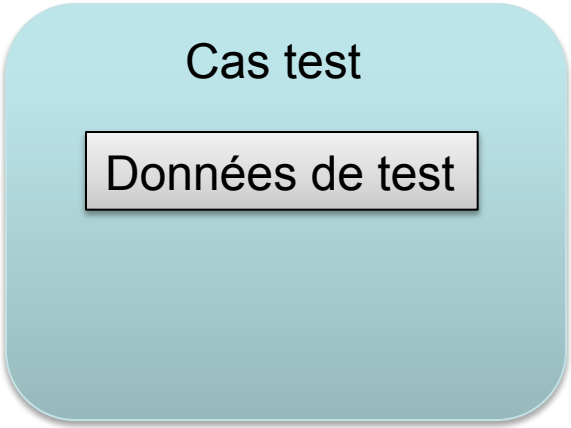


Processus simplifié de test

- **Tester :** réaliser une exécution d'un programme pour y trouver des défauts
- *Exemple*
  - *un programme pour trier un tableau d'entiers en enlevant les redondances*
  - *Interface :* `int[] mon_tri(int[] vec)`

- **Etape 2 :** On concrétise le cas test en lui fournissant des données de test (DT)
  - *Le tableau [1,0,12,4,43,5,21]*

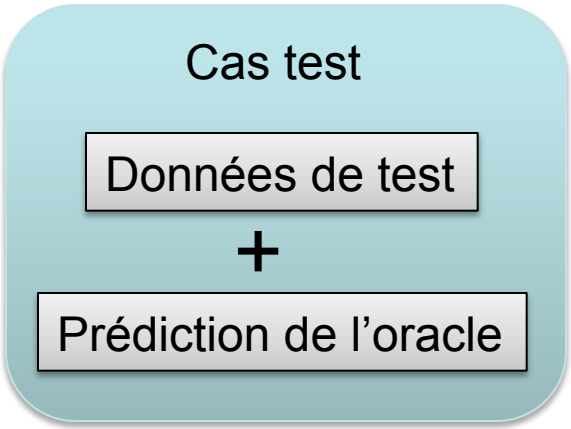
[1,0,12,4,43,5,21]



Processus simplifié de test

- **Tester :** réaliser une exécution d'un programme pour y trouver des défauts
- *Exemple*
  - *un programme pour trier un tableau d'entiers en enlevant les redondances*
  - *Interface :* `int[] mon_tri(int[] vec)`

- **Etape 3 :** On indique le résultat que l'on attend pour ce CT (prédiction de l'Oracle)
  - *[1,0,12,4,43,5,21] → [0,1,4,5,12,21,43]*



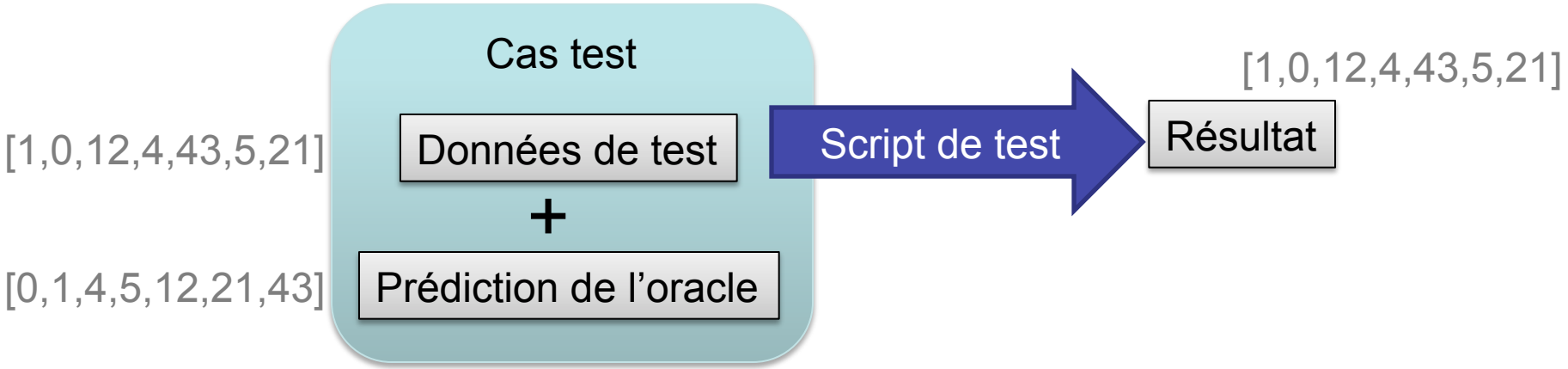
[1,0,12,4,43,5,21]

[0,1,4,5,12,21,43]

Processus simplifié de test

- **Tester :** réaliser une exécution d'un programme pour y trouver des défauts
- *Exemple*
  - *un programme pour trier un tableau d'entiers en enlevant les redondances*
  - *Interface :* `int[] mon_tri(int[] vec)`

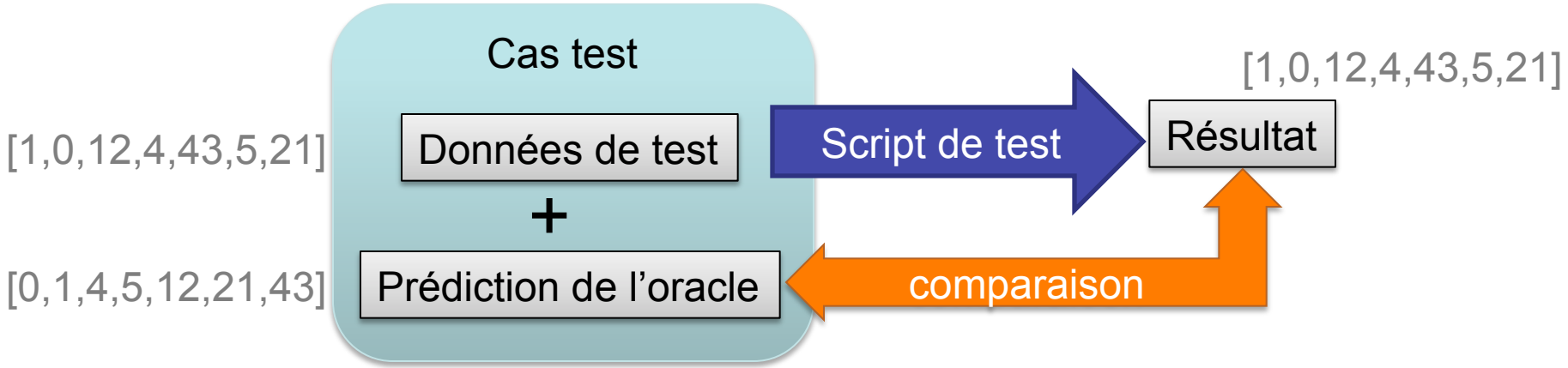
- **Etape 4 :** On exécute le script de test testant le CT sur les DT



Processus simplifié de test

- **Tester :** réaliser une exécution d'un programme pour y trouver des défauts
- *Exemple*
  - *un programme pour trier un tableau d'entiers en enlevant les redondances*
  - *Interface :* `int[] mon_tri(int[] vec)`

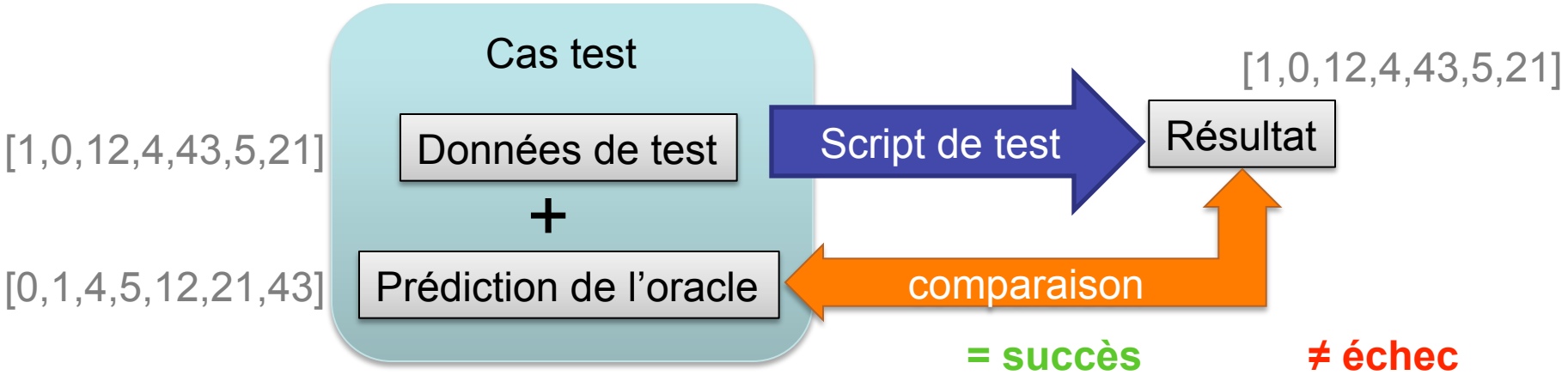
- **Etape 5 :** On compare le résultat obtenu à la prédiction de l'Oracle (verdict)



Processus simplifié de test

- **Tester :** réaliser une exécution d'un programme pour y trouver des défauts
- *Exemple*
  - *un programme pour trier un tableau d'entiers en enlevant les redondances*
  - *Interface :* `int[] mon_tri(int[] vec)`

- **Etape 6** On rapporte le résultat du test : succès / échec



Processus simplifié de test – En résumé

Processus valable pour tout type de test (unitaire, intégration, système)

Processus

1. On définit un cas test (CT) à exécuter
2. On détermine les données de test (DT) à fournir au CT (concrétisation)
3. On indique le résultat que l'on attend pour ce CT (prédiction de l'Oracle)
4. On exécute le script de test testant le CT sur les DT
5. On compare le résultat obtenu à la prédiction de l'Oracle (verdict)
6. On rapporte le résultat du test : succès / échec

Artefacts

- **Cas de tests (CT) :** "scénario" que l'on veut appliquer
- **Données de tests (DT) :** données passées en entrée à un CT
- **Script de Test :** code qui lance le programme sur le DT
- **Notion d'Oracle :** résultats attendus de l'exécution
- **Verdict :** l'exécution est-elle conforme à la prédiction de l'Oracle?
- **Suite / Jeu de tests :** ensemble de (cas de) tests

Retour à l'exemple

- Spécification : Trier un tableau d'entiers en enlevant les redondances
- Interface : `int[] mon_tri(int[] vec)`
- Quelques cas tests intuitifs à poser
  1. Un tableau de N entiers non redondants
     - L'oracle fournira un tableau trié
  2. Un tableau vide
     - L'oracle fournira un tableau vide
  3. Un tableau de N entiers dont 1 redondant
     - L'oracle fournira un tableau trié sans redondance
- Concrétisation des cas tests 1, 2 et 3 : DT et résultat attendu
  1. DT = [5,3,15,12] res = [3,5,12,15]
  2. DT = [] res = []
  3. DT = [10,20,30,5,30,0] res = [0,5,10,20,30]

Retour à l'exemple

- Mais d'autres cas test « intéressants » pour tester la robustesse de l'algorithme sous-jacent

1. DT = [2,2,2,2,2] res = [2]
2. DT = [5,4,3,2,1] res = [1,2,3,4,5]
3. DT = [1] res = [1]

Plan du cours

- Définitions autour du test
  - Métriques de qualité/fiabilité logicielle
  - Positionnement du test dans le cycle de vie du logiciel
  - Test et méthodes agiles
  - Qu'est-ce que le test ?
- Processus de test simplifié
- Comment choisir les scénarios de test à jouer ?
  - Les méthodes de test Boîte Noire
  - Les méthodes de test Boîte Blanche
- Tests unitaires avec JUnit et EclEmma
  - Présentation de JUnit
  - Couverture de test avec EclEmma
- Utilisation de doublures (mocks) pour les tests unitaires avant intégration

Sélection de tests

Deux (trois?) grandes familles de sélection de tests

1. Test fonctionnel ou boîte noire
   - test choisi à partir des spécifications (use-case, user story, etc.)
   - évaluation de l'extérieur (sans regarder le code), uniquement en fonction des entrées et des sorties
   - sur le logiciel ou un de ses composants

   (+) taille des spécifications, facilité oracle

   (-) spécifications parfois peu précises, concrétisation des DT

2. Test structurel ou boîte blanche
   - test choisi à partir du code source (portion de codes, blocs, branches)

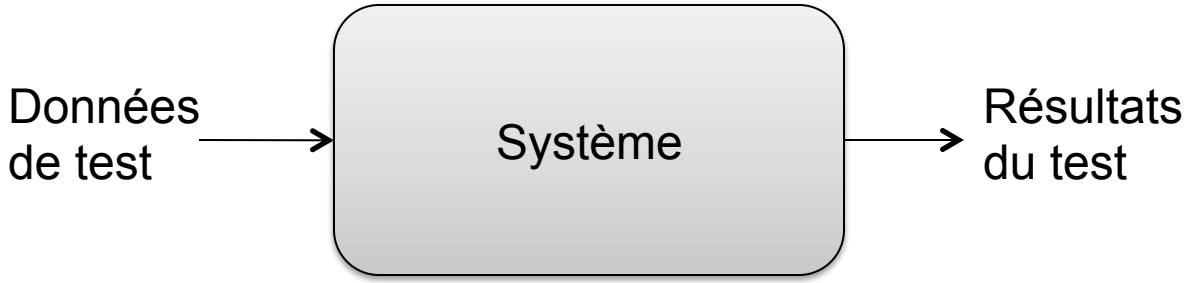
   (+) description précise, facilité script de test

   (-) oracle difficile, manque d'abstraction

3. Cas particulier du test probabiliste NON ABORDE DANS CE COURS
   - Principe de la boîte noire (juste domaine des variables)
   - mais spécificités propres (ex : oracle délicat)

Test fonctionnel – Boîte Noire

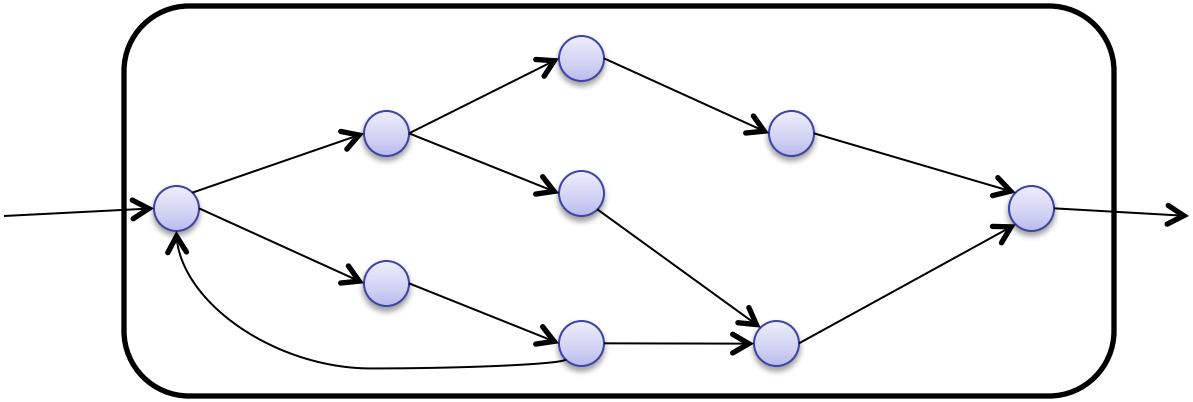
- Ne nécessite pas de connaître la structure interne du système



- Basé sur la spécification de l'interface du système et de ses fonctionnalités
- Permet d'assurer l'adéquation spécification – code, mais aveugle aux défauts fins de programmation
- Par rapport au processus de test
  - Pas trop de problème d'oracle pour le CT
  - Mais problème de la concrétisation
- Quand l'utiliser
  - test système, test d'intégration
  - test unitaire
  - leur planification et conception peut commencer tôt dans le processus de développement du logiciel, i.e., dés que les spécifications sont disponibles

Test structurel – Boîte Blanche

- La structure interne du système doit être accessible



- Les données de tests sont alors produites après analyse du code
- Se base sur le code
  - très précis, mais plus « gros » que les spécifications
- Sensible aux défauts fins de programmation, mais aveugle aux fonctionnalités absentes

- Par rapport au processus de test
  - DT potentiellement plus fines, mais très nombreuses
  - Pas de problème de concrétisation,
  - Difficile de définir l'oracle

Pourquoi faire des tests de type boîte blanche?

- On risque plus facilement de se tromper (erreur de logique, de codage) en traitant un cas peu fréquent
  - i.e., probabilité faible d'exécution et probabilité élevée d'erreur vont souvent de paire
- Certains chemins qui, intuitivement, nous semblent improbables, peuvent quand même être exécutés
- Les erreurs typographiques sont réparties un peu au hasard et il est hautement probable qu'un chemin qui n'a pas été testé en contienne

Test probabiliste

- Les données sont choisies dans leur domaine selon une loi statistique
  - loi uniforme (test aléatoire)
  - loi statistique du profil opérationnel (test statistique)

- **Test aléatoire**
  - En général, sélection aisée des DT
  - test massif si oracle (partiel) automatisé
  - « objectivité » des DT (pas de biais)
  - Mais peine à produire des comportements très particuliers (ex : x=y sur 32 bits)
- **Test statistique**
  - permet de déduire une garantie statistique sur le programme
  - Trouve les défauts les plus probables : défauts mineurs ?
  - Mais il difficile de définir la loi statistique

Sélection de tests

- Les tests boîte noire et boîte blanche sont complémentaires
  - Les approches structurelles trouvent plus facilement les défauts de programmation
  - Les approches fonctionnelles trouvent plus facilement les erreurs d'omission et de spécification
- Exemple: fonction retournant la somme de 2 entiers modulo 20 000

```
fun (x:int, y:int) : int =
      if (x==500 and y==600) then x-y      (bug 1)
                            else x+y       (bug 2)
```

  - Avec l'approche fonctionnelle (boîte noire),
    - le bug 1 est difficile à détecter, le bug 2 est facile à détecter
  - Avec l'approche structurelle (boîte blanche),
    - le bug 1 est facile à détecter, le bug 2 est difficile à détecter

Méthodes de sélection de tests

- Problèmes de la sélection de tests
  - L'efficacité du test dépend crucialement de la qualité des CT/DT
  - Ne pas manquer un comportement fautif
  - MAIS trop de CT/DT est coûteux (design, exécution, stockage, etc.)
  - L'exhaustivité est inenvisageable
- Deux enjeux
  - DT suffisamment variées pour espérer trouver des erreurs
  - Maîtriser la taille : éviter les DT redondantes ou non pertinentes

  Méthodes de sélection de tests : « recettes » pour trouver des DT pertinentes et pas trop nombreuses
- Remarques
  - Les méthodes présentées ici ne sont pas automatiques a priori certaines peuvent être +/- automatisées
  - Trade-off qualité du test, nombre de tests : uniquement des tests « pertinents »

Méthodes de sélection de tests

- Méthodes Boîte Noire
  - Combinatoire
  - partition / classes d'équivalence
  - Test aux limites
  - *graphe de cause-effet*
  - *ad hoc*
  - *couverture fonctionnelle*

- Méthodes Boîte Blanche
  - couverture structurelle
  - *Mutation*

    sélection des CT par rapport à leur effet au changement sur système

BN – Méthode Combinatoire (1/4)

- Constat : test exhaustif souvent impraticable
  - espace des entrées non borné / paramétré / infini (BD, pointeurs, espace physique, etc.)
  - Rappelez-vous les entiers 32 bits

- Approche ***pairwise*** : sélectionner les DT pour couvrir toutes les paires de valeurs
  - observation 1 : # paires beaucoup plus petit que # combinaisons

    → # DT diminue fortement par rapport au test exhaustif
  - observation 2 : une majorité de défauts sont souvent détectables par des combinaisons de 2 valeurs de variables
  - Observation 3: un seul test peut couvrir plusieurs paires
  - dans certains cas, cela peut même rendre l'énumération possible

BN – Méthode Combinatoire (2/4)

Exemple : 3 variables booléennes A, B ,C

Nombre de combinaisons de valeurs / tests : $2^3 = 8$

Nombre de paires de valeurs : 12

(A=1,B=1), (A=1,B=0), (A=1,C=1), (A=1,C=0)
(A=0,B=1), (A=0,B=0), (A=0,C=1), (A=0,C=0)
(B=1,C=1), (B=1,C=0), (B=0,C=1), (B=0,C=0)

- Le DT (A=1,B=1,C=1) couvre 3 paires

  (A=1,B=1), (A=1,C=1), (B=1,C=1)

  Ici 6 tests pour tout couvrir

- Sur de plus gros exemples avec N variables :
  - Nombre de combinaisons : 2^N
  - Nombre de paires : $2N(N-1)$

BN – Méthode Combinatoire (3/4)

Exemple 2: 4 variables a,b, c et d, pouvant chacune prendre 3 valeurs {1,2,3}

Nombre de combinaisons de valeurs / tests : $3^4 = 81$

Mais 9 cas test sont suffisants pour couvrir toutes les paires de valeurs

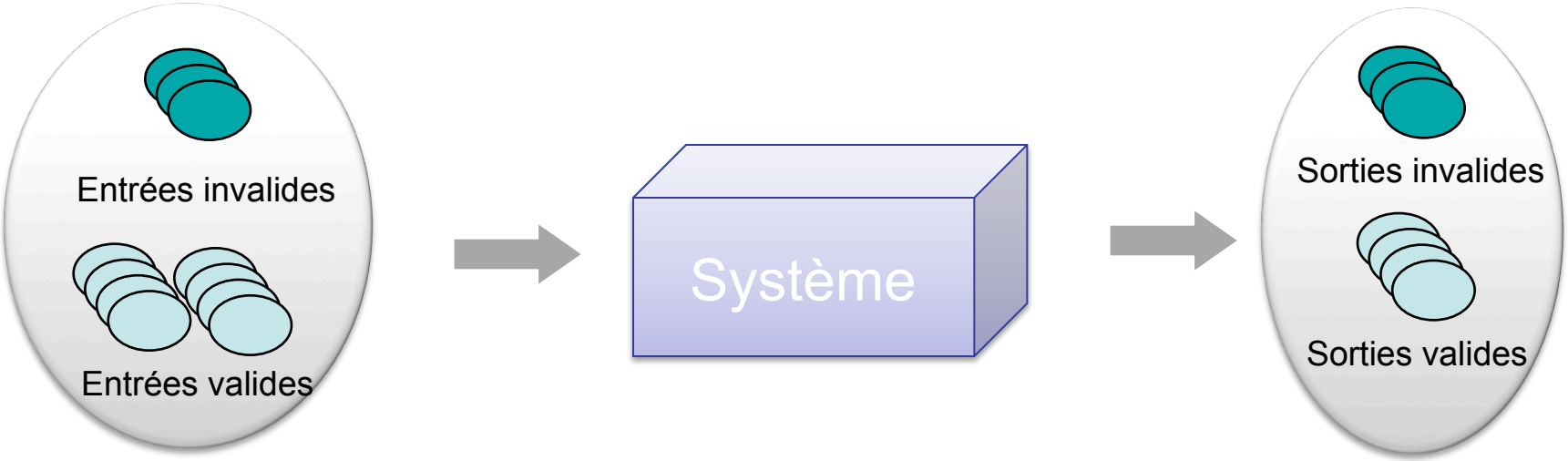
	Cas 1	Cas 2	Cas 3	Cas 4	Cas 5	Cas 6	Cas 7	Cas 8	Cas 9
a	1	2	2	1	1	3	3	2	3
b	1	2	3	2	3	1	2	1	3
c	1	2	1	3	2	2	1	3	3
d	1	1	2	2	3	2	3	3	1

BN – Méthode Combinatoire (4/4)

- Bénéfices
  - La majorité des fautes détectées par des combinaisons de 2 valeurs de variables.
  - Grande réduction si beaucoup de variables à petits domaines
- Désavantages
  - Aucune chance de détecter des bugs demandant des combinaisons de plus de 2 valeurs
  - Le résultat attendu de chaque test doit être fournis manuellement
- Remarques
  - L'approche Pairwise se décline avec des triplets, des quadruplets, …. mais le nombre de tests augmente très vite
  - Une dizaine d'outils permette de calculer les combinaisons en Pairwise(ou n-valeurs) :
    - http://www.pairwise.org/default.html
    - Prise en charge des exclusions entre valeurs des domaines et des combinaisons déjà testées

BN – Classes d'équivalence (1/3)

- Principe : diviser le domaine des entrées en un nombre fini de classes tel que le programme réagisse de la même façon pour toutes les valeurs d'une classe
  - conséquence : il ne faut tester qu'une valeur par classe !
  - évite des tests redondants (si classes bien identifiées)



- Procédure :
  1. Identifier les classes d'équivalence des entrées
     - Sur la base des conditions sur les entrées/sorties
     - En prenant des classes d'entrées valides et invalides
  2. Définir un ou quelques CTs pour chaque classe

BN – Classes d’ équivalence – Exemple 1

- Supposons que la donnée à l’entrée est un entier naturel qui doit contenir cinq chiffres, donc un nombre compris entre 10,000 et 99,999.

- Le partitionnement en classes d’équivalence identifierait alors les trois classes suivantes (trois conditions possibles pour l’entrée):
  a. N < 10000
  b. 10000 <= N <= 99999
  c. N >= 100000

- On va alors choisir des cas de test représentatif de chaque classe, par exemple, au milieu et aux frontières (cas limites) de chacune des classes:
  a. 0, 5000, 9999
  b. 10000, 50000, 99999
  c. 100000, 100001, 200000

BN – Classes d'équivalence – Exemple 2

Retour à l'exemple célèbre (G.J. Myers, *The Art of Software Testing*, 1979)

> Un programme prend 3 entiers en entrée, qui sont interprétés comme représentant les longueurs des côtés d'un triangle. Le programme doit retourner un résultat précisant s'il s'agit d'un triangle scalène, isocèle ou équilatéral.

Quels cas test pour ce programme avec des classes d'équivalence ?

- Pas un triangle

  1 côté à une longueur supérieure à la somme des 2 autres
- Isocèle

  2 côtés égaux uniquement
- Équilatéral

  3 côtés égaux
- Scalène

  Les autres

BN – Classes d’ équivalence – Exemple 3

- Soit un programme calculant la valeur absolue d’un entier relatif saisi sous forme d’une chaîne de caractères au clavier, *quelles classes d’équivalences pour le tester ?*

BN – Classes d'équivalence – Exemple 3

- Soit un programme calculant la valeur absolue d'un entier relatif saisi sous forme d'une chaîne de caractères au clavier, *quelles classes d'équivalences pour le tester ?*

- **Classe 1**
  Entrée invalide, chaîne vide
- **Classe 2**
  Entrée invalide, chaîne avec plusieurs mots
  « 123 983 321 »
- **Classe 3**
  Entrée invalide, un seul mot mais pas un entier relatif
  « 123.az+23 »
- **Classe 4**
  Entrée valide, un mot représentant un entier relatif positif ou nul
  « 673.24»
- **Classe 5**
  Entrée valide, un mot représentant un entier relatif négatif
  « -2.4 »

BN – Classes d'équivalence (3/3)

Règles de sélection des classes d'équivalence

- Si la valeur appartient à un intervalle, construire :
  - une classe pour les valeurs inférieures,
  - une classe pour les valeurs supérieures,
  - N classes valides.
- Si la donnée est un ensemble de valeurs, construire :
  - une classe avec l'ensemble vide,
  - une classe avec trop de valeurs,
  - N classes valides.
- Si la donnée est une obligation ou une contrainte (forme, sens, syntaxe), construire :
  - une classe avec la contrainte respectée,
  - une classe avec la contrainte non respectée

BN – Test aux limites (1/4)

- Le test des valeurs limites n'est pas vraiment une famille de sélection de test, mais une tactique pour améliorer l'efficacité des DT produites par d'autres familles.
  - Il s'intègre très naturellement au test par classe d'équivalence
  - On peut s'en servir dans d'autres techniques de sélection (exemple: graphes cause-effet)

- Idée : les erreurs se nichent dans les cas limites, donc tester principalement les valeurs aux limites des domaines ou des classes d'équivalence.
  - test partitionnel en plus agressif
  - plus de DT donc plus cher
  - Exemples classiques: accéder à la n+1ème case d'un tableau de n cases, erreur de +/- 1 dans des boucles

- Stratégie de test :
  - Tester les bornes des classes d'équivalence, et juste à côté des bornes
  - Tester les bornes des entrées et des sorties

BN – Test aux limites (2/4)

Principe : on s'intéresse **aux bornes des intervalles** partitionnant les domaines des variables d'entrées

- Pour chaque intervalle, on garde les 2 valeurs correspondant aux 2 limites, et les 4 valeurs correspondant aux valeurs des limites ± le plus petit delta possible

  n ∈3 .. 15 ⇒v1 = 3, v2 = 15, v3 = 2, v4 = 4, v5 = 14, v6 = 16

- Si la variable appartient à un ensemble ordonné de valeurs, on choisit le premier, le second, l'avant dernier et le dernier

  n ∈ {-7, 2, 3, 157, 200} ⇒ v1 = -7, v2 = 2, v3 = 157, v4 = 200

- Si une condition d'entrée spécifie un nombre de valeurs, définir les cas de test à partir du nombre minimum et maximum de valeurs, et des test pour des nombres de valeurs hors limites invalides

  Un fichier d'entrée contient entre 1 et 100 enregistrements, produire un cas de test pour 0, 1, 100 et 101

BN – Test aux limites (3/4)

Autres valeurs non numériques

- Chaînes de caractères (avec au plus n caractères)
  - chaîne vide, chaîne avec un seul caractère, chaîne avec n-1, n, n+1 (?) caractères
- Tableaux
  - tableau vide, avec un seul élément, avec n-1 ou n éléments
- Fichiers
  - fichier vide, avec une seule ligne, fichier de taille maximale, fichier trop gros.
- Test sur des ensembles
  - ensemble vide, ensemble trop gros, ensemble valide

BN – Test aux limites (4/4)

- Méthode de test fonctionnel très productive :
  - le comportement du programme aux valeurs limites est souvent pas ou insuffisamment examiné

- Couvre l'ensemble des phases de test
  - unitaires, d'intégration, de conformité et de régression

- Inconvénient : caractère parfois intuitif ou subjectif de la notion de limite

⇒ Difficulté pour caractériser la couverture de test.

- Remarque: les tests aux limites sont aussi applicables pour les tests de type boîte blanche

BN – Test aux limites et classes d'équivalence

- **L'analyse partitionnelle** est une méthode qui vise à diminuer le nombre de cas de tests par calcul de classes d'équivalence
  - importance du choix des classes d'équivalence : on risque de ne pas révéler un défaut

- Le choix des conditions d'entrée aux limites est une heuristique solide de choix de données d'entrée au sein des **classes d'équivalence**
  - cette heuristique n'est utilisable qu'en présence de relation d'ordre sur la donnée d'entrée considérée.
  - Le test aux limites produit à la fois des cas de test nominaux (dans l'intervalle) et de robustesse (hors intervalle)

BN – Classes d’équivalence et multi-variables (1/2)

Question : si plusieurs variables, comment définir les classes d’´equivalence ?

- Approche 1 (naive) : produit cartésien
  - partitions sur chaque variable puis produit cartésien
  - # P = ∏ #Pi
  - on peut retrouver le problème de la combinatoire
  - risque 1 = explosion du nombre de partitions
  - risque 2 = manquer des DT avec relations spécifiques (ex : x ≤ y)

- Approche 2 : union des partitions
  - partitions sur chaque variable, puis union des partitions
  - un test passe par plusieurs partitions (1 par variable)
  - # P = ∑ #Pi
  - risque 1 = on test chaque Pi dans un seul contexte
  - risque 2 = manquer des DT avec relations spécifiques (ex : x ≤ y)

BN – Classes d'équivalence et multi-variables (2/2)

- Approche 3 : partitionner le domaine total
  - partition définies directement sur le produit cartésien des variables
  - Résout le risque 2
  - #P souvent plus petit que ∏ #Pi
  - #P peut être plus grand que ∑ #Pi
  - Risque = classes plus difficiles à définir
- Approche 4 : combiner partitions + pairwise
  - partitions comme pour l'approche 1
  - on essaie de couvrir les paires de classes d'équivalences
  - risque = manquer certaines combinaisons de partitions
  - risque 2 = manquer des DT avec relations spécifiques (ex : $x \leq y$)
- Il n'existe pas forcément de meilleure solution, cela dépend du problème
  - avoir ces solutions en tête, et s'adapter / innover
  - Dans tous les cas, ne pas mixer les cas d'erreurs entre eux

Méthodes de sélection de tests

- Méthodes Boîte Noire
  - Combinatoire
  - partition / classes d'équivalence
  - Test aux limites
  - *graphe de cause-effet*
  - *ad hoc*
  - *couverture fonctionnelle*

- Méthodes Boîte Blanche
  - couverture structurelle
  - *Mutation*

    sélection des CT par rapport à leur effet au changement sur système

BB – Qu'est-ce que la couverture structurelle ?

- Cas de tests générés en partant du code, en inspectant les chemins d'exécution du système
- Comme le nombre de chemins d'exécution peut être infini, on définit des critères de couverture afin d'augmenter la probabilité (informel) de trouver des erreurs avec des chemins pas trop longs et en petit nombre
- Critère de couverture = quels sont les chemins d'exécution à tester afin de couvrir le plus de comportements du système
- Approprié pour le test unitaire ou d'intégration, mais il passe mal à l'échelle
- Les comportements sont définis sur le graphe de flot du programme
  - **Flot de contrôle (CFG)**
    → toutes les instructions, toutes les branches, tous les chemins
  - **Flot de données (DFG)**
    → toutes les définitions de variable, toutes les utilisations

BB – Graphe de flot de contrôle

- Un graphe de flot de contrôle est une représentation abstraite et simplifiée de la structure du programme
  - Décrit le flot de contrôle du programme
  - Chaque nœud représente un segment de code strictement séquentiel (sans choix ou boucle)
  - Chaque arc dénote un lien de contrôle (possibilité d'aller d'un nœud à l'autre)

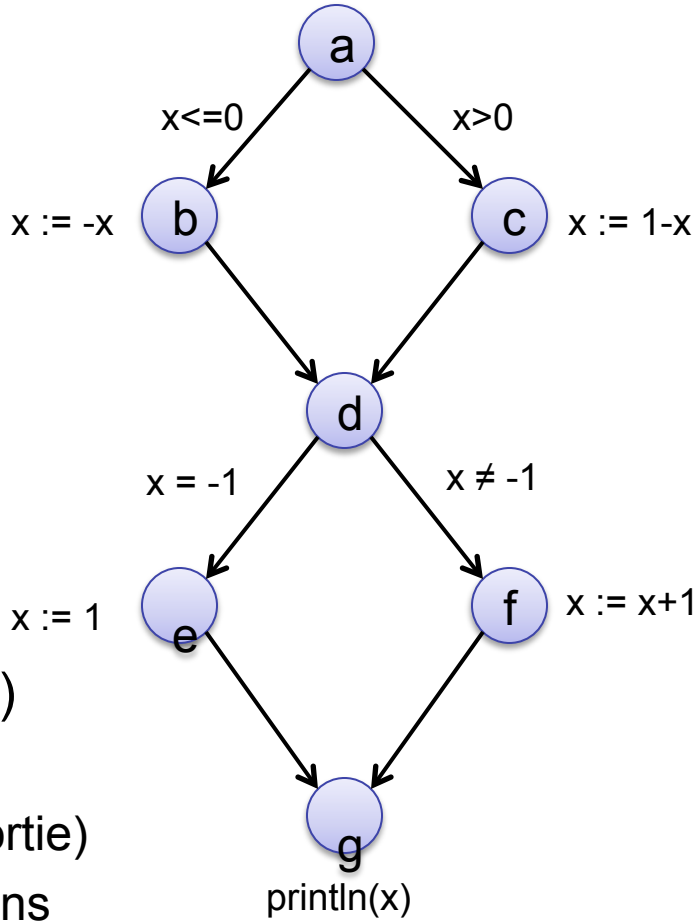
- Utile pour calculer la complexité cyclomatique et déterminer le nombre de cas de tests à définir pour assurer une couverture acceptable
  - La complexité cyclomatique est une mesure de la complexité logique d'un programme
  - Dans le contexte des tests, la complexité cyclomatique indique le nombre maximum des chemins complets indépendants, ce qui donne le nombre de cas de tests nécessaires pour assurer une bonne couverture du code (niveaux N1 et N2, voir plus loin).

BB – Exemple de Graphe de flot de contrôle

- Soit le programme P1 suivant :

```
if x <= 0 then x := -x
else x := 1 -x;
if x = -1 then x=1
else x := x+1;
println(x)
```

Ce programme admet le graphe de flot de contrôle

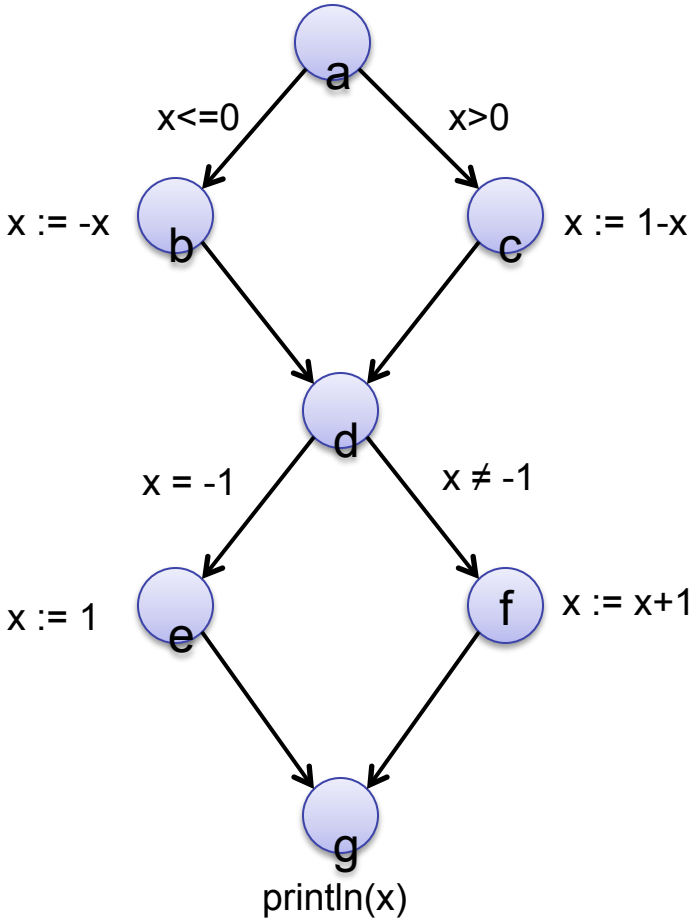


- Graphe orienté et connexe (N,A,s,P)
  - S : un sommet source (entrée)
  - P : un ou plusieurs sommet puit (sortie)
  - Un sommet est un bloc d'instructions
  - Un arc est la possibilité de transfert de l'exécution d'un bloc vers un autre

BB – Exemple de graphe de flot de contrôle

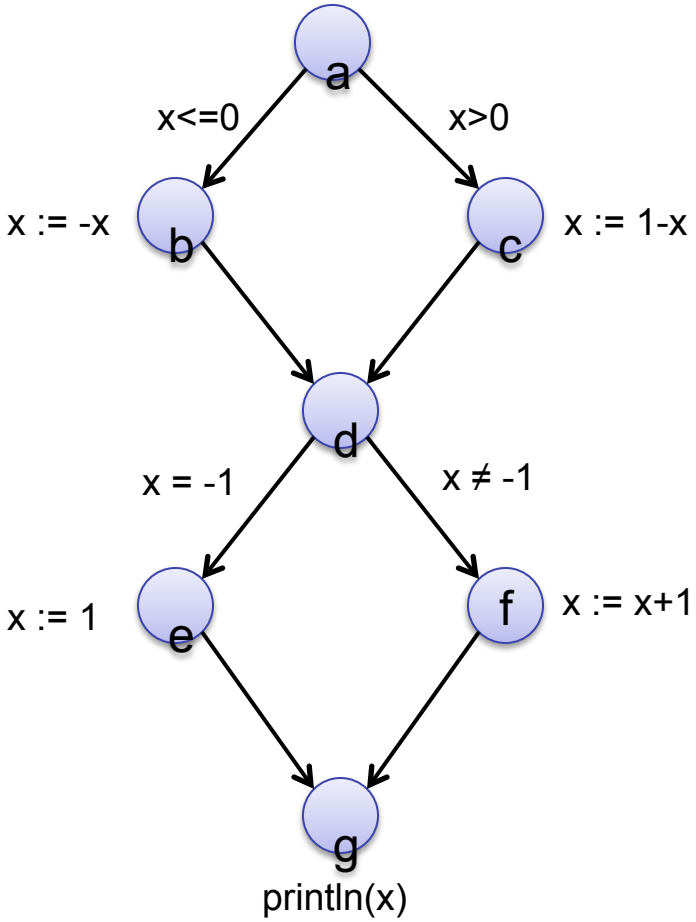
- Ce graphe est un graphe de contrôle qui admet une entrée (le nœud a), une sortie (le nœud g).
  - le chemin *[a, c, d, e, g] est un chemin de contrôle,*
  - le chemin *[b, d, f, g] n'est pas un chemin de contrôle.*

- Le graphe comprend 4 chemins de contrôle :
  - β1 = *[a, b, d, f, g]*
  - β2 = *[a, b, d, e, g]*
  - β3 = *[a, c, d, f, g]*
  - β4 = *[a, c, d, e, g]*



BB – Exemple de graphe de flot de contrôle

- Le graphe comprend 4 chemins de contrôle :
  - β1 = *[a, b, d, f, g]*
  - β2 = *[a, b, d, e, g]*
  - β3 = *[a, c, d, f, g]*
  - β4 = *[a, c, d, e, g]*



- Forme algébrique

abdfg + abdeg + acdfg + acdeg

=

a(bdf + bde + cdf + cde)g

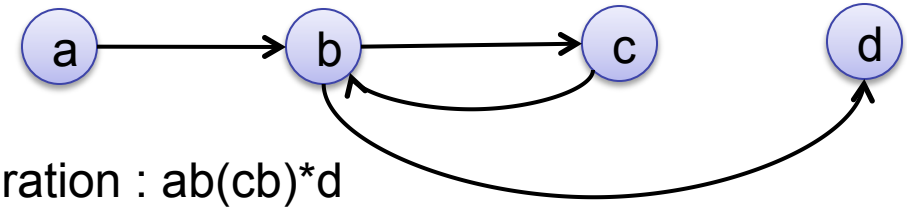
=

a(b + c)d(e + f)g

BB – Calcul de l'expression des chemins de contrôle

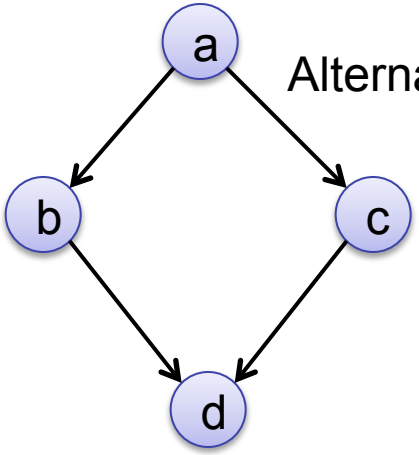
- Une opération d'addition ou de multiplication est associée à toutes les structures primitives apparaissant dans le graphe de flot de contrôle

Séquence : ab



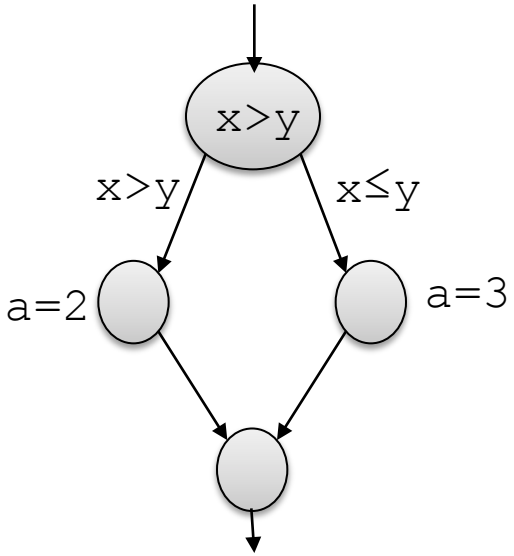
Itération : ab(cb)*d

Alternative : a(b+c)d

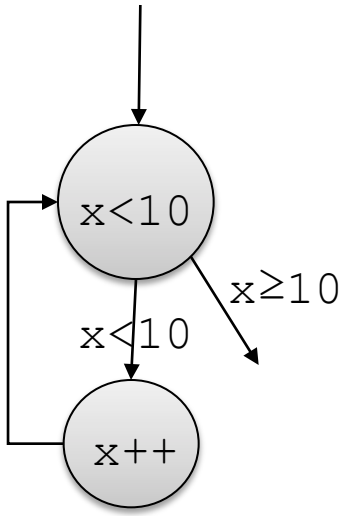


- Soient deux nœuds a et b
  - ε représente l'absence de nœud
  - $(ab)^2$ = abab exactement 2 fois
  - $(ab)^{\underline{2}}$ = ε+ab+abab au plus 2 fois
  - a^* = $\varepsilon+a+a^2+a^3+a^4+\ldots$
  - a^+ = $a+a^2+a^3+a^4+\ldots$

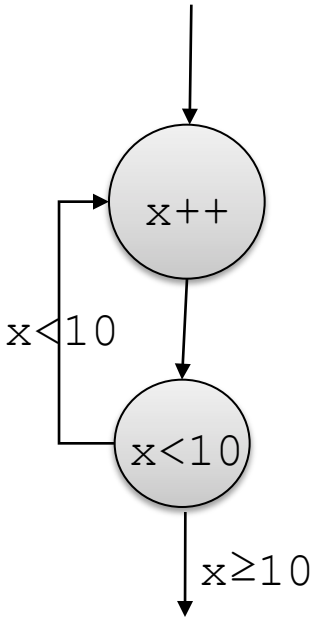
BB – Les graphes de flot pour diverses structures de contrôle



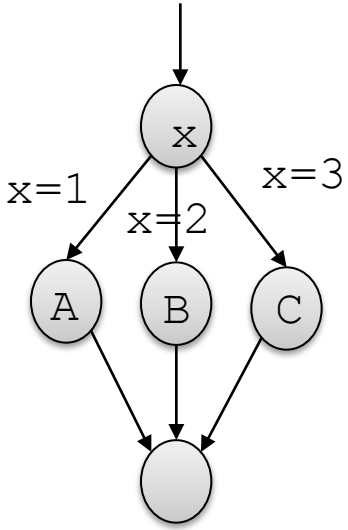

```
if (x>y)
   a = 2;
else
   a = 3;
```




```
while(x<10)
      x++;
```



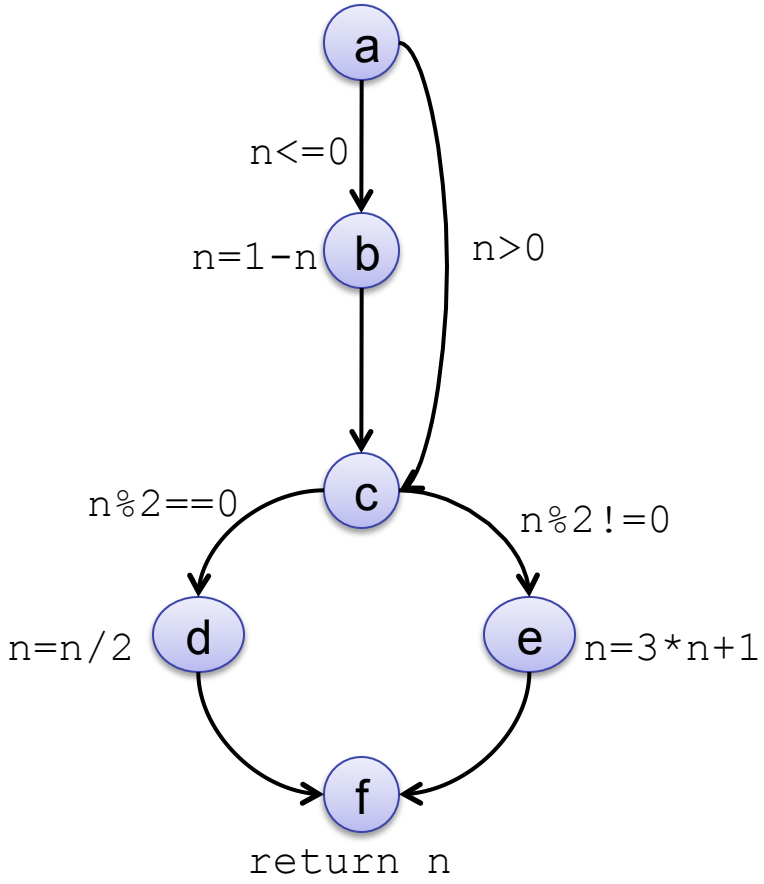

```
do
   x++;
while (x<10)
```




```
switch(x) {
case 1:
      A;break;
case 2:
      B;break;
case 3:
      C;break;
}
```

BB – Exercice 1

```
int f(int n){
 if (n<=0)
   n = 1-n;
 if (n%2==0)
   n = n/2;
 else
   n = 3*n+1;
 return n;
}
```

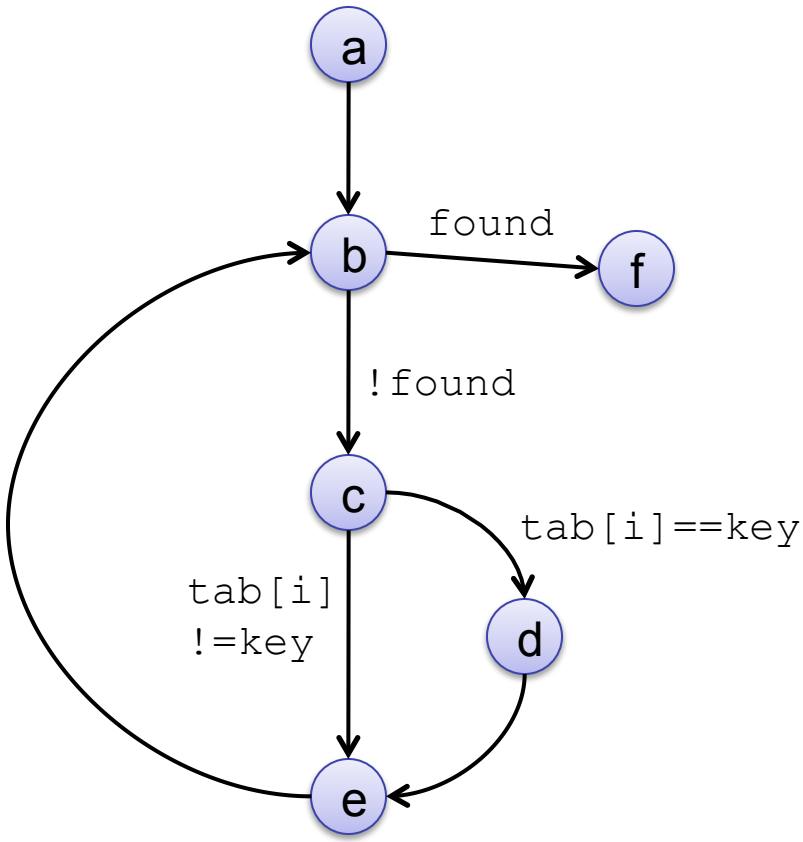


1. Etablir le graphe de flot de contrôle de ce programme
2. Fournir l'expression des chemins

a (1 + b) c (e + d) f

BB – Exercice 2

```
int f(int* tab, int key){
  int i = 0;                    a
  int res;
  bool found = false;
  while(!found){
    if(tab[i]==key){
      found=true;               d
      res=i;
    }
    i = i+1;                    e
  }
  return res;                   f
}
```



1. Etablir le graphe de flot de contrôle de ce programme
2. Fournir l'expression des chemins

ab (c (1 + d) eb)* f

BB – Couverture du CFG

Pour trouver quels chemins parcourir pour couvrir le plus de comportements du système, voici quelques critères de couverture sur un flot de contrôle

- Tous les nœuds (I) : le plus faible
  - Couverture des instructions**, niveau 1**
- Tous les arcs / décisions (D) : test de chaque décision
  - Couverture des branchements et conditions, **niveau 2**
- Tous les chemins : le plus fort, impossible à réaliser s'il y a des boucles
  - Couverture exhaustive, **niveau 0**
  - De plus, attention, certains chemins peuvent ne pas être atteignables
- On peut baser des méthodes de couverture sur ces critères ou en utiliser de plus avancés

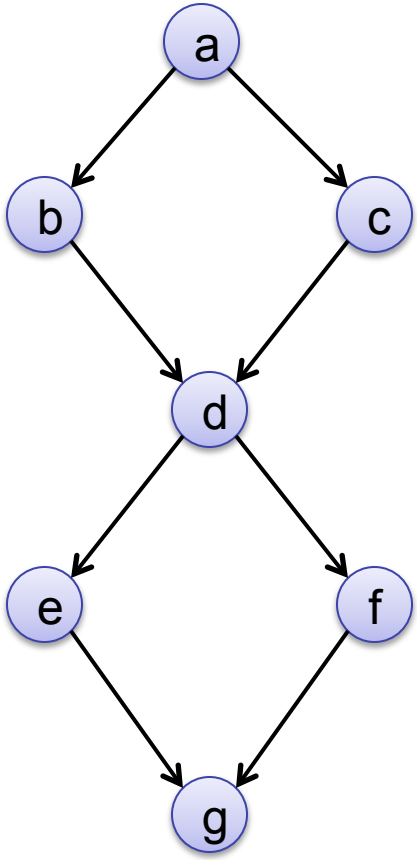
BB – Couverture du CFG

Comment effectuer la couverture du CFG ?

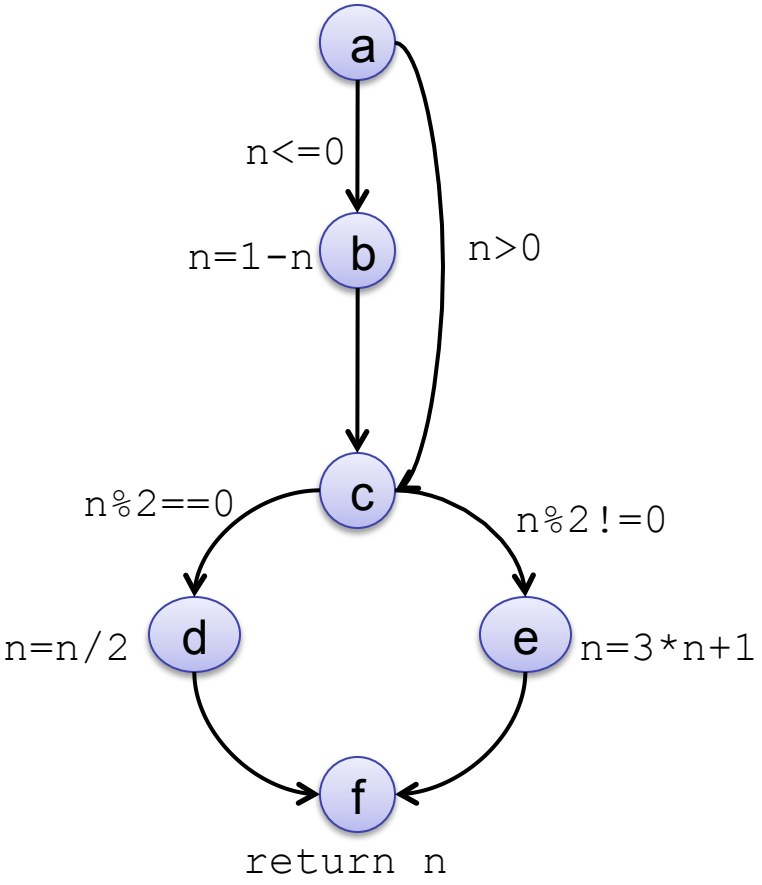
- Définir une donnée de test (DT) permettant de **sensibiliser** les chemins du CFG
  - Si un chemin est sensibilisé par une DT, il est dît **exécutable**
  - Si aucune DT ne permet de sensibiliser un chemin, il est dît **non exécutable**

Combien existe t'il de chemins pour une CFG ?

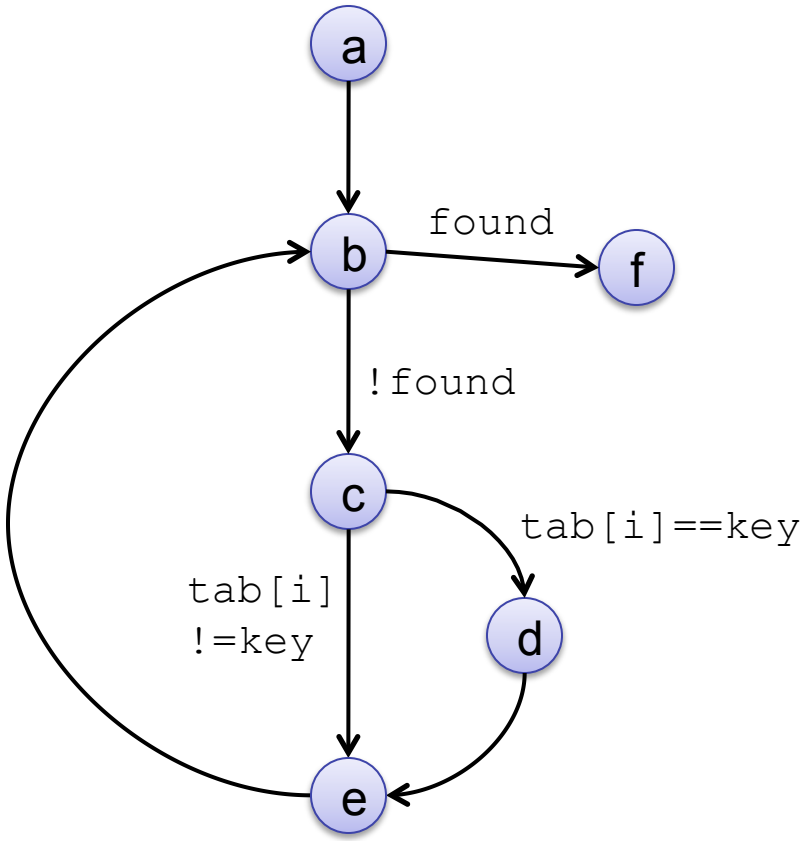
- Se déduit directement de l'expression algébrique du CFG
  - a(b + c)d(e + f)g → 1.(1+1).1.(1+1).1=4
  - Fournit le nombre de chemins exécutables et non exécutables



BB – Couverture du CFG



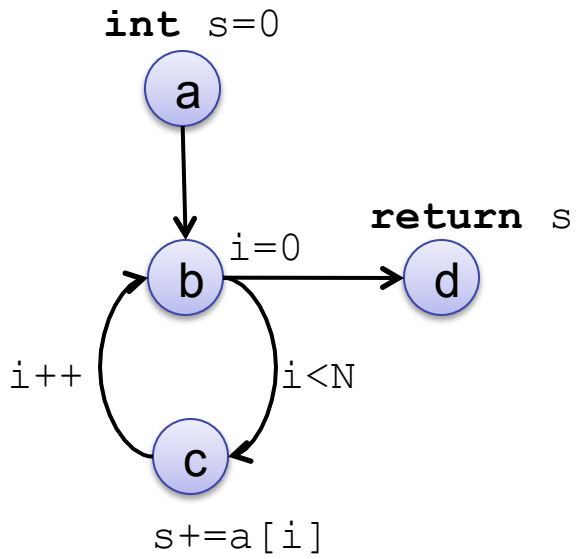
a (1 + b) c (e + d) f
soit 1.(1+1).1.(1+1).1 =4



ab (c (1 + d) eb)* f
soit l'infini

BB – Couverture du CFG

```
int somme(int* a, int N){
 int s = 0;

 for (int i=0;i<N;i++)
   s+=a[i];

 return s;
}
```



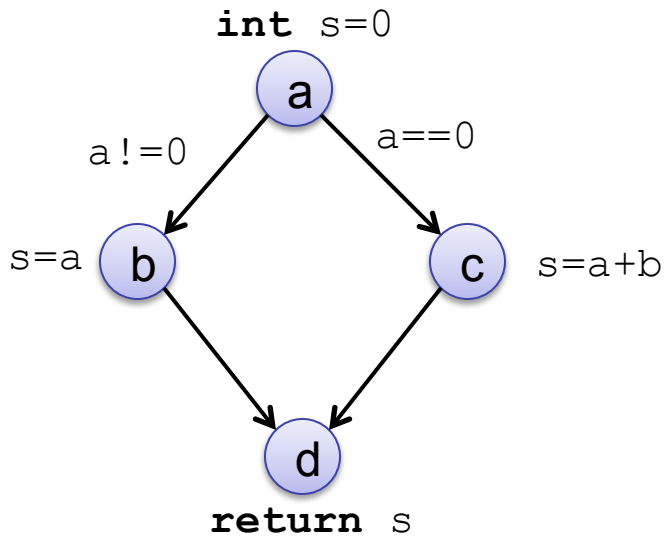
Expression des chemins :

$ab(cb)^N d$ → $1.1.(1.1)^N.1 = 1^N = N$

Niveau 1 – couverture de tous les nœuds

```
int somme(int a, int b){
 int s = 0;
 if (a==0)
   s = a;
 else
   s = a+b;
 return s;
}
```

Taux de couverture =
nb de nœuds couverts/nb total de nœuds



2 chemins de contrôle pour couvrir tous les nœuds

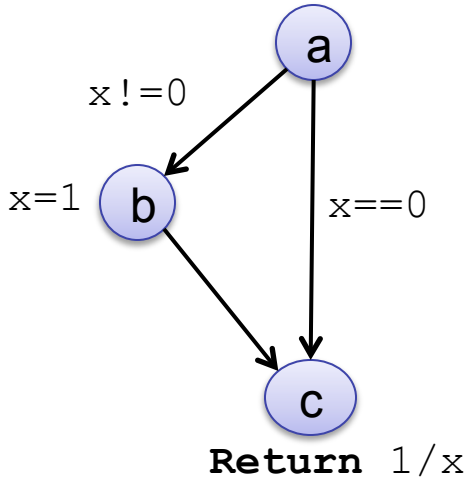
β_1 = abc

β_2 = acd

→ Détection de l'erreur avec le chemin β_1

Niveau 1 – couverture de tous les nœuds

```
int f(int x){
 if (x!=0)
   x = 1;
 return 1/x;
}
```



Le chemin de contrôle ***abc*** couvre tous les nœuds

→Sensibilisé par la DT = {x=2}

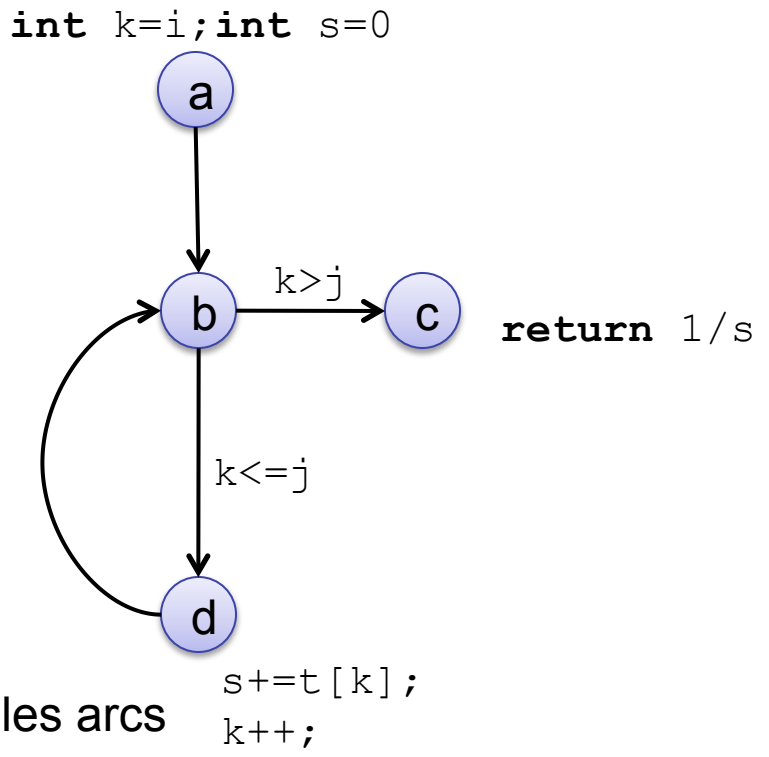
→ Mais pas de détection de l'erreur (division par 0)

Niveau 2 – couverture de tous les arcs

Calcul de l'inverse de la somme des éléments d'un tableau entre les indices **i** et **j**

Taux de couverture =
nb d'arcs couverts/nb total d'arcs

```
int somme(int* t, int i, int j)
{
   int k = i;
   int s=0;
   while(k<=j){
        s+=t[k];
        k++;
   }
   return 1/s;
}
```



La DT = {t = {1,2,3},i=0;j=2} permet de couvrir tous les arcs

Or, c'est la DT = {i=1,j=0} qui met le programme en erreur

BB – Couverture du CFG

- Remarque sur les couvertures de niveau 0, 1 et 2
  - N0 est impossible en présence de boucles : on ne peut pas tester tous les chemins distincts dans le cas d'une boucle avec un grand nombre (possiblement variable) d'itérations (chaque façon de terminer la boucle est un chemin distinct) ☹
    - *On couvrira plutôt tous les chemins indépendants ou toutes les portions linéaires de code suivies d'un saut*
  - N1 n'est pas suffisant : si une instruction `if` ne possède pas de branche `else`, on peut avoir exécuté toutes les instructions, mais sans avoir testé ce qui arrive lorsque la condition est fausse ☹
  - Si le programme est bien structuré (pas de `goto`), alors N2 implique N1 ☺

BB – CFG – Boucles

Attention au traitement des boucles

- Une seule boucle (simple), tester les exécutions :
  - aucune itération
  - une seule itération
  - deux itérations
  - un nombre typique d'itérations
  - n − 1, n, n + 1 itérations avec n le nombre d'itérations.
- Boucles imbriquées : commencer par fixer le nombre d'itérations de la boucle la plus extérieure et tester les boucles intérieures comme boucle simple.
- Boucles en suite : si dépendantes, tester comme imbriquées, sinon tester comme simple.

BB – Critère des chemins indépendants

- Définitions
  - Chemin : ensemble d'instructions de l'entrée à la sortie
  - Chemin indépendant : chemin introduisant un ensemble nouveau d'instructions ou le traitement d'une nouvelle condition (dans le cadre d'une suite de chemins que l'on teste)

- Principe du critère des chemins indépendants
  - Chaque chemin indépendant doit être parcouru au moins une fois

- Le critère *tous les chemins indépendants* vise à parcourir tous les arcs dans chaque configuration possible (et non pas au moins une fois comme dans le critère *tous les arcs*)

- Lorsque le critère *tous les chemins indépendants* est satisfait, cela implique :
  - le critère *tous les arcs* est satisfait
  - le critère *tous les nœuds* est satisfait

- Nombre minimal de cas de tests
  - Le nombre de chemins indépendants pour un graphe G est V(G) c'est-à-dire *Nb arcs – Nb nœuds + 2* ou *Nb conditions + 1* (If, While, for, case, and, …)

BB – Critère des chemins indépendants

- Procédure

1. Evaluer le nombre minimal de cas tests à générer

2. Produire une DT couvrant le maximum de nœuds de décisions du graphe

3. Puisque qu'un chemin indépendant du précédent introduit un ensemble nouveau d'instructions ou le traitement d'une nouvelle condition, produire la DT qui modifie la valeur de vérité de la première instruction de décision de la dernière DT calculée.

- Recommencer l'étape 3 jusqu'a la couverture de toutes les décisions.

BB – Critère des chemins indépendants

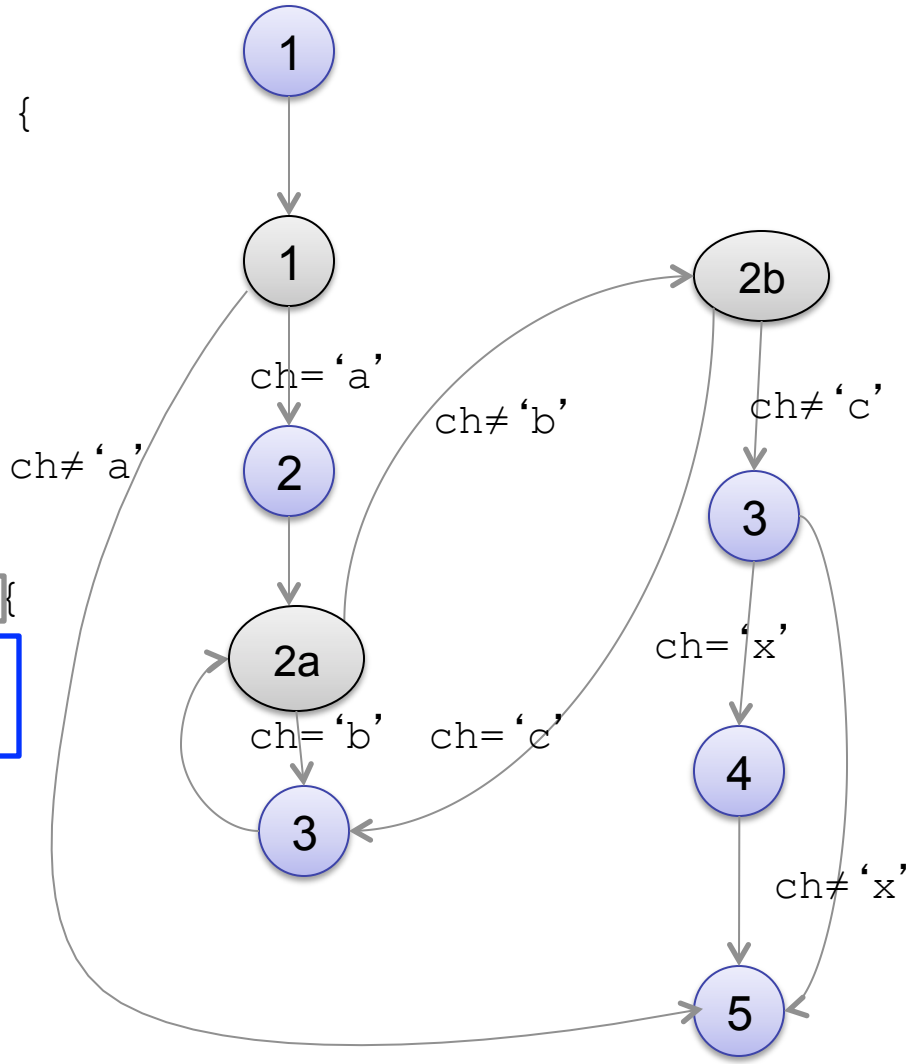
Exemple

```
bool goodstring(int count){
char ch;
bool good = false;
count := 0;
read(ch);
if ch = ‘a’ then {
   read(ch);
   while(ch==‘b’||ch==‘c’){
        count := count + 1;
        read(ch);
   }
   if ch = ‘x’
        good = true;
}
return good;
```

BB – Critère des chemins indépendants

Exemple

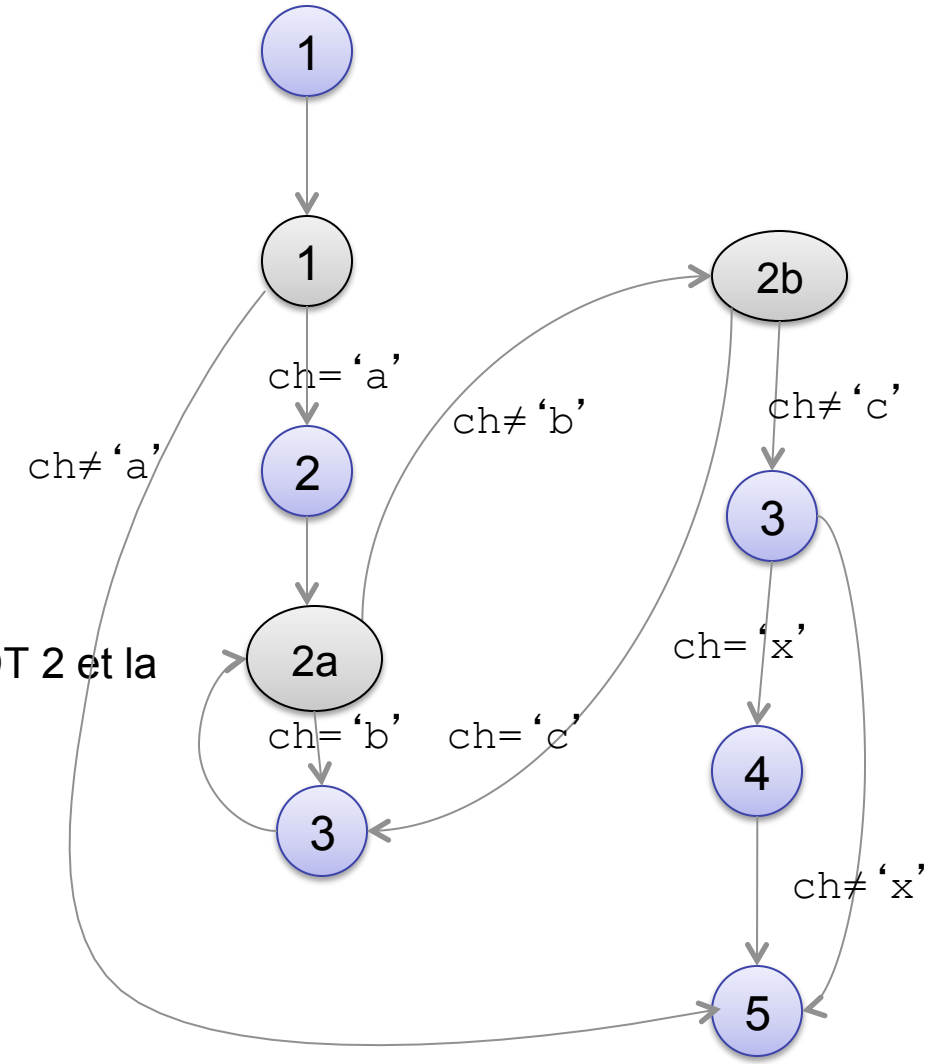
```
bool goodstring(int count){
1     char ch;
      bool good = false;
      count := 0;
      read(ch);
  1   if ch = 'a' then {
2         read(ch);
 2 a-b  while(ch=='b'||ch=='c'){
3           count := count + 1;
            read(ch);
        }
  3     if ch = 'x'
4           good = true;
      }
5   return good;
```



BB – Critère des chemins indépendants

Exemple

- Nombre minimal de DT
  - 12 – 9 + 2 = 5
- DT 1 = « abcx »
  - Couvre tous les nœuds
- DT 2 = « b »
  - Modifie la 1ère instruction de DT 1
- DT 3 = « acx »
  - Modifie la 1ère instruction de DT 2 et la 2ème de DT 1
- DT 4 = « ax »
- DT 5 = « aba »



BB – Critère des chemins indépendants, à vous de jouer!

Extrait de R.S. Pressman (2001) *Software Engineering: A Practitioner's Approach*, 5th ed. McGraw-Hill.

Entrée:
-1 tableau d'au plus 100 nombres
-1 borne min
-1 borne max

Retour:
-Moyenne des nombres compris entre min et max,
- Somme de tous les nombres
-Somme des nombres compris entre min et max

```
PROCEDURE average:

* This procedure computes the average of 100 or fewer
  numbers that lie between bounding values; it also computes the
  sum and the total number valid.

INTERFACE RETURNS average, total.input, total.valid;
INTERFACE ACCEPTS value, minimum, maximum;

TYPE value[1:100] IS SCALAR ARRAY;
TYPE average, total.input, total.valid;
   minimum, maximum, sum IS SCALAR;
TYPE i IS INTEGER;
1  i = 1;
   total.input = total.valid = 0;
   sum = 0;
2  DO WHILE value[i] <> -999 AND total.input < 100
3     increment total.input by 1;
4     IF value[i] > = minimum AND value[i] < = maximum
5        THEN increment total.valid by 1;
6              sum = sum + value[i]
7        ELSE skip
8     ENDIF
      increment i by 1;
9  ENDDO
   IF total.valid > 0
10    THEN average = sum / total.valid;
11    ELSE average = -999;
12 ENDIF
13 END average
```

BB – Critère des chemins indépendants, à vous de jouer!

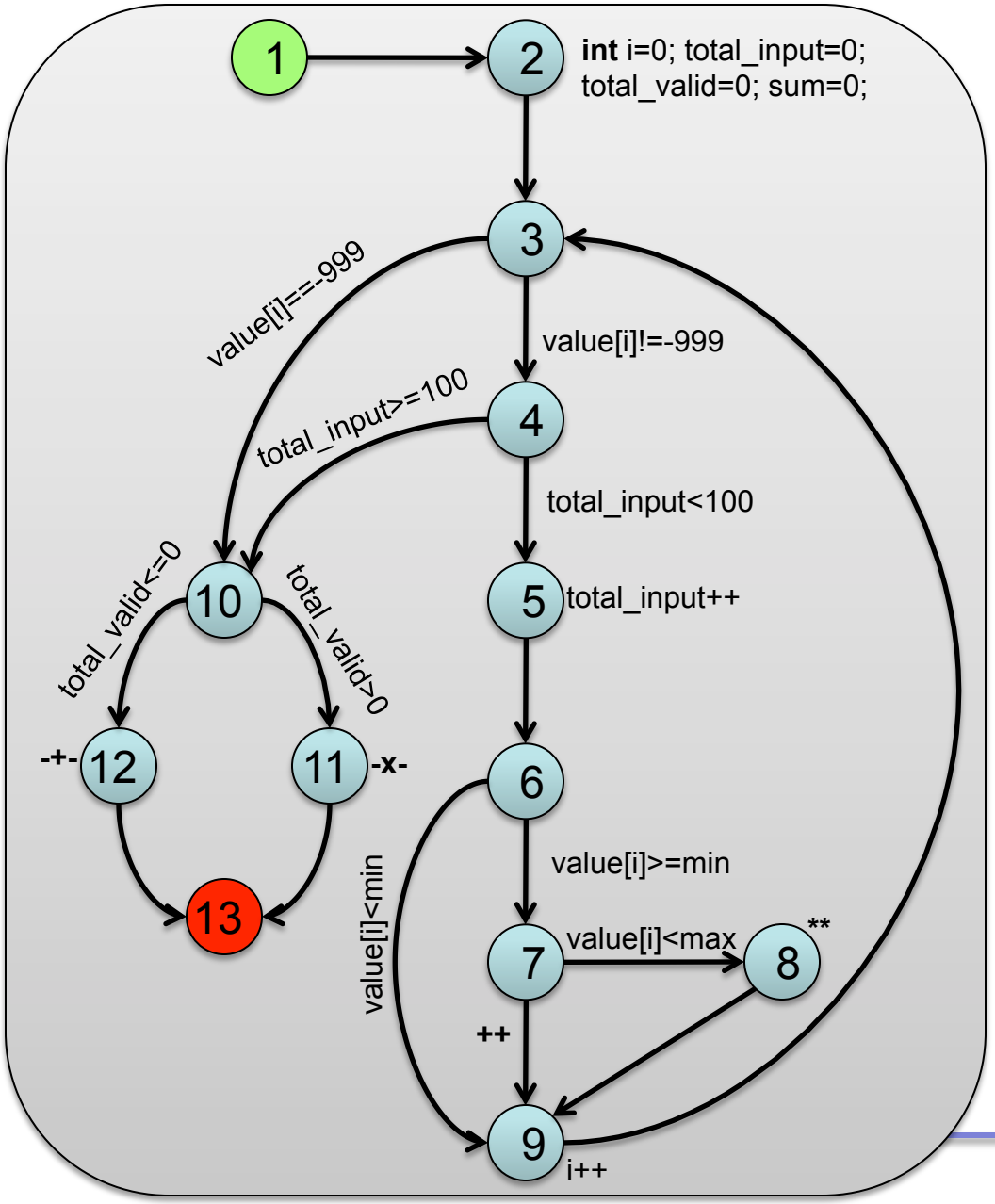
Extrait de R.S. Pressman (2001) *Software Engineering: A Practitioner's Approach*, 5th ed. McGraw-Hill.

Entrée:
-1 tableau d'au plus 100 nombres
-1 borne min
-1 borne max

Retour:
-Moyenne des nombres compris entre min et max,
- Somme de tous les nombres
-Somme des nombres compris entre min et max

```
void f( int value[], int total_input, int total_valid,
        int sum, double average)
int i=0;
total_input=0;
total_valid=0;
sum=0;
while(value[i]!=-999 && total_input<100)
{
   total_input++;
   if(value[i]>=min && value[i]<=max)
   {
      total_valid++;
      sum+=value[i];
   }
   i++;
}
if (total_valid>0)
   average=sum/total_valid;
else
   average = -999;
```

BB – Critère des chemins indépendants, à vous de jouer!




```
void f( int value[], int total_input, int total_valid,
        int sum, double average)
int i=0;
total_input=0;
total_valid=0;
sum=0;
while(value[i]!=-999 && total_input<100)
{
   total_input++;
   if(value[i]>=min && value[i]<=max)
   {
      total_valid++;
      sum+=value[i];
   }
   i++;
}
if (total_valid>0)
   average=sum/total_valid;
else
   average = -999;
```

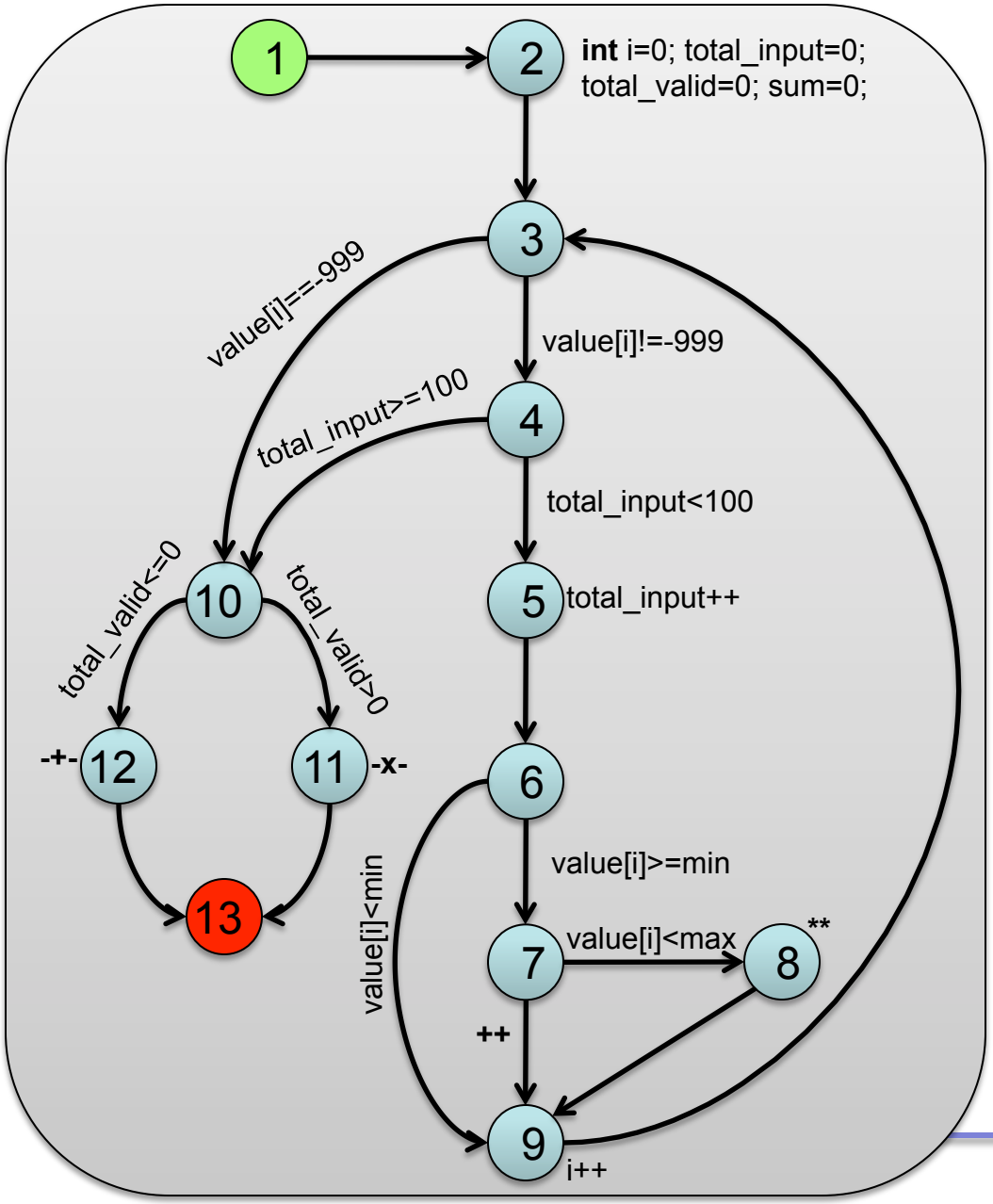
++ value[i]<max

** total_valid++; min+=value[i]

-+- average!=-999

-x- average!=sum/total_valid

BB – Critère des chemins indépendants, à vous de jouer!



Nombre cyclomatique : 17-13+2 = 6

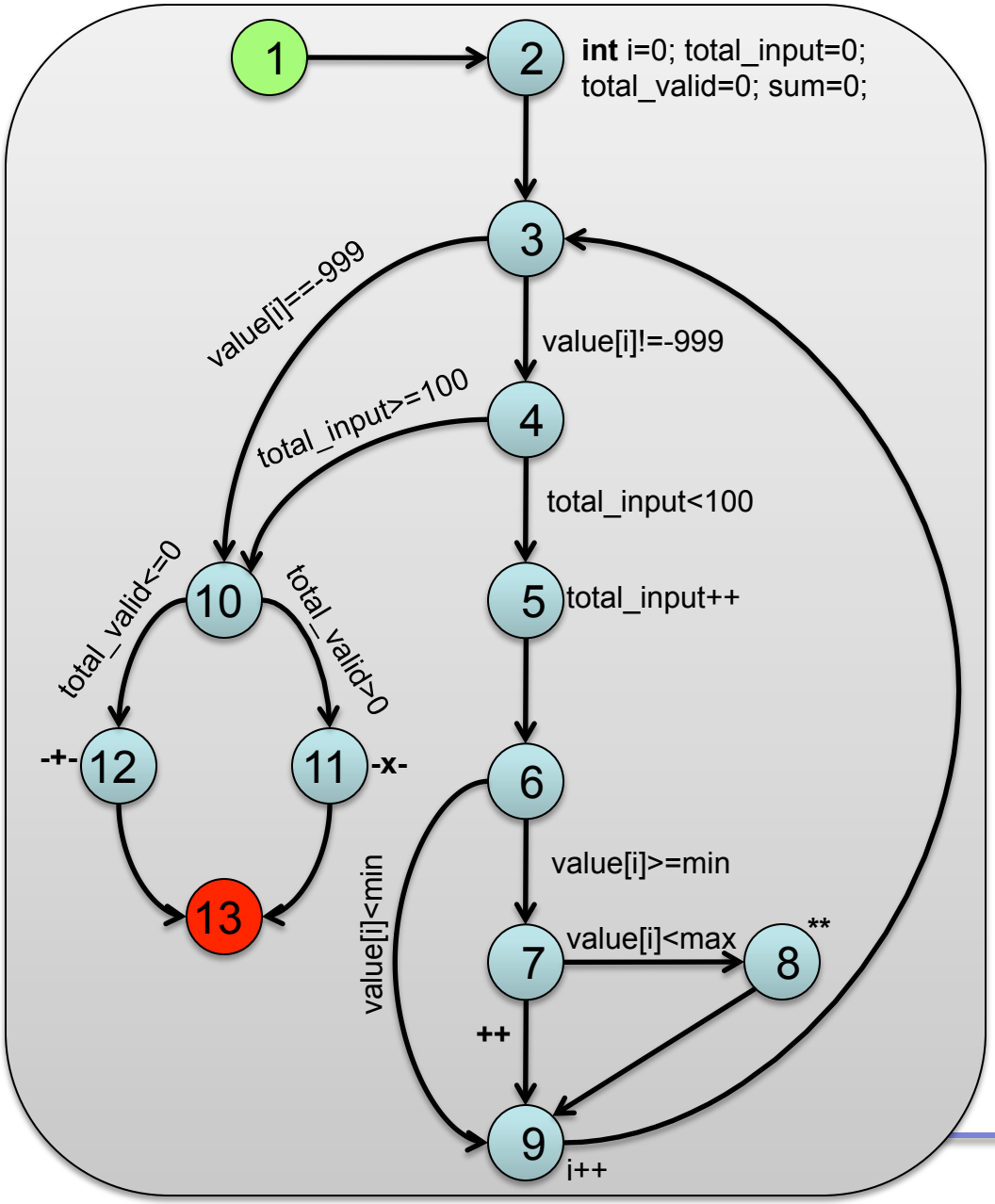
++ value[i]<max

** total_valid++; min+=value[i]

-+- average=-999

-x- average=sum/total_valid

BB – Critère des chemins indépendants, à vous de jouer!



Nombre cyclomatique : 17-13+2 = 6

Chemin 1

(couvrant max sans boucle)

[1 2 3 4 5 6 7 8 9 3 10 11 13]

DT min=1, max=10, value={2,-999}

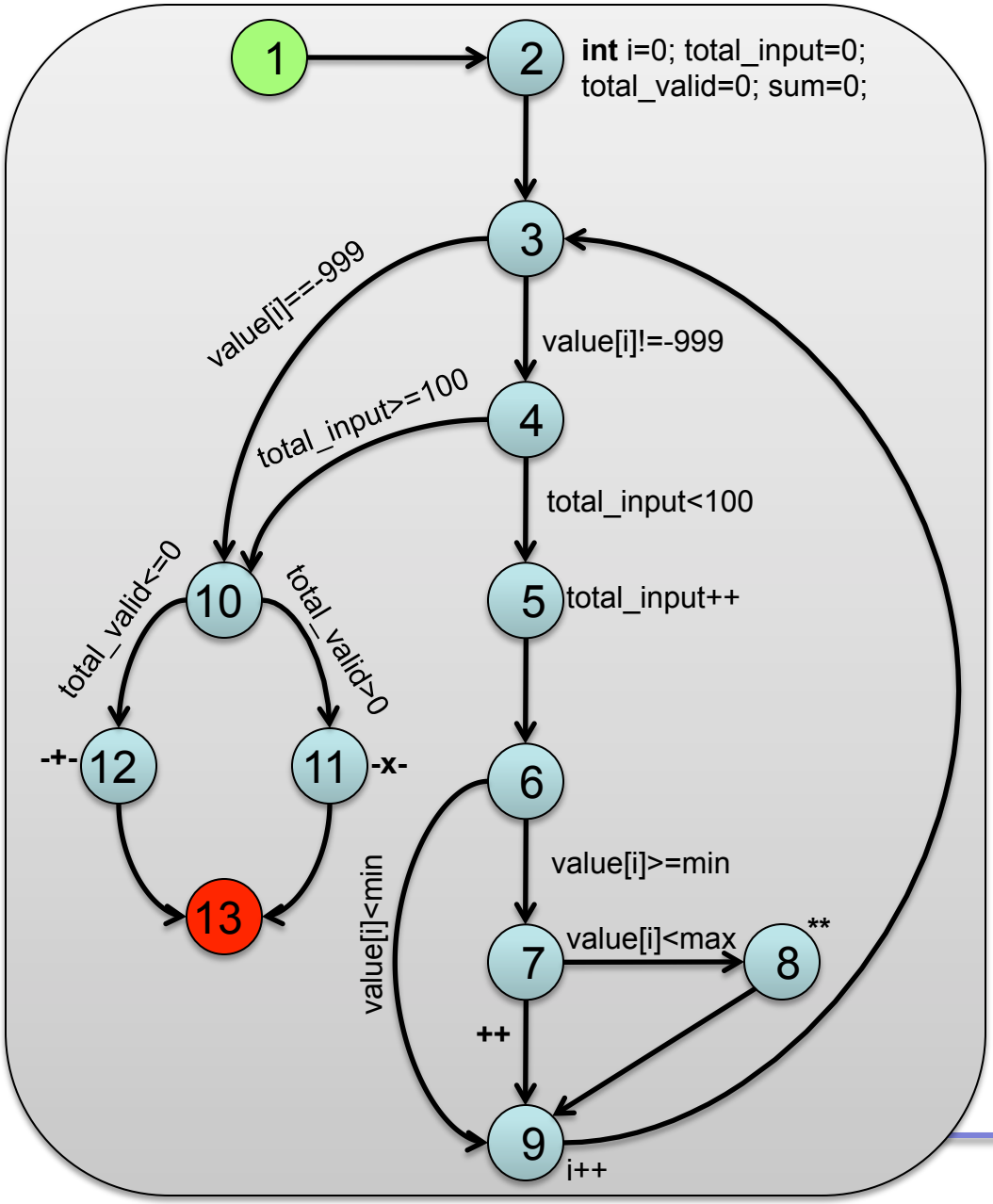
++ value[i]<max

** total_valid++; min+=value[i]

-+- average!=-999

-x- average!=sum/total_valid

BB – Critère des chemins indépendants, à vous de jouer!



Nombre cyclomatique : 17-13+2 = 6

Chemin 1

[1 2 3 4 5 6 7 8 9 3 10 11 13]

Chemin 2

On change la 1ère décision de 1 uniquement

[1 2 3 10 11 13]

Chemin non exécutable car total_valid reste nul

donc

[1 2 3 10 12 13]

DT min=1, max=10, value={-999}

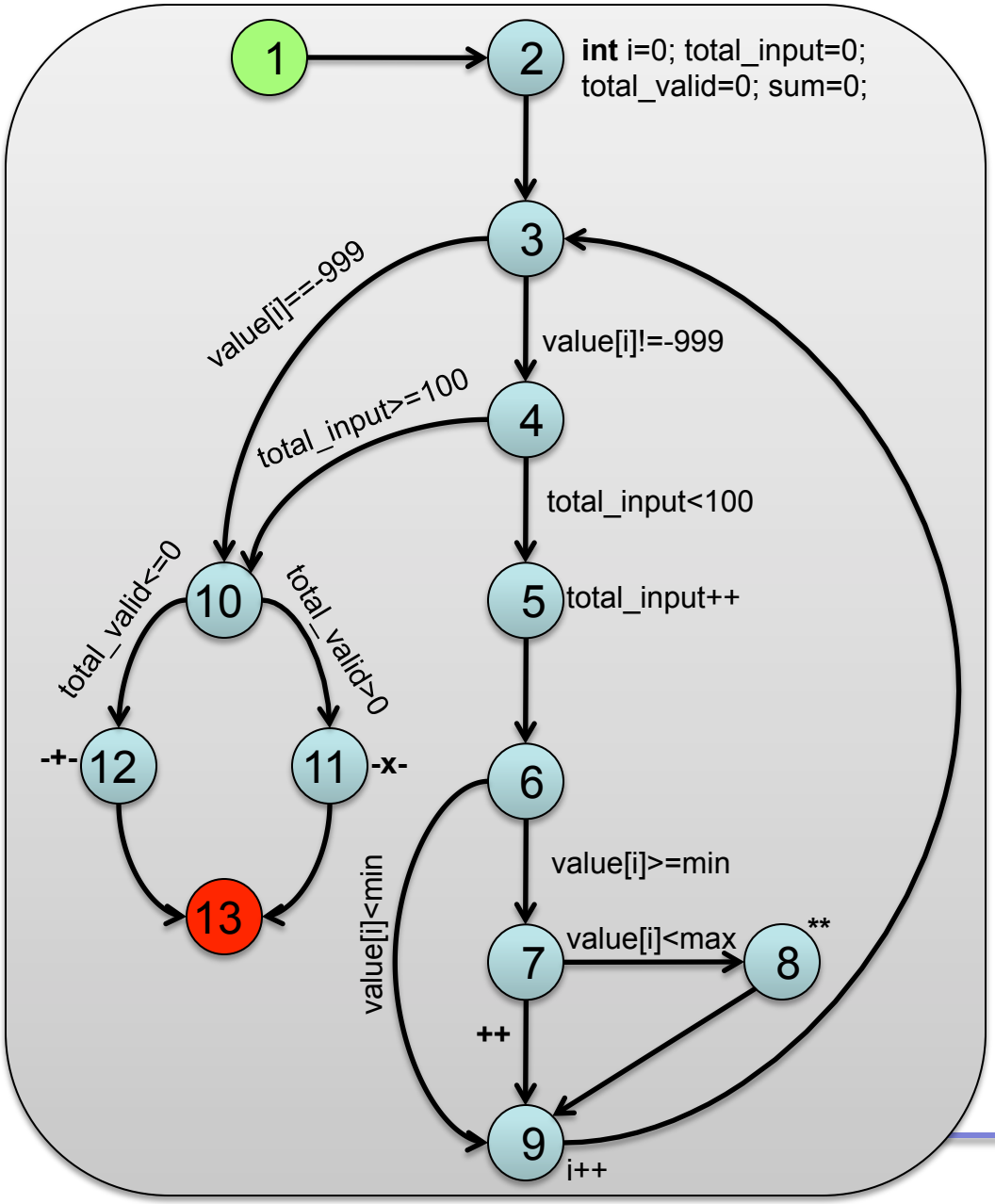
++ value[i]<max

** total_valid++; min+=value[i]

-+- average!=-999

-x- average!=sum/total_valid

BB – Critère des chemins indépendants, à vous de jouer!



Nombre cyclomatique : 17-13+2 = 6

Chemin 1

[1 2 3 4 5 6 7 8 9 3 10 11 13]

Chemin 2

[1 2 3 10 12 13]

Chemin 3

On change la 2nde décision de 1

[1 2 3 4 10 12 13]

Chemin non exécutable non plus, car la condition est dépendante de total_input!

Donc

[1 2 3(4 5 6 7 8 9 3)^{100}4 10 11 13]

DT min=0 max = 10 value un tableau de 100 fois la valeur 1

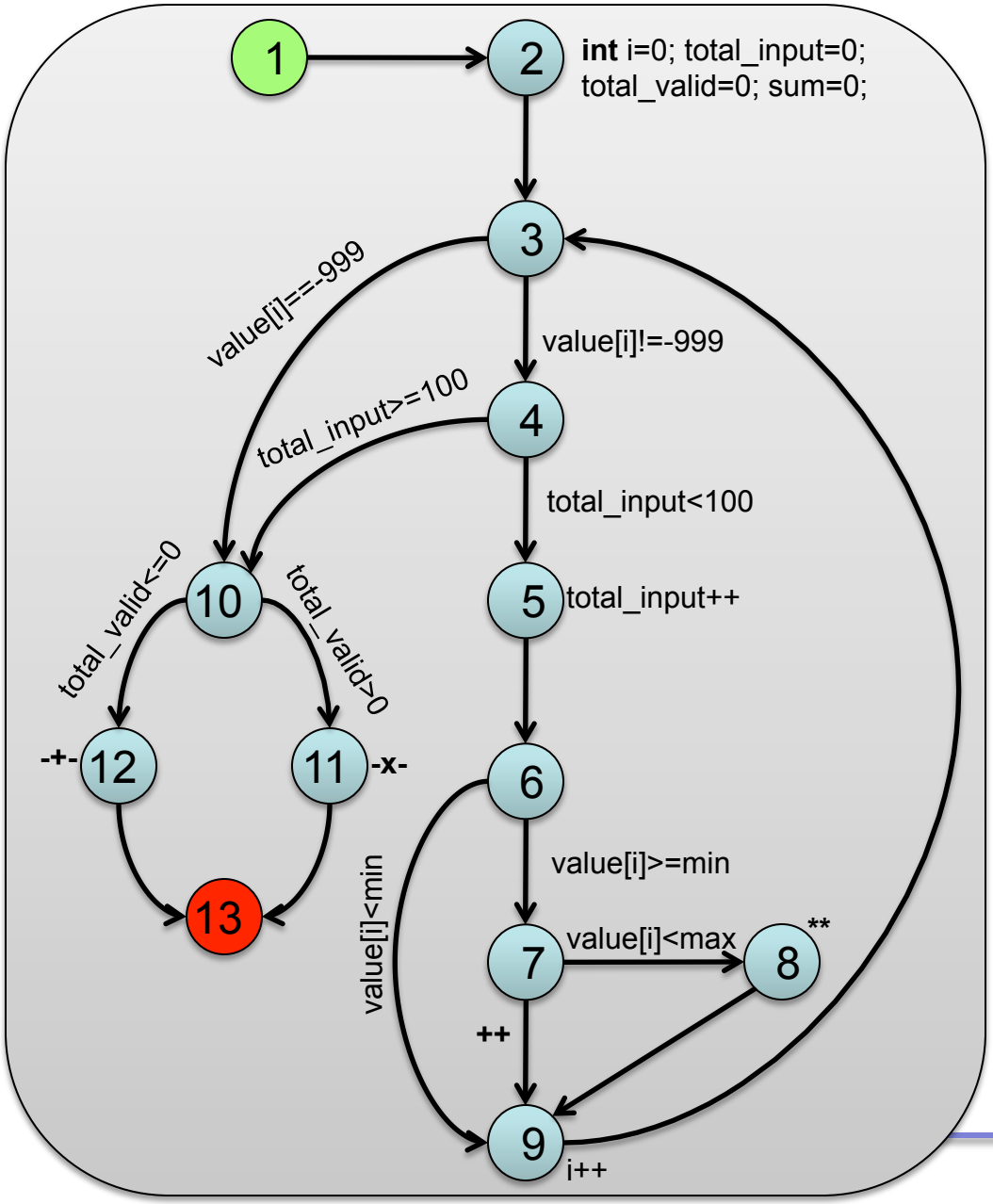
++ value[i]<max

** total_valid++; min+=value[i]

-+- average!=-999

-x- average!=sum/total_valid

BB – Critère des chemins indépendants, à vous de jouer!



Nombre cyclomatique : 17-13+2 = 6

Chemin 1

[1 2 3 4 5 6 7 8 9 3 10 11 13]

Chemin 2

[1 2 3 10 12 13]

Chemin 3

[1 2 3(4 5 6 7 8 9 3)^{100}4 10 11 13]

Chemin 4

On change la 4ème condition de 1

[1 2 3 4 5 6 9 3 10 12 13]

DT min =4 max = 10 value ={2,-999}

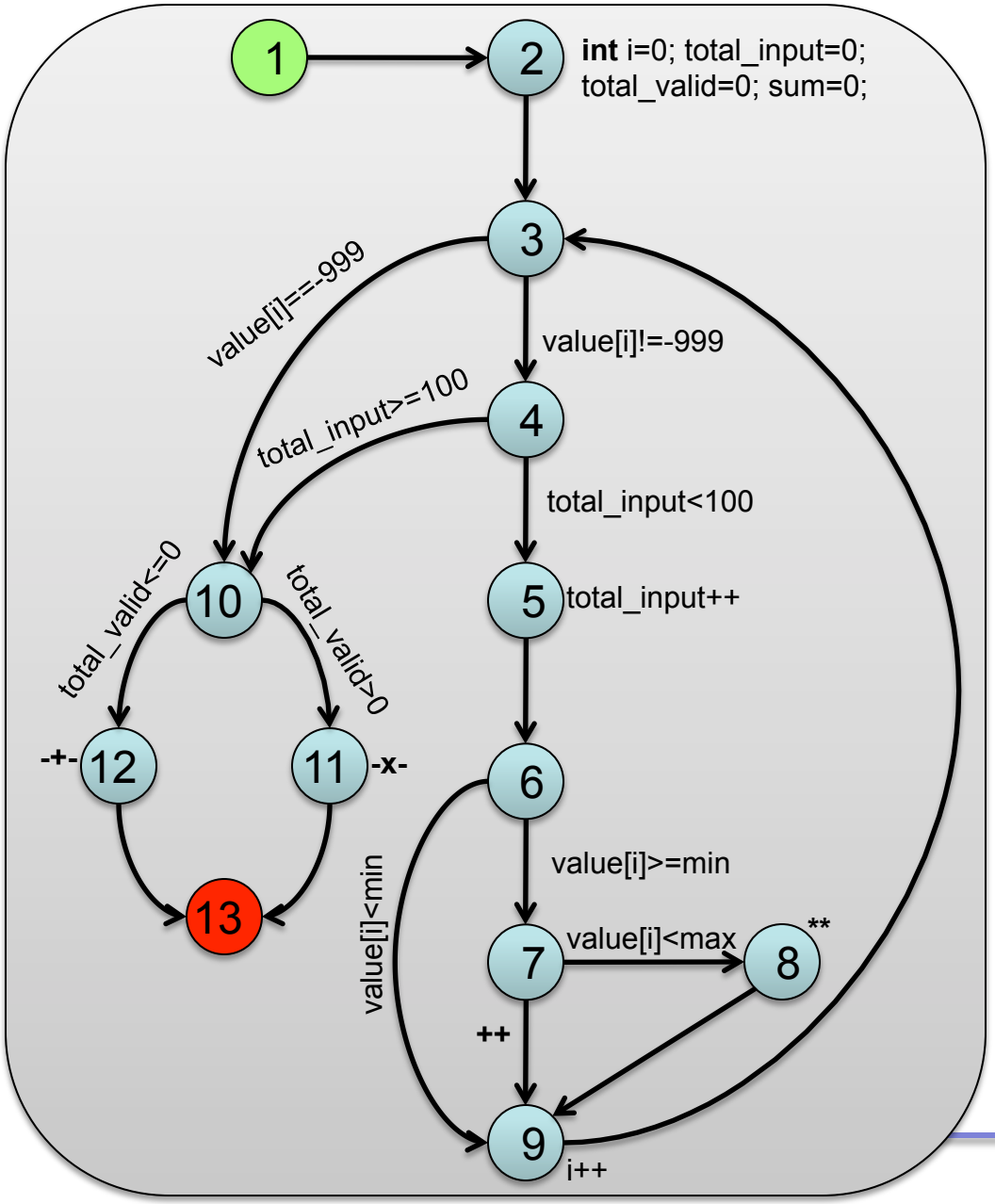
++ value[i]<max

** total_valid++; min+=value[i]

-+- average!=-999

-x- average!=sum/total_valid

BB – Critère des chemins indépendants, à vous de jouer!



Nombre cyclomatique : 17-13+2 = 6

Chemin 1

[1 2 3 4 5 6 7 8 9 3 10 11 13]

Chemin 2

[1 2 3 10 12 13]

Chemin 3

[1 2 3(4 5 6 7 8 9 3)^{100}4 10 11 13]

Chemin 4

[1 2 3 4 5 6 9 3 10 12 13]

Chemin 5

On change la 5ème condition de 1

[1 2 3 4 5 6 7 9 3 10 12 13]

DT min =0 max = 1 value ={2,-999}

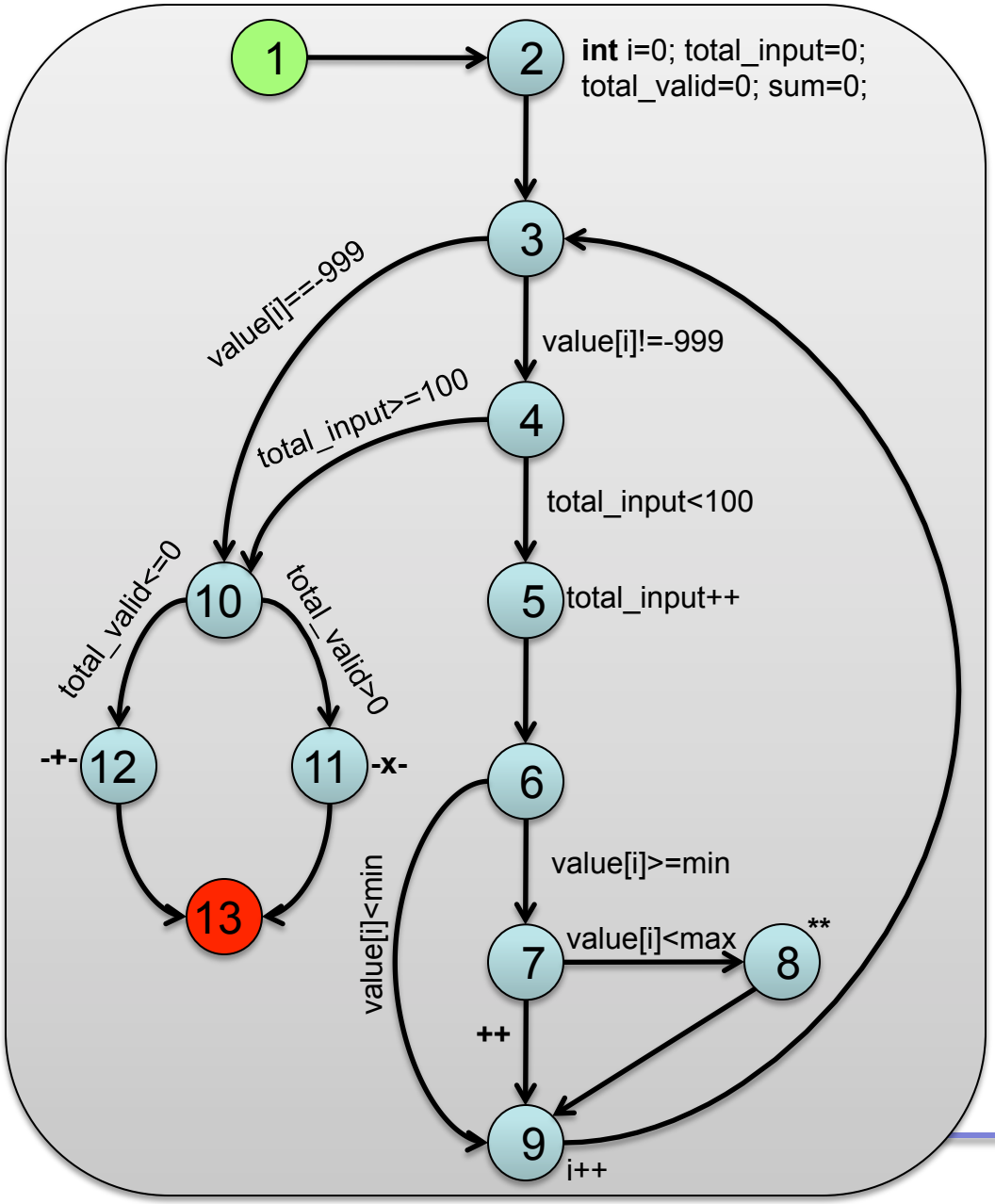
++ value[i]<max

** total_valid++; min+=value[i]

-+- average!=-999

-x- average!=sum/total_valid

BB – Critère des chemins indépendants, à vous de jouer!



Nombre cyclomatique : 17-13+2 = 6

Chemin 1

[1 2 3 4 5 6 7 8 9 3 10 11 13]

Chemin 2

[1 2 3 10 12 13]

Chemin 3

[1 2 3(4 5 6 7 8 9 3)^{100}4 10 11 13]

Chemin 4

[1 2 3 4 5 6 9 3 10 12 13]

Chemin 5

[1 2 3 4 5 6 7 9 3 10 12 13]

Chemin 6

On change la 6ème condition de 1

[1 2 3 4 5 6 7 8 9 3 10 12 13]

IMPOSSIBLE

++ value[i]<max

** total_valid++; min+=value[i]

-+- average!=-999

-x- average!=sum/total_valid

BB – Critère des PLCS

- Une PLCS est une portion linéaire de code suivie d'un saut, i.e. une séquence d'instructions entre deux branchements

Définition d'un saut

Soit un graphe de flot de contrôle *G*, une arête (**a**,**b**) de G est un saut si l'une des conditions suivantes est vérifiée :

- **a** est le nœud source de *G,*
- **b** est le nœud puits de *G,*
- **a** est une condition et **b** le nœud atteint dans le cas où **a** est évaluée à faux
- **b** est la condition d'une boucle et **a** le sommet terminal du corps de cette boucle

Définition d'une PLCS

Soit un graphe de flot de contrôle *G*, une PLCS est une couples [**c**,**s**] où $\mathbf{c} = \mathbf{s_1 s_2 \ldots s_n}$ est un chemin tel que

- $\mathbf{s_1}$ est le nœud d'arrivée d'un saut ou le nœud source
- $\mathbf{s_1 s_2 \ldots s_n}$ ne contient aucun saut
- ($\mathbf{s_n}$,**s**) est un saut

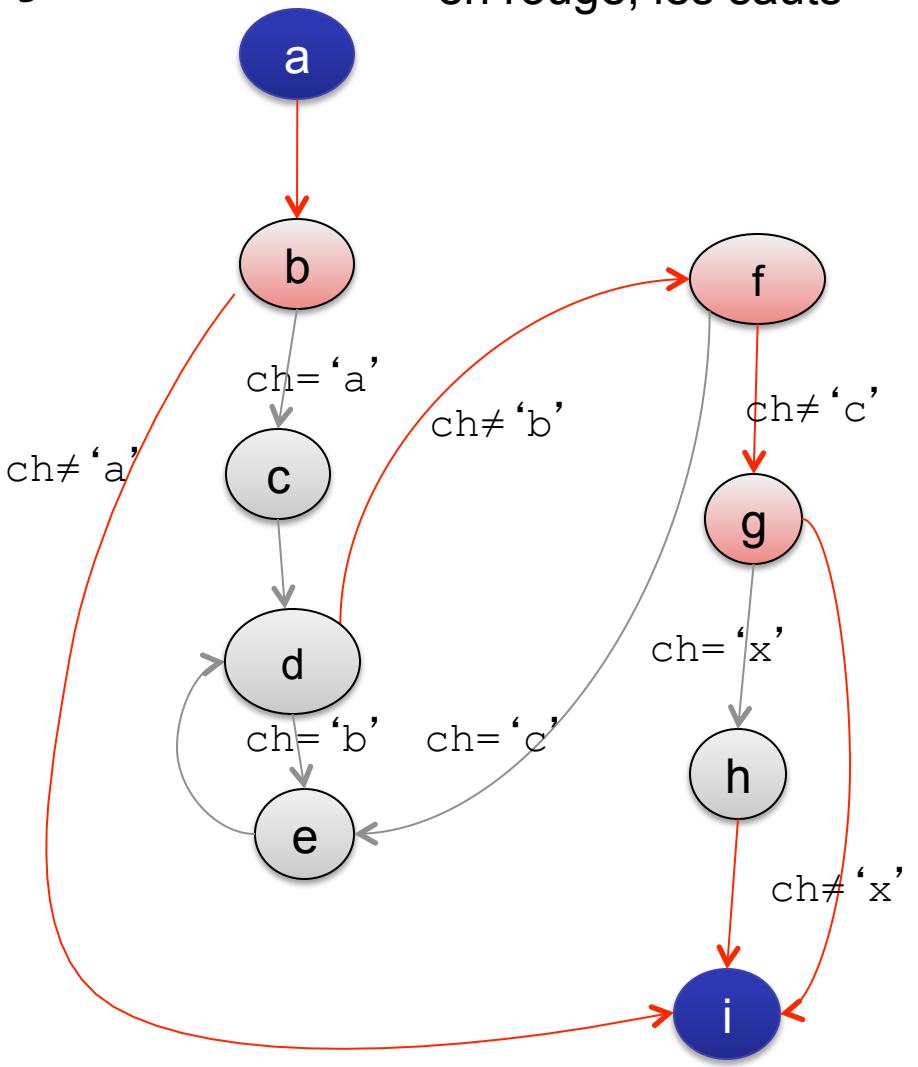
BB – Critère des chemins indépendants

Exemple – retour à `goodstring`

PLCS

1.[a, b]
2.[b, c, d, f]
3.[b, c, d, e, d, f]
4.[b, i]
5.[f, e, d, f]
6.[f, g]
7.[g, i]
8.[g, h, i]

en rouge, les sauts



BB – Graphe de flot de contrôle + Def/Use

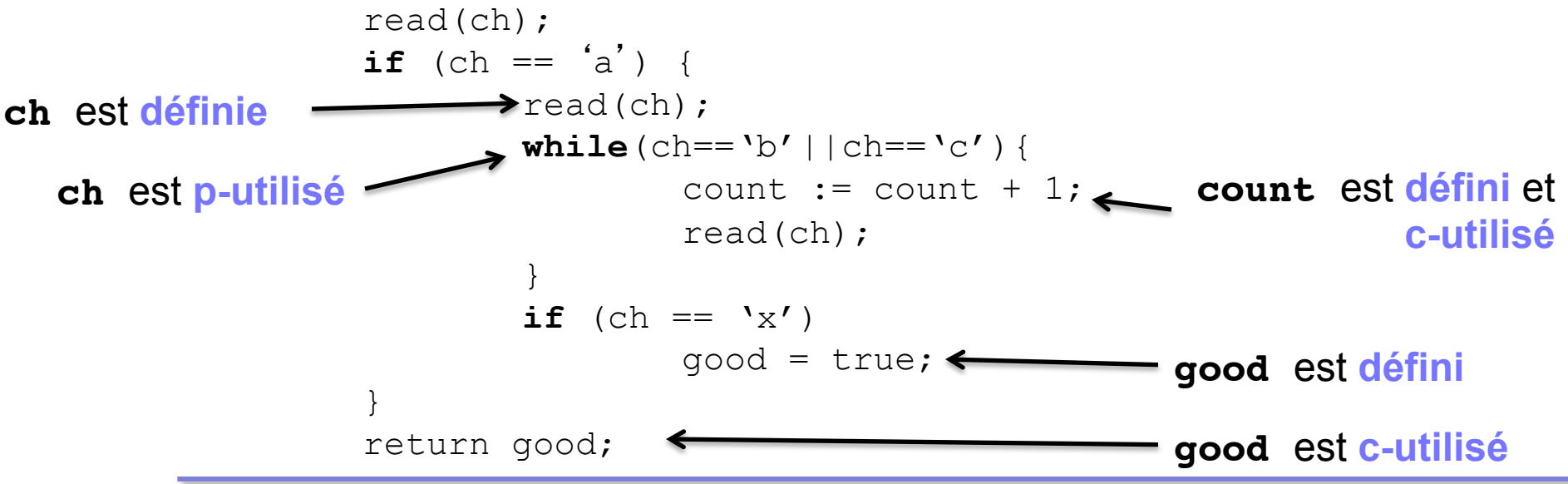
DEFINITIONS

- On analyse les relations entre instructions en tenant compte des variables qu'elles utilisent/définissent
- On va donc couvrir les différentes façons de définir et utiliser les variables :
  - Une variable est **définie** dans une instruction si sa valeur est modifiée
  - Une variable est **p-utilisée** si sa valeur est utilisée dans le prédicat d'une instruction de décision (**if**, **while**,…)
  - Une variable est **c-utilisée** si sa valeur est utilisée dans les autres cas (pour un calcul par exemple)

BB – Graphe de flot de contrôle + Def/Use

DEFINITIONS

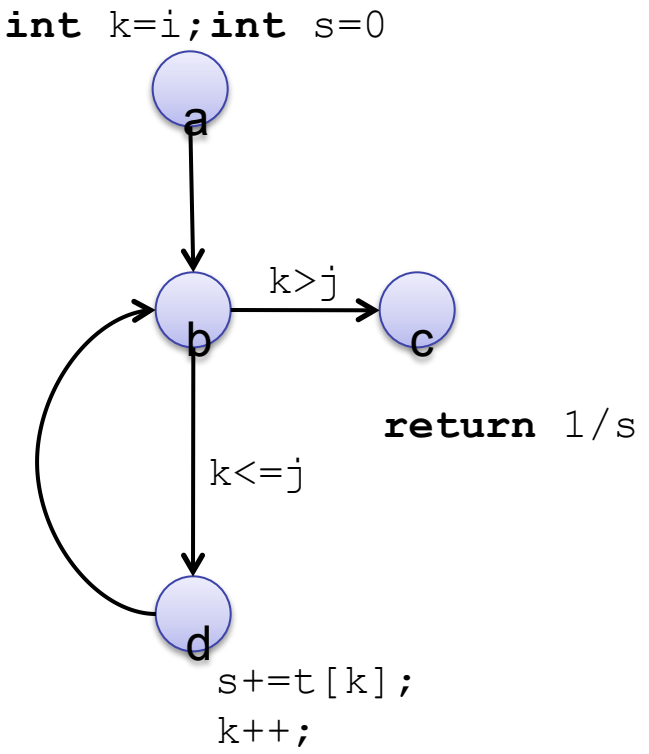
- On analyse les relations entre instructions en tenant compte des variables qu'elles utilisent/définissent
- On va donc couvrir les différentes façons de définir et utiliser les variables :
  - Une variable est **définie** dans une instruction si sa valeur est modifiée
  - Une variable est **p-utilisée** si sa valeur est utilisée dans le prédicat d'une instruction de décision (**if**, **while**,…)
  - Une variable est **c-utilisée** si sa valeur est utilisée dans les autres cas (pour un calcul par exemple)



BB – Graphe de flot de contrôle + Def/Use

- Une instruction **l** est **utilisatrice** d'une variable **x** par rapport à une instruction de définition **J** si:
  - **x** est définie en **l** et référencé en **J**
  - **x** n'est pas redéfinie entre **l** et **J**

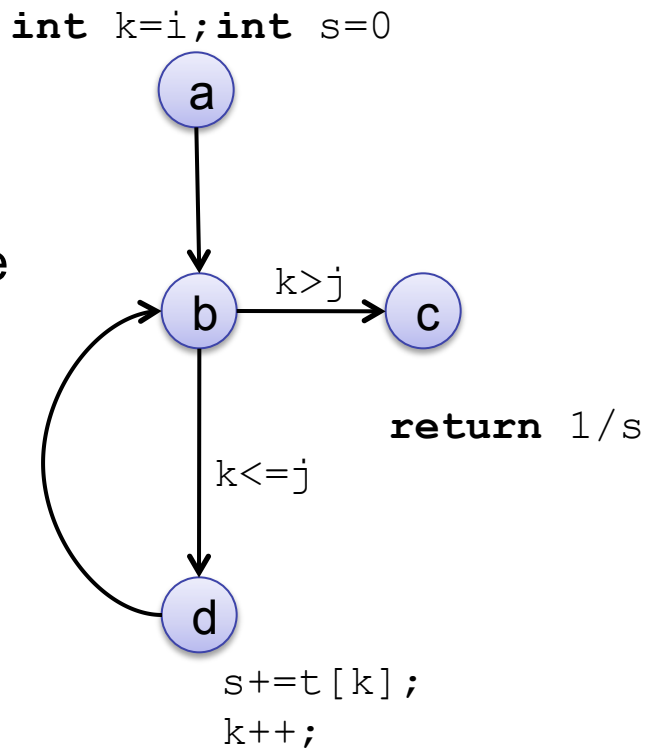
- Exemples
  - L'arc (**b**, **c**) est p-utilisateur de k par rapport au nœud **a**
  - L'instruction **`return`** `1/s` est c-utilisatrice de s par rapport au nœud **a**



BB – Graphe de flot de contrôle + Def/Use

- Un chemin d'utilisation (c-utilisation ou p-utilisation) relie l'instruction de définition d'une variable à une instruction utilisatrice

- Exemples
  - Le chemin [**a**, **b**, **c**] est un chemin de p-utilisation de k
  - C'est aussi un chemin de c-utilisation de s
  - Le chemin [**a**, **b**, **d**] est un chemin de p-utilisation et de c-utilisation de la variable k



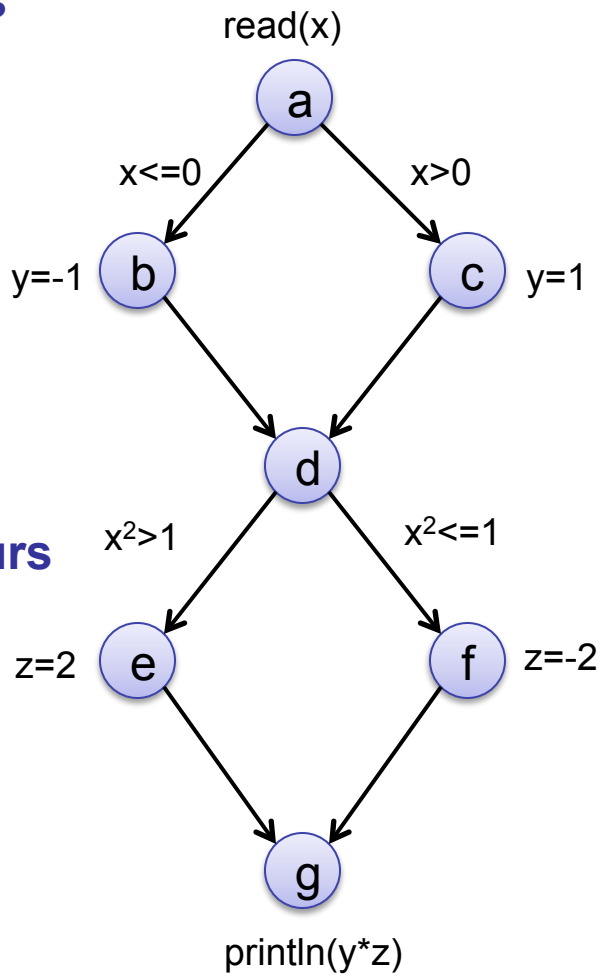
BB – Graphe de flot de contrôle + Def/Use

Critères de couverture sur GFC + Def/Use

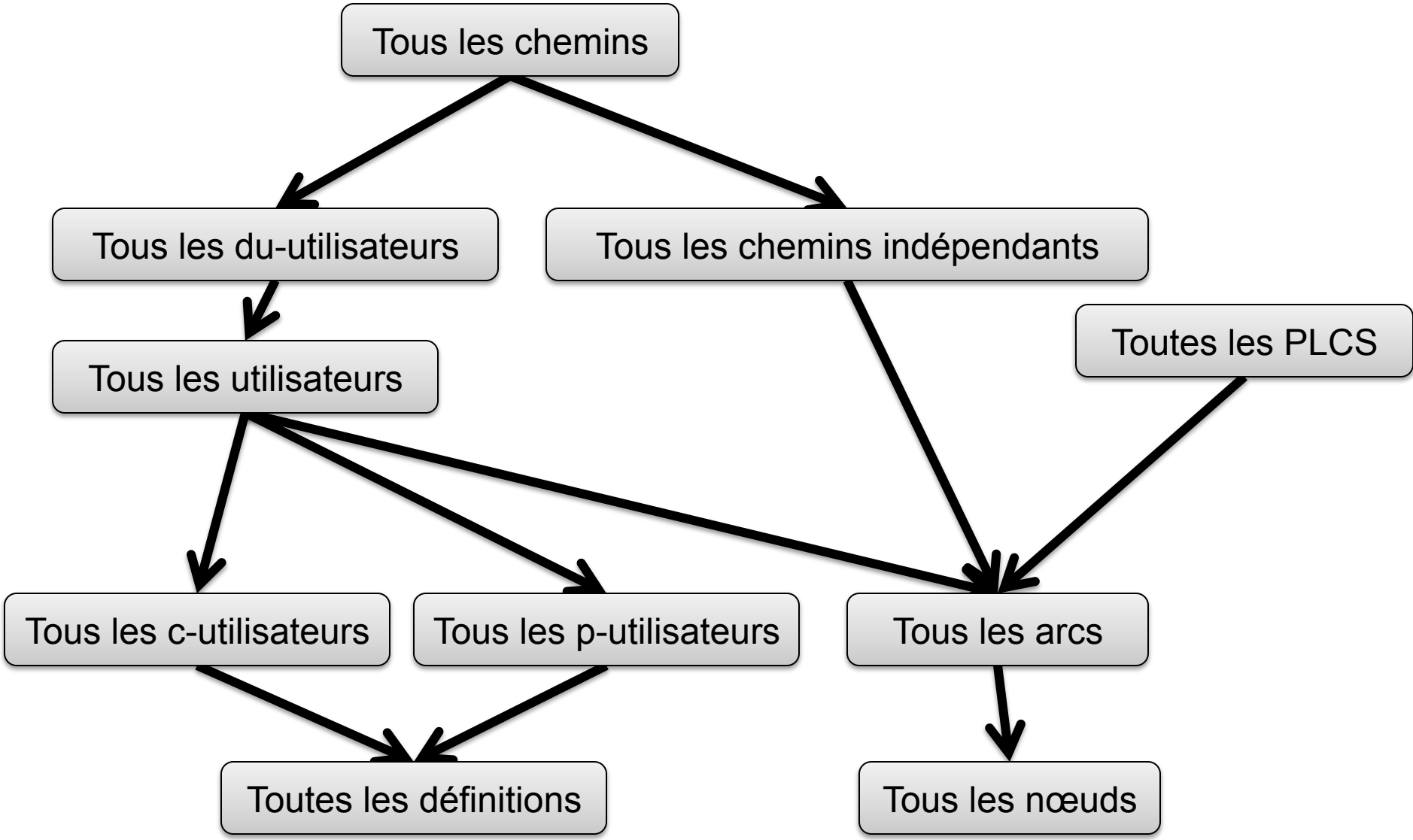
- Tous les utilisateurs
  - Nécessite la couverture de tous les utilisateurs (nœuds c-utilisateurs et arcs p-utilisateurs) pour chaque définition et pour chaque référence à cette définition
- Tous les c-utilisateurs
  - Nécessite la couverture de tous les nœuds c-utilisateurs pour chaque définition et pour chaque référence à cette définition
- Tous les p-utilisateurs
  - Nécessite la couverture de tous les arcs p-utilisateurs pour chaque définition et pour chaque référence à cette définition
- Toutes les définitions
  - Nécessite la couverture d'au moins un chemin d'utilisation pour chaque définition
- Tous les du-utilisateurs du=definition/utilisation
  - Tous les utilisateurs + la couverture de tous les chemins possibles entre définition et référence (sans les cycles)

BB – Graphe de flot de contrôle + Def/Use

- Couverture du critère **toutes les définitions**
  - [a, b, d, f, g]
  - [a, c, d, e, g]
- Couverture du critère **tous les utilisateurs**
  - [a, b, d, f, g] les mêmes!
  - [a, c, d, e, g]
- Couverture du critère **tous les du-utilisateurs**
  - [a, b, d, f, g]
  - [a, b, d, e, g]
  - [a, c, d, f, g]
  - [a, c, d, e, g]

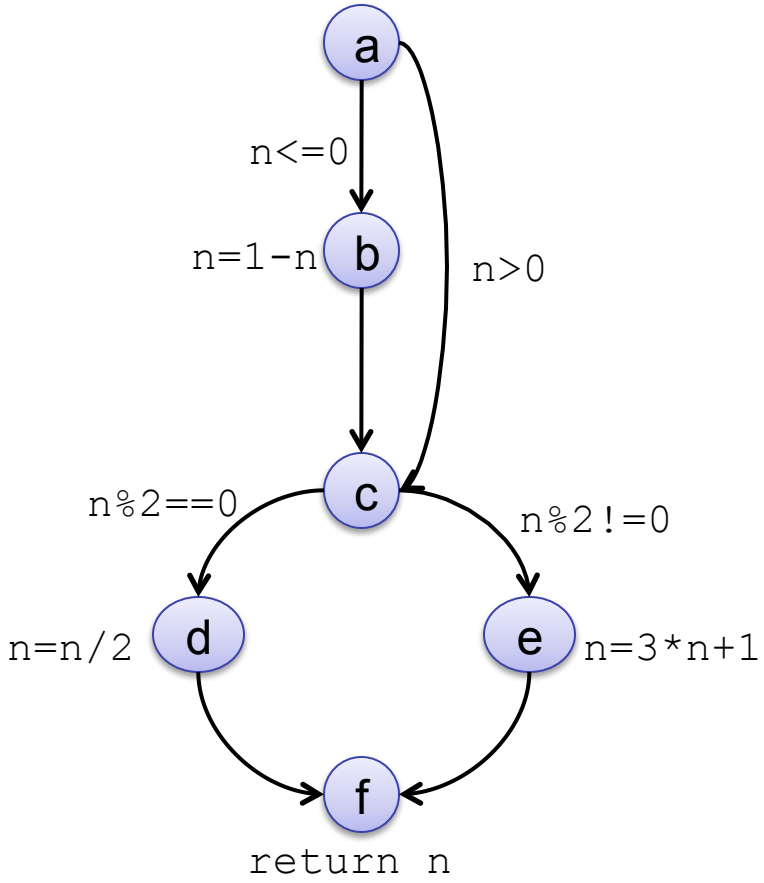


Hiérarchie des tests sur les GFCs



BB –Retour Exercice 1

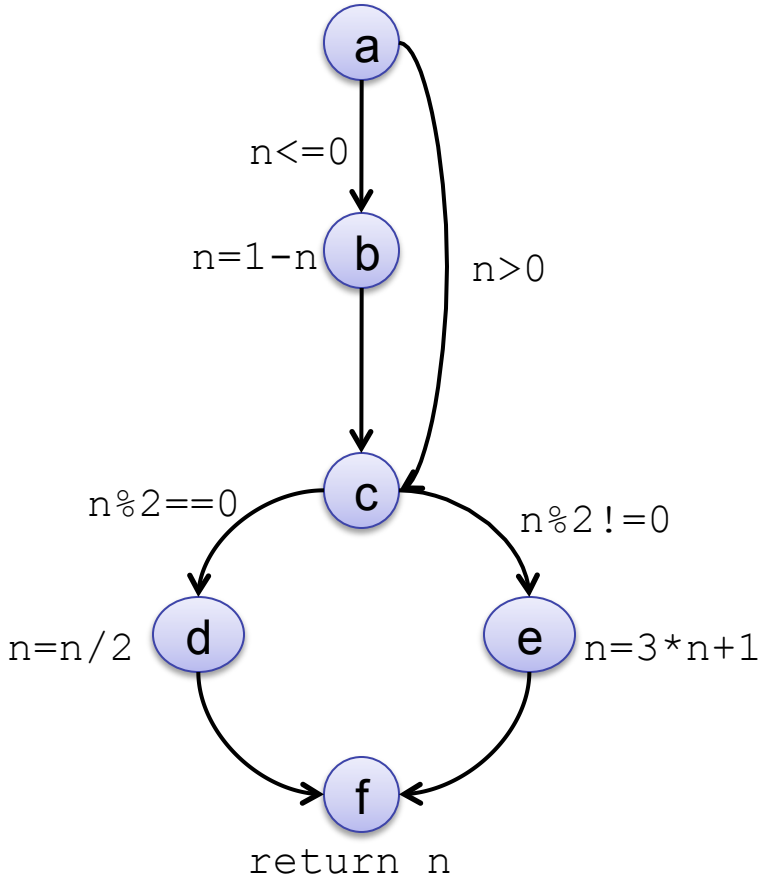
```
int f(int n){
 if (n<=0)
   n = 1-n;
 if (n%2==0)
   n = n/2;
 else
   n = 3*n+1;
 return n;
}
```



1. Donner des DTs pour tous les nœuds,
2. Donner des DTs pour tous les arcs,
3. Donner des DTs pour tous les chemins indépendants
4. Donner des DTs pour toutes les PLCS

BB –Retour Exercice 1

```
int f(int n){
 if (n<=0)
   n = 1-n;
 if (n%2==0)
   n = n/2;
 else
   n = 3*n+1;
 return n;
}
```

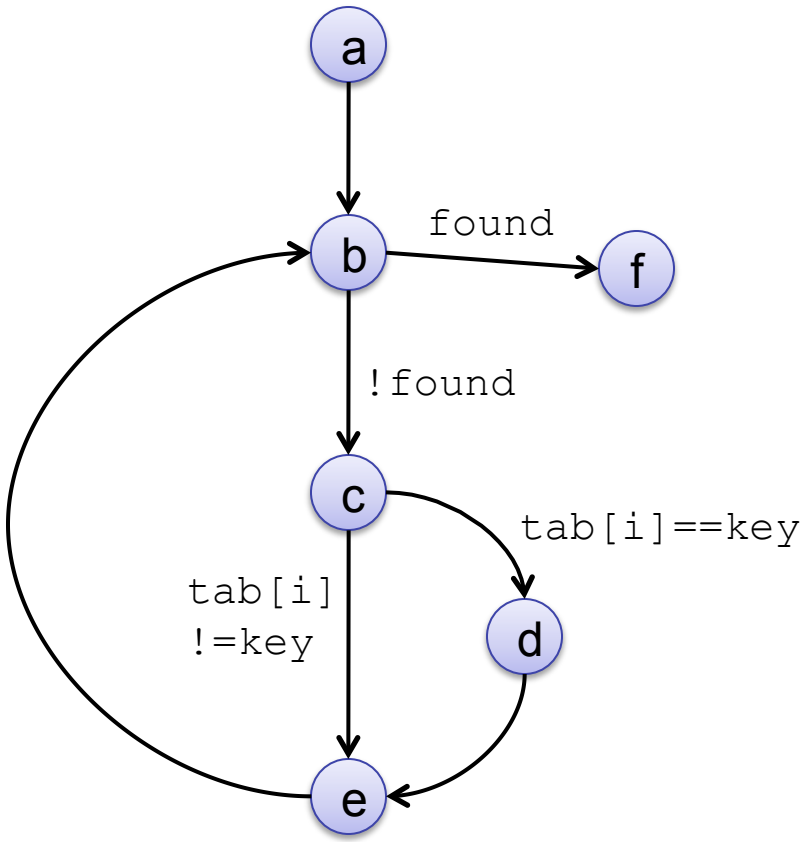


1. Donner des DTs pour tous les nœuds, **n = 0, n = -1**
2. Donner des DTs pour tous les arcs, **n = 2, n = -2**
3. Donner des DTs pour tous les chemins indépendants **n=-1,n=1, n=3**
4. Donner des DTs pour toutes les PLCS **[a,b,c], [a,c], [c,d,f], [c,e,f]**

n=-1, n=1

BB – Retour Exercice 2

```
int f(int* tab, int key){
  int i = 0;                    a
  int res;
  bool found = false;
  while(!found){
    if(tab[i]==key){
      found=true;               d
      res=i;
    }
    i = i+1;                    e
  }
  return res;                   f
}
```



1. Donner des DTs pour tous les nœuds, **[1], 1**
2. Donner des DTs pour tous les arcs, **[2,1], 1**
3. Donner des DTs pour tous les chemins indépendants , **[1], 1 – [2,1], 1 – IMP**
4. Donner des DTs pour toutes les PLCS **[a,b], [b,f] , [b,c,e] , [b,c,d,e,b]**

BB – Exercice

```
int a=0, b=0, p=0;
read(a, b)
if (a<0)
   b = b-a;
if (b%2==0)
   p = b*b;
else
   p = 2*a;
println p;
```

1. Fournir le graphe de flot de contrôle
2. Donner des DTs pour tous les nœuds,
3. Donner des DTs pour tous les arcs,
4. Donner des DTs pour tous les chemins indépendants
5. Donner des DTs pour toutes les PLCS

Plan du cours

- Définitions autour du test
  - Métriques de qualité/fiabilité logicielle
  - Positionnement du test dans le cycle de vie du logiciel
  - Test et méthodes agiles
  - Qu'est-ce que le test ?
- Processus de test simplifié
- Comment choisir les scénarios de test à jouer ?
  - Les méthodes de test Boîte Noire
  - Les méthodes de test Boîte Blanche
- Tests unitaires avec JUnit et EclEmma
  - Présentation de JUnit
  - Couverture de test avec EclEmma
- Utilisation de doublures (mocks) pour les tests unitaires avant intégration

Junit, un outil pour le TDD

JUnit

- Junit
  - plateforme de tests unitaires pour Java (déclinée pour de nombreux autres langages) écrite par Erich Gamma and Kent Beck
  - Dédié au test « boite blanche » de type unitaire ou intégration
  - Intégré à Eclipse et Maven
  - Version actuelle 4.x basée sur l'utilisation des annotation Java 5
  - Disponible à http://www.junit.org/

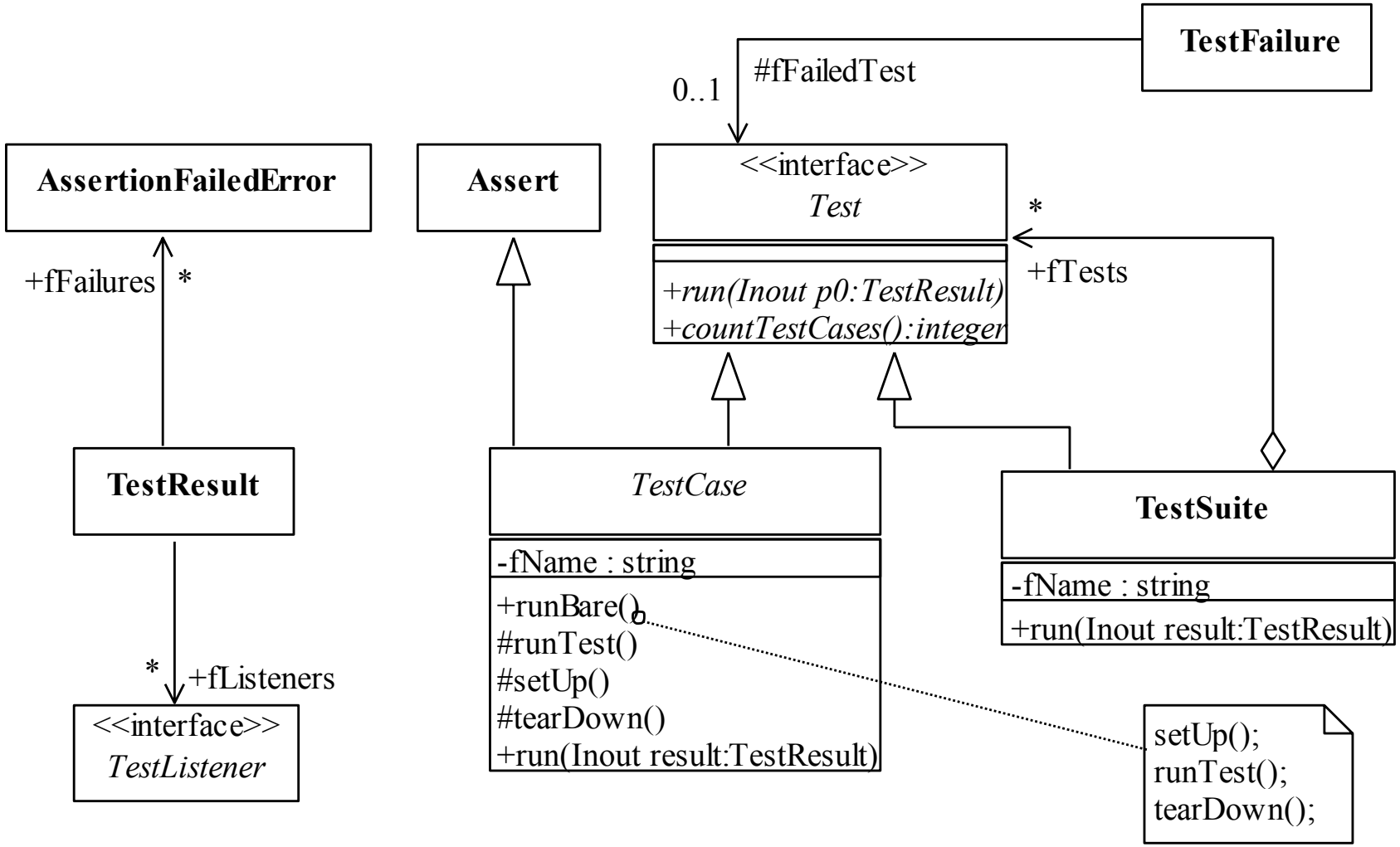
- Junit propose
  - D'écrire facilement des tests unitaires
  - D'utiliser des assertions pour exprimer les oracles
  - Le lancement automatique de suites de test
  - Un formatage du diagnostic

Tests en JUnit

- Les tests sont regroupés selon leur application aux méthodes d'une classe donnée
- Chaque test vérifie le comportement d'une méthode sous certaines conditions
  - Vérification des comportements corrects
  - Vérification des comportements incorrects (valeurs incohérentes des paramètres, ...)
  - Vérification des levées d'exceptions
- Écrits par le programmeur de la classe
- Surtout utiles pour tester qu'une classe reste valide après des modifications du code
- Les tests sont regroupés sous forme de « suite de tests » exécutés par un « Suite runner »

Junit:Framework

JUnit



Test par Assertions

- Une assertion = une confrontation avec l'oracle
- Méthodes statiques de la classe `org.junit.Assert`
  - `assertEquals`
  - `assertTrue`
  - `assertFalse`
  - `assertNull`
  - `assertNotNull`
  - `assertSame`
  - `assertNotSame`
- Paramètres d'invocation
  - `expected, actual`
  - `expected, actual, delta`
  - condition
  - ... + message

Méthode de test

- Méthode qui exécute un test unitaire
- Convention de nommage *test[méthode à tester]()*
  - Mais aucune obligation (nom quelconque possible)
- Utilise l'annotation **`@Test`**
- Publique, sans paramètre, type de retour `void`

```
@Test
public void testSetMontant(){
  compte.setMontant(100000.0);
  assertEquals(compte.getMontant(), 100000.0);
}
```

- L'annotation **`@Ignore`** permet d'ignorer un test

Levée d'exception et échec

- Le test d'une levée d'exception s'écrit comme suit

```
@Test(expected=ArrayIndexOutOfBoundsException.class)
public void testElementAt(){
  Vector<String> v = new Vector<String>();
  String s = v.elementAt(2);
}
```

- Une méthode peut provoquer l'échec d'un test

```
@Test
public void testSetMontant() {
  if (compte == null)
    fail("Erreur d'initialisation du test");
  compte.setMontant(100000.0);
  assertEquals(compte.getMontant(), 100000.0);
}
```

Méthodes d'initialisation et de finalisation

- Méthodes d'initialisation utilisée avant chaque test
  - setUp() est une convention de nommage
  - L'annotation **@Before** est utilisée

```
@Before
public void setUp(){
  compte = new Compte();
}
```

- Méthodes de finalisation utilisée après chaque test
  - tearDown() est une convention de nommage
  - L'annotation **@After** est utilisée

```
@After
public void tearDown(){
  declaration = null;
}
```

- Dans les 2 cas, possibilité d'annoter plusieurs méthodes
  - Ordre d'exécution indéterminé

Méthodes d'initialisation et de finalisation

- Méthodesd'initialisation globale
  - Annotée `@BeforeClass`
  - Publique et statique
  - Exécutée une seule fois avant la première méthode de test
- Méthode de finalisation globale
  - Annotée `@AfterClass`
  - Publique et statique
  - Exécutée une seule fois après la dernière méthode de test
- Dans les 2 cas, une seule méthode par annotation

Tests unitaires fonctionnels et structurels

- Outils utilisé
  - Eclipse
  - JUnit
  - EclEmma

- *Exemple*: fonction retournant la somme de 2 entiers modulo 20 000

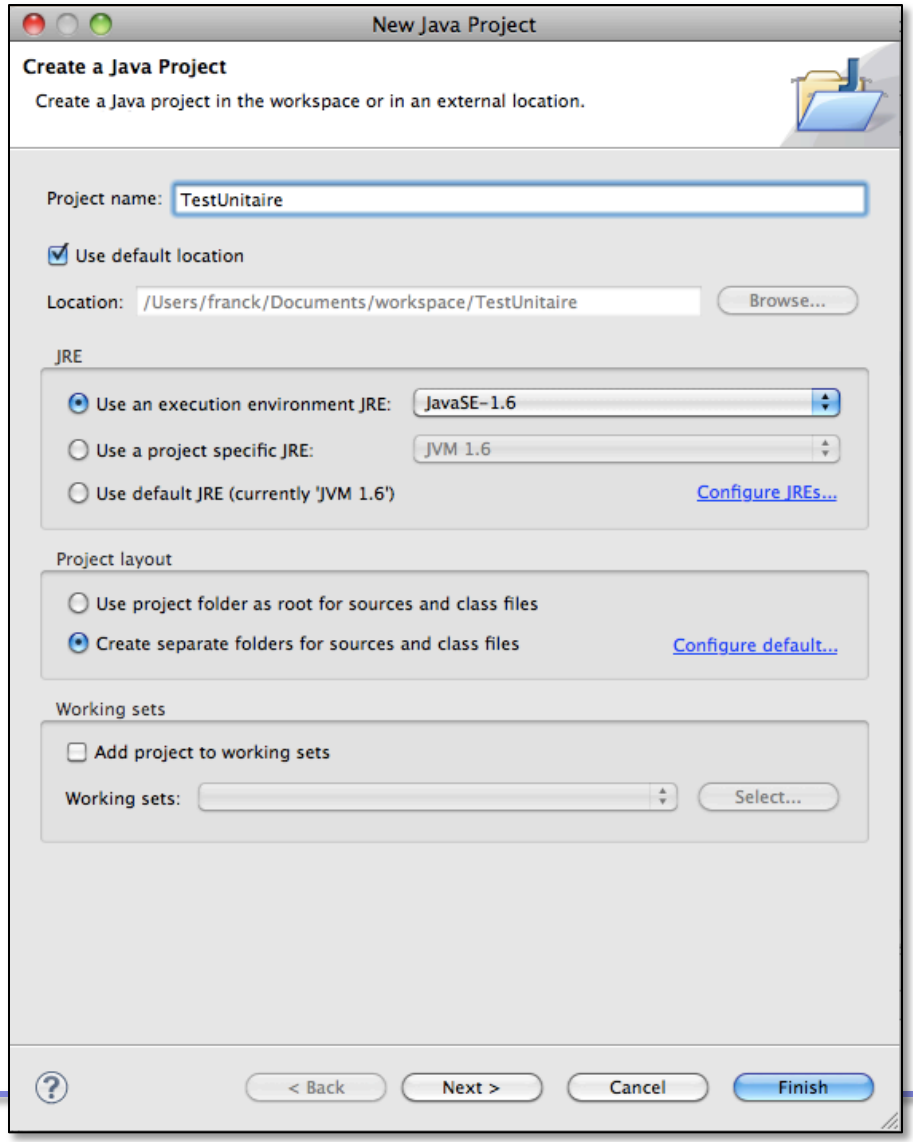
```
fun (x:int, y:int) : int =
        if (x==500 and y==600) then x-y (bug 1)
                               else x+y (bug 2)
```

  - Avec l'approche fonctionnelle (boîte noire),
    - le bug 1 est difficile à détecter, le bug 2 est facile à détecter

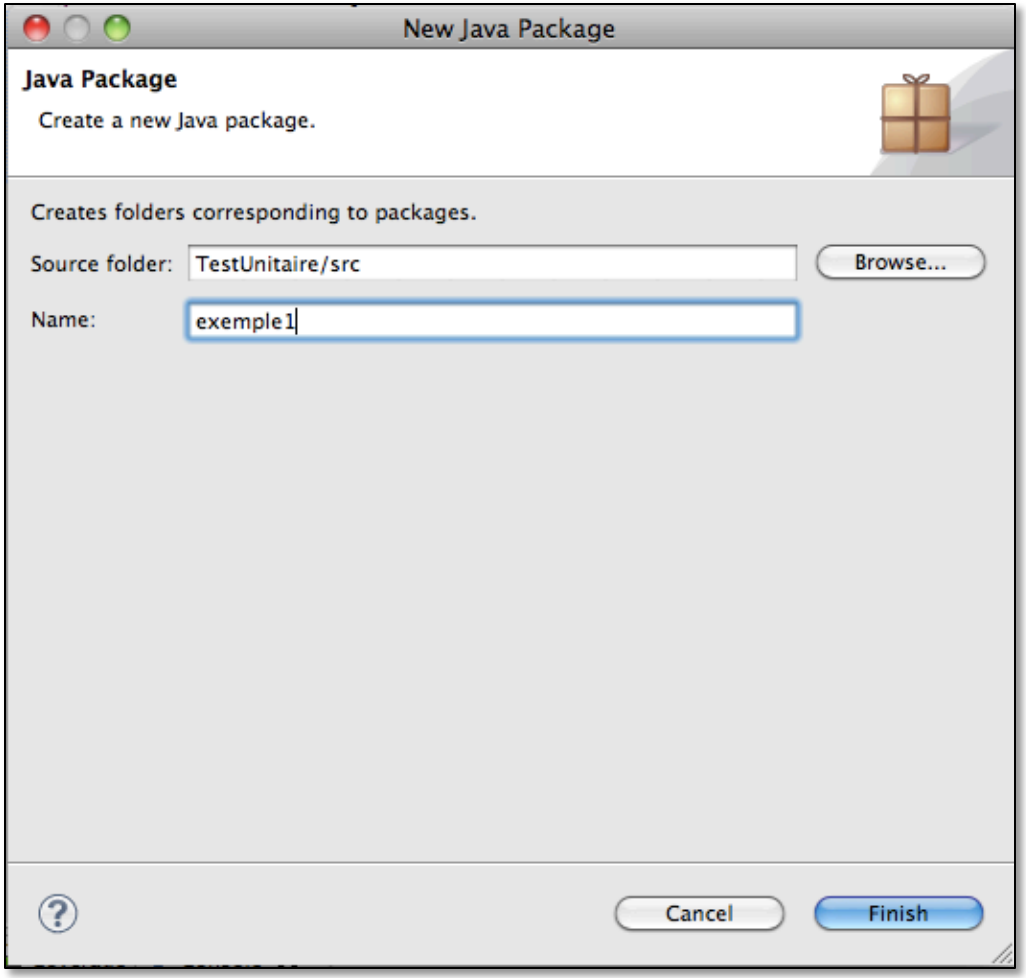
  - Avec l'approche structurelle (boîte blanche),
    - le bug 1 est facile à détecter, le bug 2 est difficile à détecter

Tests unitaires fonctionnels et structurels

- Création d'un projet Eclipse Java → TestUnitaire

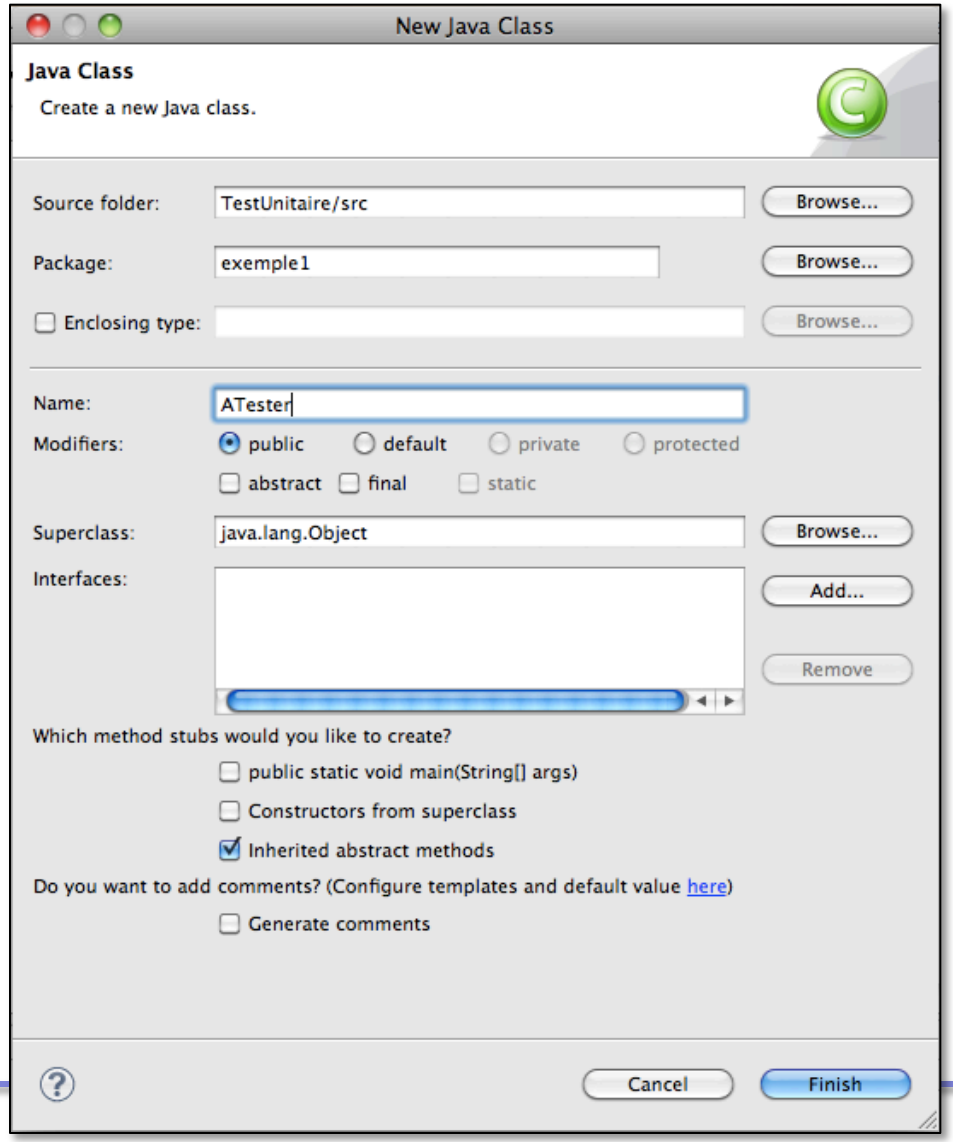


- Dans le projet, création d'un package « exemple1 »



- Puis d'un package exemple1.test

- Dans le package « exemple1 », création d'une classe « ATester »



- Enfin écriture du code de l'opération fun

```
fun (x:int, y:int) : int =
        if (x==500 and y==600) then x-y (bug 1)
                               else x+y (bug 2)
```

```
package exemple1;

public class ATester {
    public int f(int x, int y){
        if(x==500 && y==600)
            return y-x;
        else
            return x+y;
    }
}
```

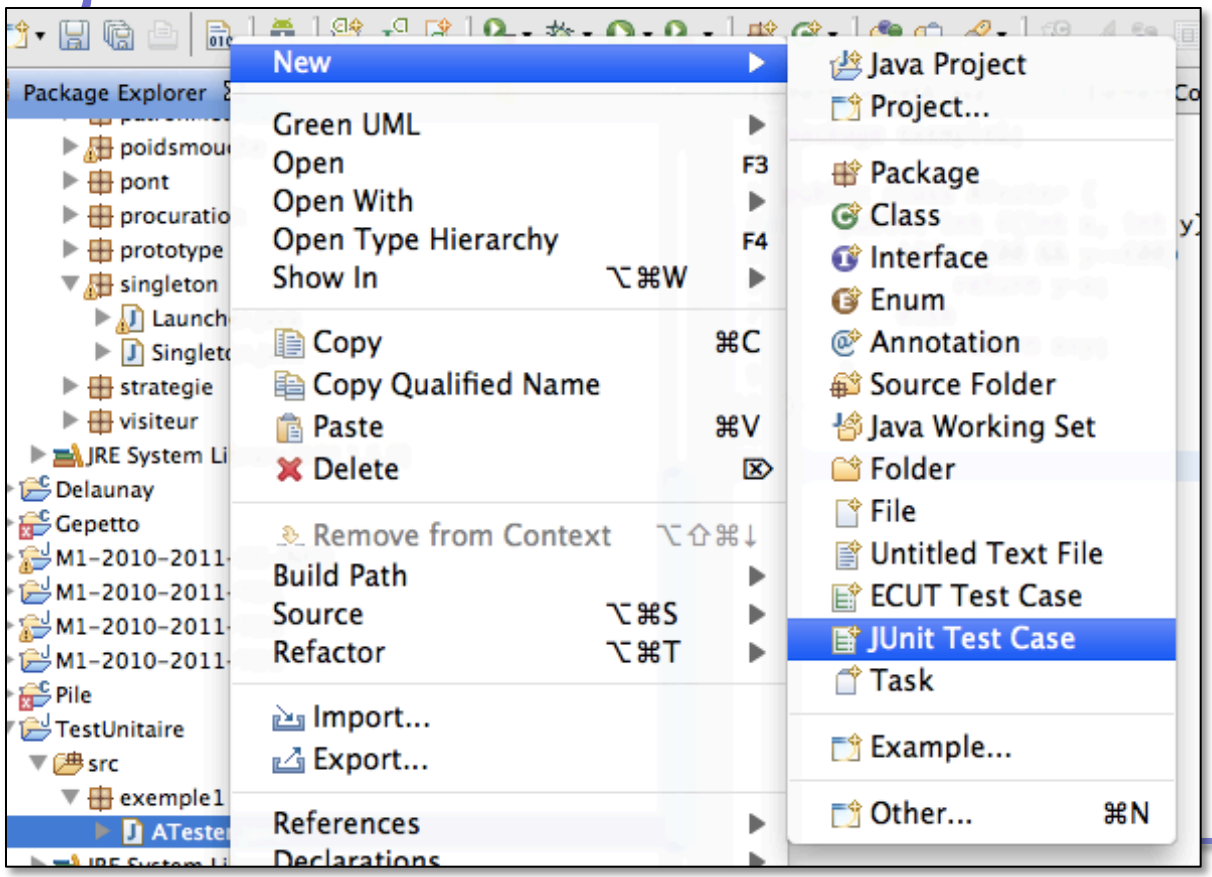
Tests unitaires fonctionnels et structurels

- On commence par le test fonctionnel
  - *La fonction doit retourner la somme de 2 entiers modulo 20000*
  - Test 1 - DT : {x=10, y=20}, prédiction de l'oracle : 30
  - Test 2 - DT : {x=10000, y=10050}, prédiction de l'oracle : 50

Tests unitaires fonctionnels et structurels

- On commence par le test fonctionnel
  - *La fonction doit retourner la somme de 2 entiers modulo 20000*
  - Test 1 - DT : {x=10, y=20}, prédiction de l'oracle : 30
  - Test 2 - DT : {x=10000, y=10050}, prédiction de l'oracle : 50

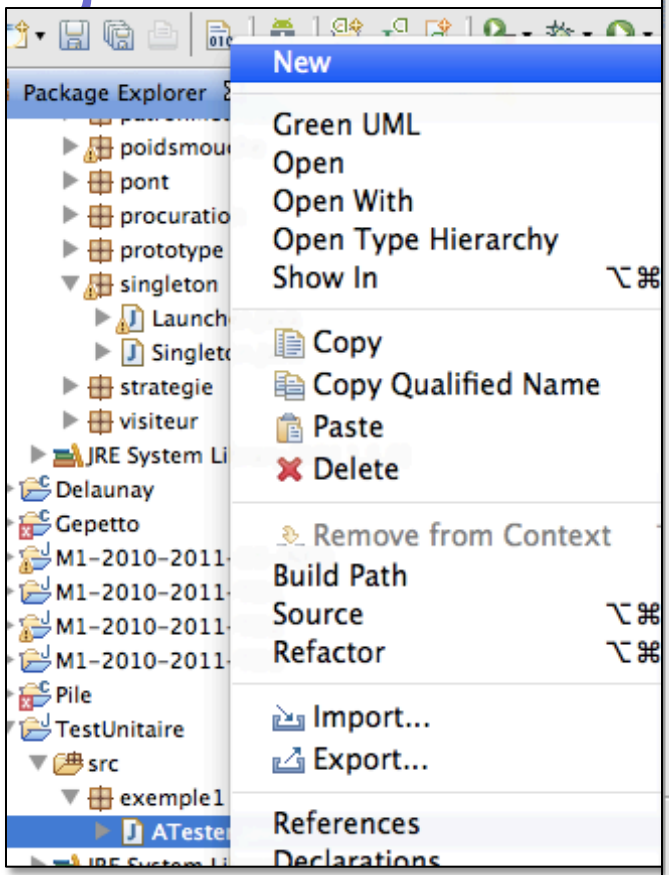
Unit

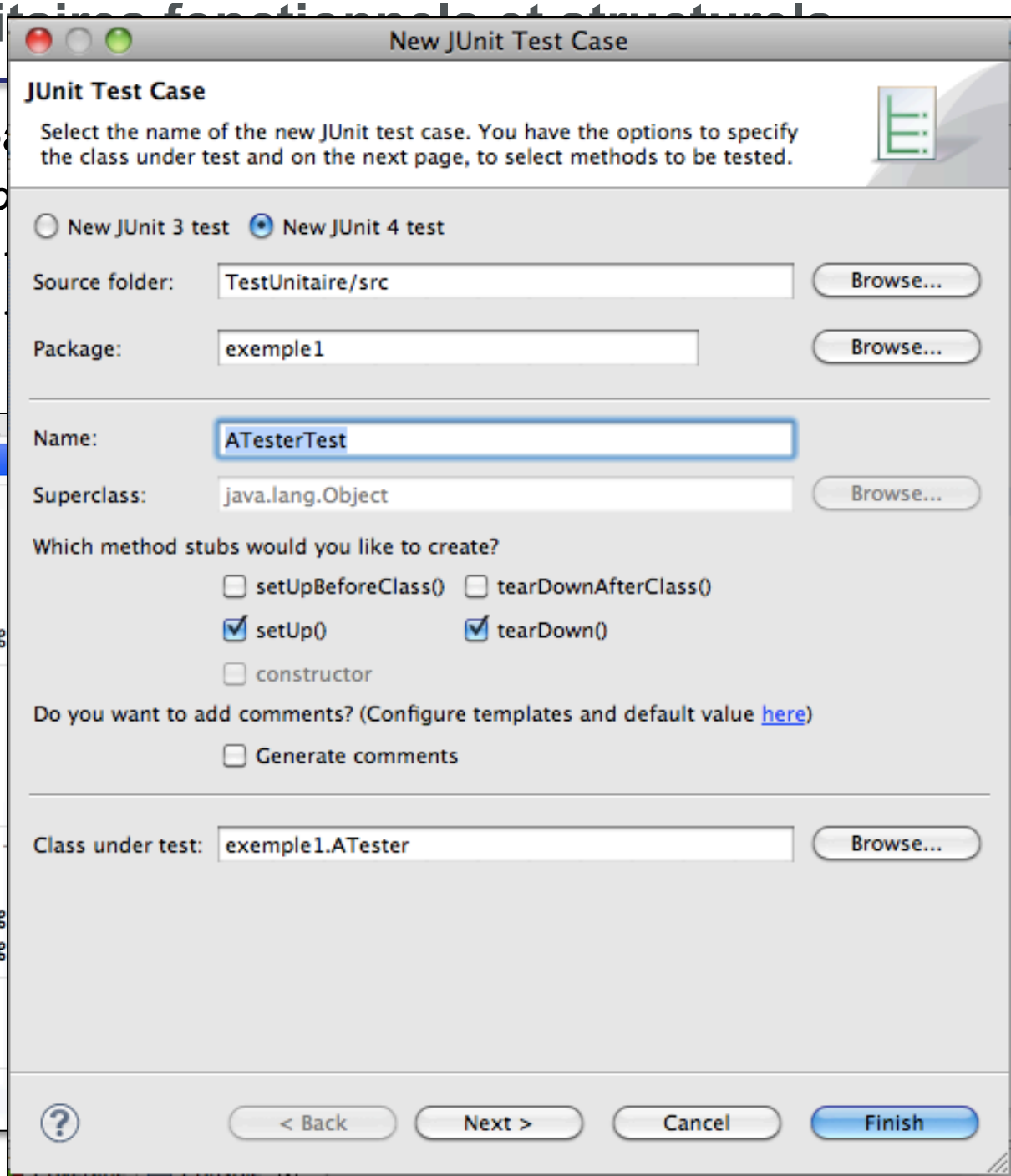


Tests unitaires fonctionnels et structurels

- On commence pa
  - *La fonction do*
  - Test 1 - DT :
  - Test 2 - DT :

Unit





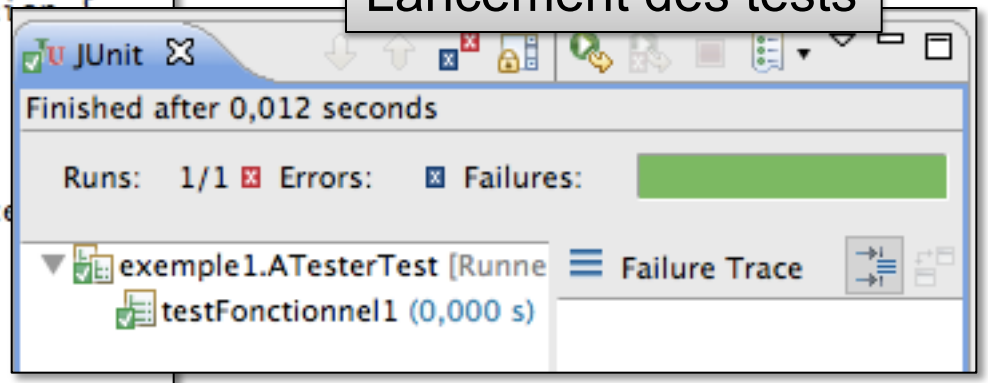
Tests unitaires fonctionnels et structurels

- On commence par le test fonctionnel
  - *La fonction doit retourner la somme de 2 entiers modulo 20000*
  - Test 1 - DT : {x=10, y=20}, prédiction de l'oracle : 30
  - Test 2 - DT : {x=10000, y=10050}, prédiction de l'oracle : 50

Ecriture du test 1

```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.junit.Assert;

public class ATesterTest {

    ATester a;
    @Before
    public void setUp() throws Exception {
        a = new ATester();
    }

    @After
    public void tearDown() throws Exce
    }

    @Test
    public void testFonctionnel1(){
        int r = a.f(10, 20);
        Assert.assertEquals(30, r);
    }
}
```

Lancement des tests



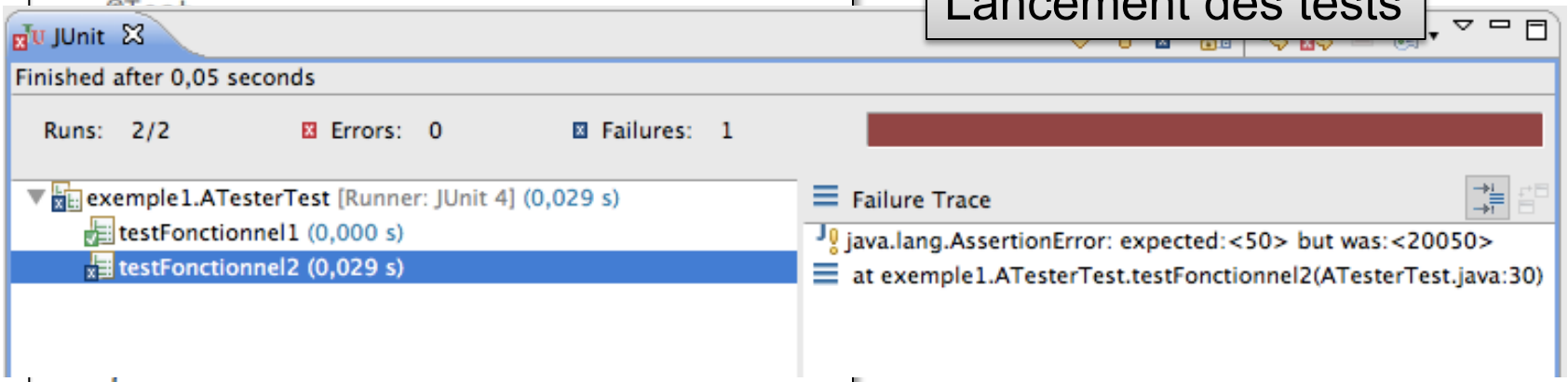
JUnit

- On commence par le test fonctionnel
  - *La fonction doit retourner la somme de 2 entiers modulo 20000*
  - Test 1 - DT : {x=10, y=20}, prédiction de l'oracle : 30
  - Test 2 - DT : {x=10000, y=10050}, prédiction de l'oracle : 50

Ecriture du test 2

```
public class ATesterTest {

    ATester a;
    @Before
    public void setUp() throws Exception {
        a = new ATester();
    }

    @After
    public void tearDown() throws Exception {
    }

    }
}
```

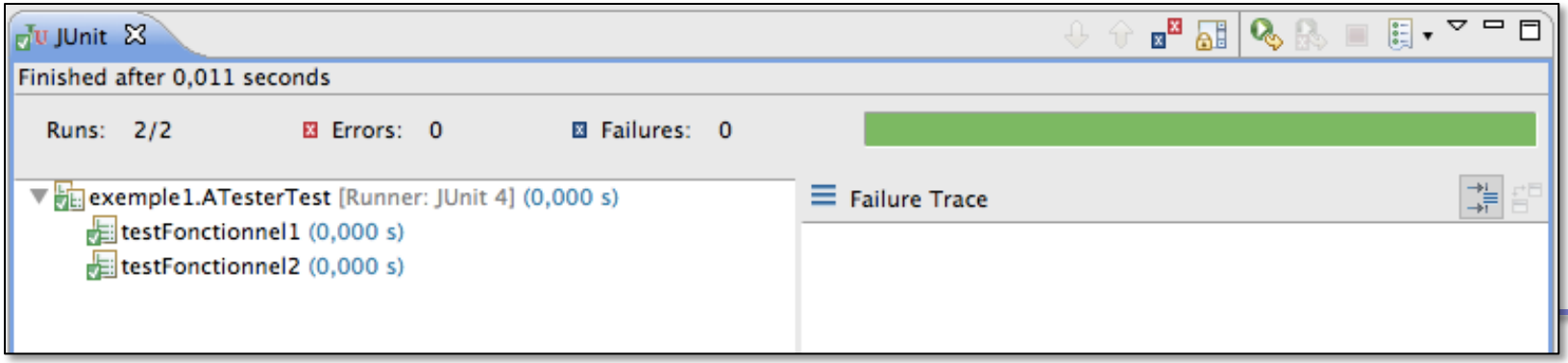
Lancement des tests



Tests unitaires fonctionnels et structurels

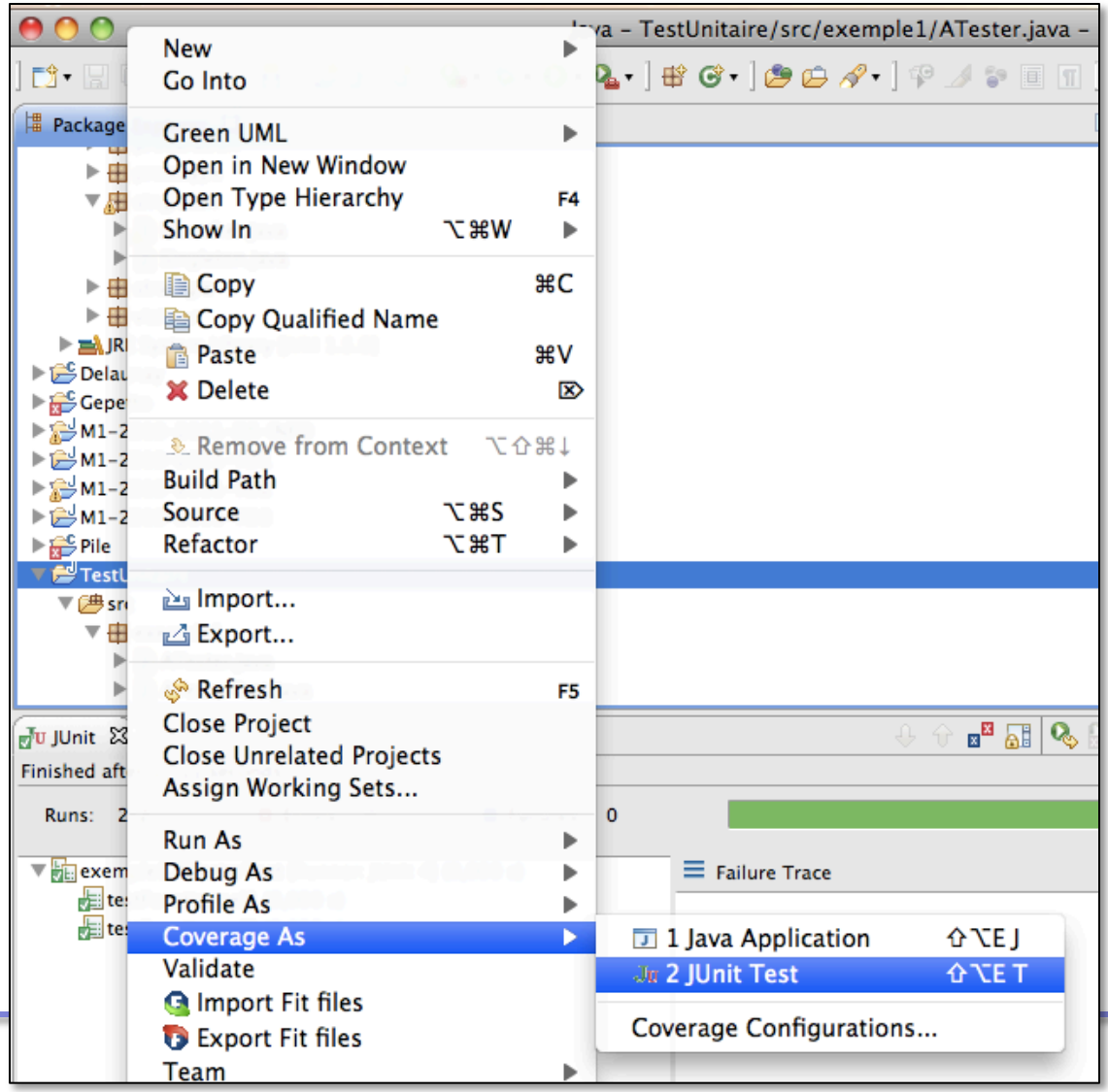
- On commence par le test fonctionnel
  - *La fonction doit retourner la somme de 2 entiers modulo 20000*
  - Test 1 - DT : {x=10, y=20}, prédiction de l'oracle : 30
  - Test 2 - DT : {x=10000, y=10050}, prédiction de l'oracle : 5
- Correction du code et on relance le test

```
public class ATester {
    public int f(int x, int y){
        if(x==500 && y==600)
            return y-x;
        else
            return (x+y)%20000;
    }
}
```



EclEmma

- Et maintenant le test structurel
  - Lancement d'une première couverture de code basée sur les tests



Tests unitaires fonctionnels et structurels

EclEmma

- Et maintenant le test structurel
  - Lancement d'une première couverture de code basée sur les tests

```
public class ATester {
    public int f(int x, int y){
        if(x==500 && y==600)
            return y-x;
        else
            return (x+y)%20000;
    }
}
```

Tout le code n'est pas couvert !!

En vert, ligne couverte,
En rouge, ligne non couverte,
En jaune, ligne partiellement couverte

EclEmma fait de la couverture de nœuds en splittant les conditions multiples

Coverage | Console

ATesterTest (19 sept. 2011 22:31:13)

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
TestUnitaire	86,3 %	44	7	51
src	86,3 %	44	7	51
exemple1	63,2 %	12	7	19
ATester.java	63,2 %	12	7	19
ATester	63,2 %	12	7	19
f(int, int)	56,2 %	9	7	16
exemple1.test	100,0 %	32	0	32
ATesterTest.java	100,0 %	32	0	32

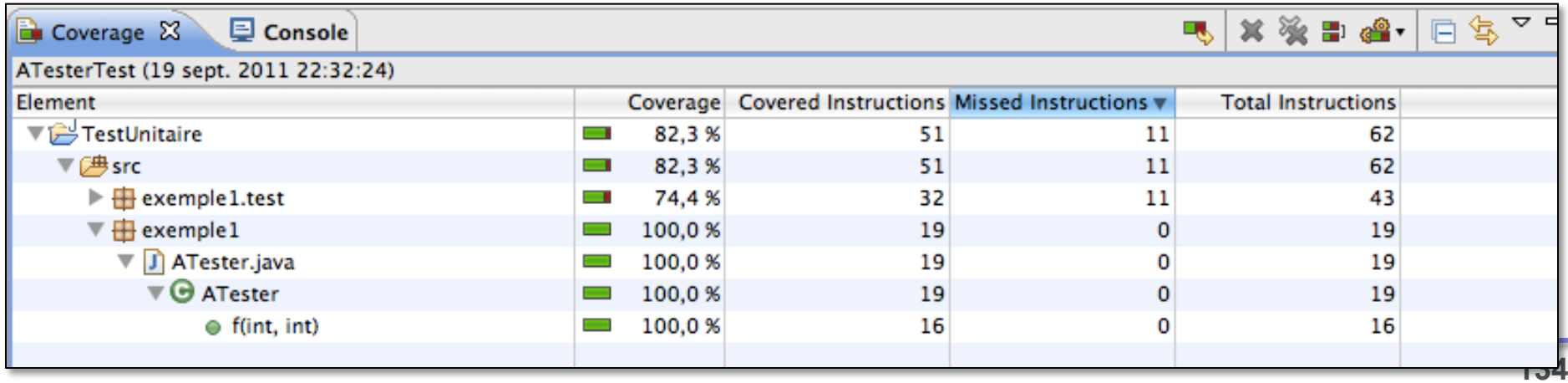
Tests unitaires fonctionnels et structurels

- Et maintenant le test structurel
  - Lancement d'une première couverture de code basée sur les tests
  - Ajout d'un test unitaire

```
@Test
public void testStructurel1(){
    int r = a.f(500, 600);
    Assert.assertEquals(1100, r);
}
```

```
public class ATester {
    public int f(int x, int y){
        if(x==500 && y==600)
            return y-x;
        else
            return (x+y)%20000;
    }
}
```

Couverture complète atteinte



Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
TestUnitaire	82,3 %	51	11	62
src	82,3 %	51	11	62
exemple1.test	74,4 %	32	11	43
exemple1	100,0 %	19	0	19
ATester.java	100,0 %	19	0	19
ATester	100,0 %	19	0	19
f(int, int)	100,0 %	16	0	16

Couverture partielle de blocs avec EclEmma

EclEmma

- Une ligne est partiellement couverte car l'évaluation du && est fainéante
  - x vaut 1 dans le test
  - Donc x==2 est évalué mais pas y==1

```
public class A {

    public boolean testAND(int x, int y){
        if (x==2 && y==1)
            return true;
        else
            return false;
    }
}
```

```
@Test
public void testStructurel3() {
    Assert.assertEquals(false, a.testAND(1,2));

}
```

Couverture partielle de blocs avec EclEmma

EclEmma

- Avec x=2, le test y==1 est effectué

```
public class A {

    public boolean testAND(int x, int y){
        if (x==2 && y==1)
            return true;
        else
            return false;
    }
}
```

```
@Test
public void testStructurel3() {
    Assert.assertEquals(false, a.testAND(2,3));

}
```

Couverture partielle de blocs avec EclEmma

- Il en va de même avec l'évaluation de l'opérateur ||

EclEmma

```
@Test
public void testStructurel4() {
    Assert.assertEquals(true, a.testOR(2,1));

}
```

```
@Test
public void testStructurel4() {
    Assert.assertEquals(true, a.testOR(1,2));

}
```

```
public boolean testOR(int x, int y){
    if (x==2 || y==1)
        return true;
    else
        return false;
}
```

```
public boolean testOR(int x, int y){
    if (x==2 || y==1)
        return true;
    else
        return false;
}
```

Couverture partielle de blocs avec EclEmma

EclEmma

- La ligne 6 est un `if…then…else`,
  - `vi` valant 1 seule une branche de la condition est couverte
- Ceci implique pour la ligne 8, que
  - `vk<vj` est évalué à faux pour `vk=0`
  - Donc `++vk` n'est jamais exécuté, d'ou une couverture partielle aussi

```
 1 public class MyClass
 2 {
 3     public static void main (final String [] args)
 4     {
 5         int vi = 1;
 6         int vj = vi > 0 ? -1 : 1;
 7
 8         for (int vk = 0; vk < vj; ++ vk)
 9         {
10             System.out.println ("vk = " + vk);
11         }
12     }
13
14     public MyClass () {}
15 }
```

Couverture partielle de blocs avec EclEmma

- L'attribut `m_i` est initialisé lors de l'appel au constructeur de la classe
  - 2 constructeurs (défaut ou argument entier)
  - Seule le constructeur avec une valeur entière exécutée
  - Donc couverture partielle de l'initialisation de `m_i`

```
 1 public class MyClass
 2 {
 3     public static void main (final String [] args)
 4     {
 5         MyClass mc = new MyClass (5);
 6     }
 7
 8     public MyClass ()
 9     {
10     }
11
12     public MyClass (int size)
13     {
14         m_i = new Integer (size);
15     }
16
17     private Integer m_i = new Integer (0);
18 }
```

Couverture partielle de blocs avec EclEmma

EclEmma

- Avec un bloc `finally`
  - Le compilateur vérifie que le nettoyage du code des lignes 16-25 est bien exécuté (quelque soit le résultat du `try`)
  - Donc que le code …

```
 9          InputStream in = null;
10          try
11          {
12              in = new FileInputStream (args [0]);
13          }
14          finally
15          {
16              if (in != null)
17              {
18                  try
19                  {
20                      in.close ();
21                  }
22                  catch (Exception ignore)
23                  {
24                  }
25              }
26          }
27      }
28
29      public MyClass () {}
30  }
```

Plan du cours

- Définitions autour du test
  - Métriques de qualité/fiabilité logicielle
  - Positionnement du test dans le cycle de vie du logiciel
  - Test et méthodes agiles
  - Qu'est-ce que le test ?
- Processus de test simplifié
- Comment choisir les scénarios de test à jouer ?
  - Les méthodes de test Boîte Noire
  - Les méthodes de test Boîte Blanche
- Tests unitaires avec JUnit et EclEmma
  - Présentation de JUnit
  - Couverture de test avec EclEmma
- Utilisation de doublures (mocks) pour les tests unitaires avant intégration

Mocks, doublures ou simulacres

- Les frameworks de mock (Jmock, Mockito, EasyMock, MockRunner, etc.) fournissent une solution à des questions comme :
  - Comment faire si pour un test, vous avez besoin d'un objet qui n'est pas encore codé ou géré par un autre développeur qui ne l'a pas intégré ?

    → Dans le cas du test d'un composant avant son intégration
  - Comment faire si un test doit faire appel à une base de donnée et que la connexion est lente ou que la BD est absente, le disque dur saturé ?

    → dans le cas du test avant test système

- Un mock ou doublure ou simulacre permet de
  - Renvoyer des résultats prédéterminés ;
  - D'obtenir des états particuliers du contexte ;
  - D'invoquer des ressources qui prennent du temps ;
  - De simuler/doubler un objet au comportement spécifié mais pas implémenté.

Mocks, doublures ou simulacres

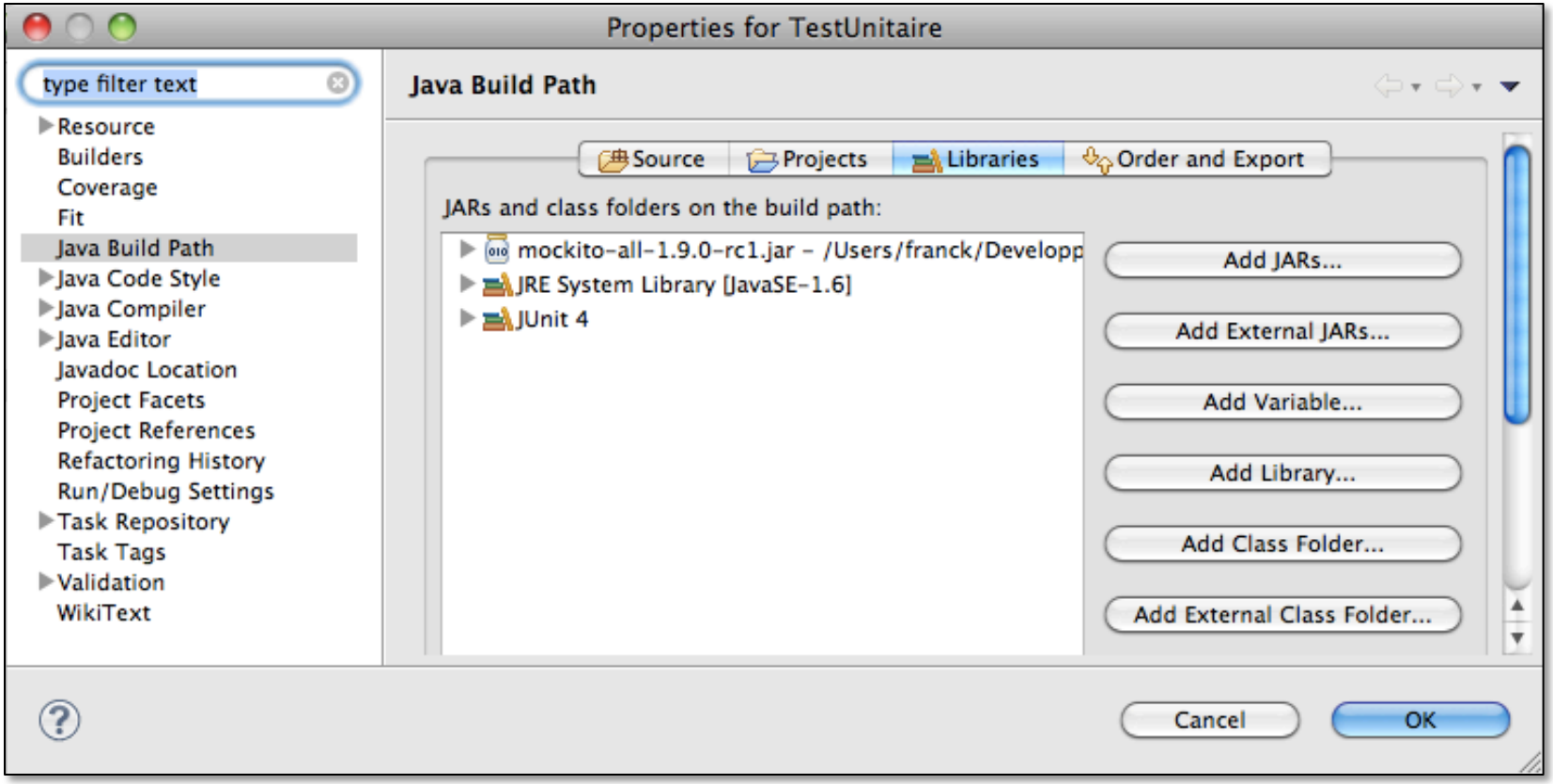
- Dans le paradigme objet, on distingue différents types d'objets/concepts pour simuler le comportement d'un autre objet :
  - **Le dummy ou fantôme, bouffon**
    - objet "vide" qui n'a pas de fonctionnalités implémentées
  - **Le stub ou bouchon**
    - classe qui retourne en dur une valeur pour une méthode invoquée
  - **Le fake ou substitut, simulateur**
    - classe qui est une implémentation partielle et qui retourne toujours les mêmes réponses selon les paramètres fournis
  - **Le spy ou espion**
    - classe qui vérifie l'utilisation qui en est faite après l'exécution
  - **Le mock ou doublure, simulacre**
    - classe qui agit à la fois comme un stub et un spy

Les principes

- Mockito est un générateur automatique de doublures/simulacres
- Il fonctionne sur le mode de l'espion :
  - On crée les mocks
  - On décrit leur comportement (mode fake ou substitut) ;
  - Ensuite, à l'exécution, toutes les interactions avec les mocks sont mémorisées ;
  - A la fin du test, on interroge les mocks pour savoir comment ils ont été utilisés

Installation et utilisation de Mockito

- Récupérer la dernière release mockito-all-1.9.0-rc1.jar
- Placer cette archive dans un répertoire pour vos développements
- importer-le dans le build path de votre projet



Exemple d'un jeu de plateau

Mockito

- Vous développez la classe `GameBoard`
- Un autre développeur se charge de la classe `Cell`
  - Interface bien définie (très important pour le travail par équipe)
- Vous voulez faire des tests unitaires des opérations `add()`, `getNbCells()` et `getCell()` de la classe `GameBoard`

```
public interface Cell {
    public String getName();

    public GameBoard getGameBoard();
}
```


```
public class GameBoard {

    private ArrayList<Cell> cells;
    public GameBoard() {
        cells = new ArrayList<Cell>();
    }

    public void add(Cell c){
        cells.add(c);
    }

    public int getNbCells(){
        return cells.size();
    }

    public Cell getCell(int i){
        Cell c=null;
        if (i>=0 && i<cells.size())
            c = cells.get(i);
        return c;
    }
}
```

Exemple d'un jeu de plateau

Mockito

- On va créer un objet doublure de notre interface

Import des fonctionnalités de mocking

```
public interface Cell {
    public String getName();

    public GameBoard getGameBoard();
}
```

Création d'une doublure de cellule

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

import exempleMockito.GameBoard;
import exempleMockito.Cell;

public class TestGameBoard {

    GameBoard gb;
    @Before
    public void setUp() throws Exception {
        gb = new GameBoard();
    }

    @Test
    public void testAdd(){
        Cell c = mock(Cell.class);
        gb.add(c);
        Assert.assertEquals(1,gb.getNbCells());
    }

}
```

Exemple d'un jeu de plateau

Mockito

- On va créer un objet doublure de notre interface
- Puis lui attribuer des comportements en cas d'appel aux opérations

```
public interface Cell {
    public String getName();

    public GameBoard getGameBoard();
}
```

Création d'une doublure de cellule

Spécification de la réponse en cas d'appel à l'opération `getName()` sur l'objet `c`

```
@Test
public void testGetCell(){
    Cell c = mock(Cell.class);
    when(c.getName()).thenReturn("case 1");
    gb.add(c);

    Cell c2 = gb.getCell(0);
    Assert.assertEquals("case 1",c2.getName());
}
```

Les principes

- On importe le package qui va bien

```
import static org.mockito.Mockito.*;
```

- On crée les doublures à l'aide de la méthode `mock`

```
MonItf doublure = mock(MonItf.class);
```

- On décrit le comportement attendu de la doublure avec la méthode `when`
  - par exemple

```
when(doublure.monOpe()).thenReturn(12);
```

- On crée l'objet de la classe à tester en utilisant les doublures, on décrit le corps du test

- On vérifie que l'interaction avec les doublures est correcte par la méthode `verify`.

Fonctionnement de la méthode mock

```
public static <T> T
mock(java.lang.Class<T> classToMock)
```

- Cette méthode permet de créer une doublure
  - Pour une interface
  - Comme pour une classe

- En théorie, en test, vous ne devez pas faire des doublure de classes mais seulement d'interfaces qui décrivent les services des des objets collaborant avec l'objet à tester

- Nommage du mock pour des messages d'erreurs plus clairs

```
public static <T> T
mock(java.lang.Class<T> classToMock, String name)
```

Fonctionnement de la méthode mock – Comportement par défaut

- On suppose donnée une interface `Itf` contenant les méthodes
  - `exempleInt`
  - `exempleBool`
  - `exempleList`

  avec les types de retour correspondant.

```
Itf mock = mock(Itf.class, «Itf»)
```

- Dès l'exécution de cet appel, lobjet `mockItf` est créé avec des **comportements par défaut** :

```
assertEquals("Itf", mock.toString());
assertEquals("type int : 0 ", 0,mock.exempleInt());
assertEquals("type bool : false",false,
                                    mock.exempleBool());
 assertEquals("collection : vide",0,
                            mock.exempleList().size());
```

- Pour définir le comportement de son choix (**stubbing**), on utilise la méthode `when` associée aux fonctions de stubbing :
  - `thenReturn(T value);`
  - `thenReturn(T value1, ...,T valuen);`
  - `thenThrow(Throwable t);`
  - `thenThrow(Throwable t1, ..., Throwable tn)`

```
when(mock.exempleInt()).thenReturn(3);
```

- Utilisation avec Junit
  - `assertEquals("une fois 3",3, mock.exempleInt());`
  - `assertEquals("deux fois 3",3, mock.exempleInt());`

Retourne la valeur stubbée autant de fois que nécessaire.

Valeurs de retour consécutives

```
when(mock.exempleInt()).thenReturn(3, 4, 5);
```

- Utilisation avec JUnit
  - `assertEquals("une fois : 3",3, mock.exempleInt());`
  - `assertEquals("deux fois : 4",4, mock.exempleInt());`
  - `assertEquals("trois fois: 5",5, mock.exempleInt());`

- `when(mock.exempleInt()).thenReturn(3, 4);`

  est un raccourci pour

  `when(mock.exempleInt()).thenReturn(3).thenReturn(4);`

Fonctionnement différent suivant la valeur des paramètres

```
public int exempleIntBis(int i, int j);
```

- On écrira par exemple
  - `when(mock.exempleIntBis(4,2)).thenReturn(4);`
  - `when(mock.exempleIntBis(5,3)).thenReturn(5);`
- Utilisation avec JUnit
  - `assertEquals("param 4 2:4" , 4, mock.exempleIntBis(4,2));`
  - `assertEquals("param 5 3 : 5", 5, mock.exempleIntBis(5,3));`

Fonctionnement avec levée d'exceptions et void

- Attention, la méthode when ne permet pas de stubber des méthodes de type void

```
public void opVoidThrowExc() throws ExeException;
```

- On écrira

```
doThrow(new ExeException()).when(mock).
                                opVoidThrowExc();
```

- Utilisation avec JUnit

```
try {
    mock.opVoidThrowExc();
    fail();
} catch (ExeException e) {
    assertTrue("levee exception", true);
}
```

- La méthode `exempleBool` doit avoir été appelée. . .
  - exactement une fois :

```
verify(mock).exempleBool();
verify(mock,times(1)).exempleBool(); //équivalent
```

  - Au moins / au plus une fois:

```
verify(mock, atLeastOnce()).exempleBool();
verify(mock, atMost(1)).exempleBool();
```

  - Jamais (ce qui est faux) :

```
verify(mock, never()).exempleBool();
```

- Avec des paramètres

```
verify(mock).exempleIntBis(4, 2);
```

- Ordre des appels

```
import org.mockito.InOrder;
```

- Pour vérifier que l'appel (4,2) est effectué avant l'appel (5,3)

```
InOrder inOrder = inOrder(mock);
inOrder.verify(mock).methodeIntBis(4, 2);
inOrder.verify(mock).methodeIntBis(5, 3);
```

- Marche aussi avec plusieurs mocks :

```
InOrder inOrder = inOrder(mock1, mock2);
inOrder.verify(mock1).foo();
inOrder.verify(mock2).yo();
```

- Vérification qu'il n'y a pas d'interaction :
  - `verifyNoMoreInteractions(mock);`
  - `verifyZeroInteractions(mock);`
- Possibilité de décrire des contraintes sur les paramètres
  - `ArgumentMatcher`

Pour conclure

- Le test de logiciels est une activité très coûteuse qui gagne à être supportée par des outils.

- Les principales catégories d'outils concernent :
  - L'exécution des tests
  - La gestion des campagnes
  - Le test de performance
  - La génération de tests fonctionnels
  - La génération de tests structurels

- N'oubliez pas l'importance du test !!!