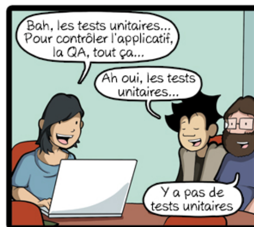
3. Méthodes et Outils de tests

Laëtitia Matignon
laetitia.matignon@univ-lyon1.fr

Département Informatique - Polytech Lyon
Université Claude Bernard Lyon 1
2016 - 2017

1. Introduction
2. Test logiciel
3. Méthodes de test
4. Outils pour le test unitaire



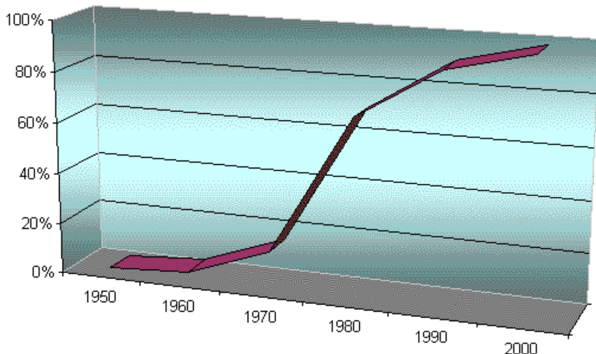
CommitStrip.com

Plan

1. Introduction
2. Test logiciel
3. Méthodes de test
4. Outils pour le test unitaire

La crise du logiciel

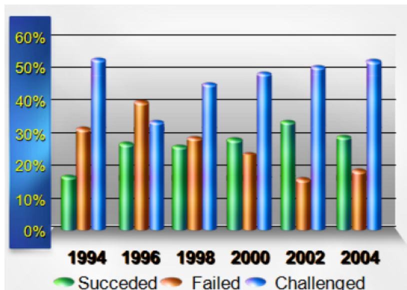
- L'aspect matériel représente 20% du coût global
- L'aspect logiciel représente 80% du coût global



Le coût du logiciel dépasse le coût du matériel, baisse significative de la qualité des logiciels

La crise du logiciel

- Etude du Standish Group sur plus de 350 entreprises totalisant plus de 8000 projets d'application :

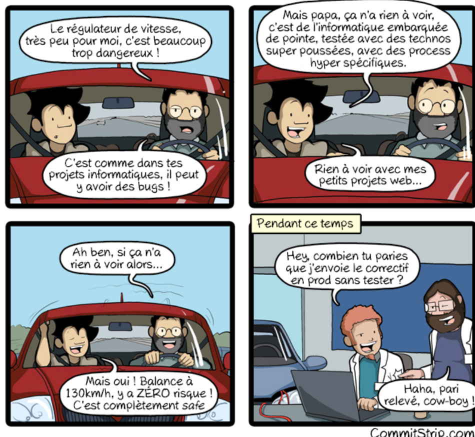


- Succès : livré à temps, sans dépassement de budget et avec toutes les fonctionnalités initialement spécifiées
- Mitigé : livré et opérationnel, mais avec moins de fonctionnalités que prévu et un dépassement de budget et/ou de temps
- Echec : abandonné en cours de route

Exemples d'échecs

- Exemple d'abandons de projets :
  - Confirm (1992) : projet d'American Airlines de système de réservation commun (avions, voitures, hôtels, etc.).
    Investissement : 125 Millions $ sur 4 ans, plus de 200 ingénieurs.
    Résultat : les différentes parties n'ont pas pu être assemblées en raison de ≪ **la description incomplète des besoins**, un manque d'implication des utilisateurs et une **constante évolution des exigences et des spécifications** ≫ [THE STANDISH GROUP REPORT, 1995].

Exemples d'échecs



- Importance des tests : Ariane V (1996) : explosion de la fusée en vol due à bug logiciel (erreur de débordement lors de la conversion d'un nombre flottant 64bits vers un entier 16bits. Code hérité de Ariane IV, manque de tests). Coût : 500 Millions de $.

Exemples d'échecs

- Importance des tests :
  - Perte de NASA Mars Climate Orbiter, une des deux sondes spatiales du programme Mars Surveyor 98 (1999) : détruite à cause d'une erreur de navigation pendant sa mise en orbite autour de Mars (certains paramètres avaient été calculés en unités de mesure anglo-saxonnes et transmis tels quels à l'équipe de navigation, qui attendait ces données en unités du système métrique)

Principales causes de la crise du logiciel

- Fuite en avant de la complexité (technologie en perpétuelle évolution, croissance taille et complexité des systèmes)
- Faible correspondance avec les besoins des utilisateurs & Coût élevé du changement des fonctionnalités
- Délais de réalisation généralement dépassés
- Faiblesse des tests
- L'importance de la maintenance est souvent sous-estimée

On maîtrise mal le développement des logiciels.

Définition du génie logiciel

Génie logiciel (software engineering)

Ensemble de moyens (techniques et méthodes) mis en oeuvre pour la construction de systèmes informatiques et de logiciels.
Objectif : maitrise de l'activité de fabrication des logiciels (temps de développement, critères de qualité).

Nécessité de méthodes/processus de développement logiciel.

Un processus de développement / cycle de vie / méthode définit un ensemble d'**activités** et leurs enchaînements pour la réalisation d'un logiciel

- analyse des besoins / spécification (cahier des charges, maquettes, ...)
- conception, implémentation, intégration, tests
- déploiement, maintenance

Modèle *code-and-fix*

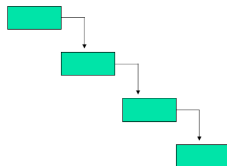
- ≪ on code d'abord et on modifie ensuite ≫
  - Développement ≪ sauvage ≫
  - Analyse courte et priorité au codage
  - Votre dernier TP ?
- Modèle primitif (< 1970)
- Inadapté aux développements en équipe ou de grande taille

Cycle de vie du logiciel

Processus de développement linéaires

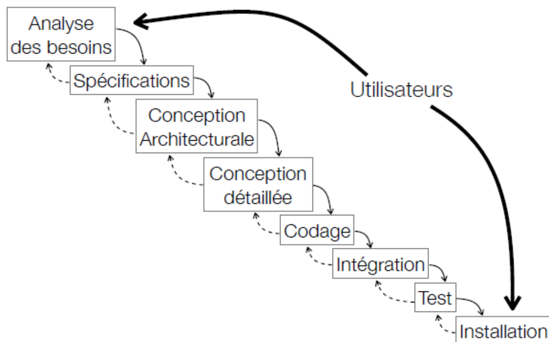
Considérer le développement logiciel comme une succession d'étapes réalisées de façon linéaire.

- Chaque étape correspond à une activité de base
- Chaque étape doit être validée
- Il n'y a pas (ou peu) de retours en arrière



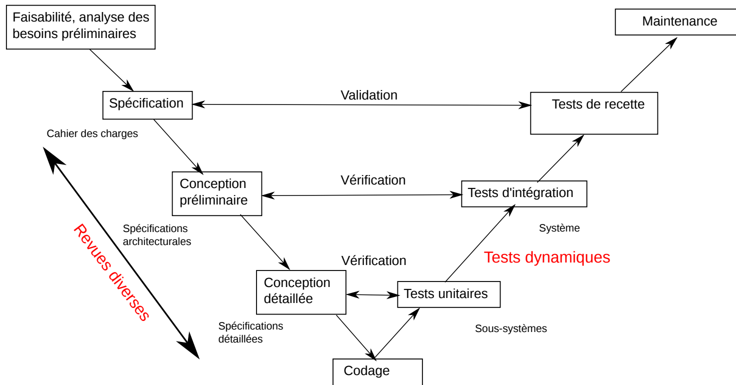
Quels processus de développement linéaires connaissez-vous ?

Cycle de vie en cascade (waterfall model) 1970



- Axé sur la documentation, Pas (ou peu) de retours en arrière
- Contre-exemple classique
- Echecs majeurs sur de gros systèmes :
  - planification trop rigide
  - le logiciel n'est délivré qu'à la fin du projet
  - tests de l'application globale uniquement à la fin
  - difficulté de définir tous les besoins au début du projet

Cycle de vie en V



- Variante du modèle en cascade, mise en évidence du besoin d'anticiper
- Tests définis à l'issue de chaque phase

Validation et Vérification

V & V

Activités exécutées en parallèle du développement d'un système logiciel :

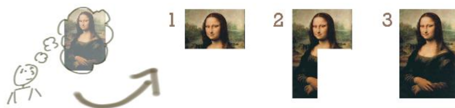
- Validation : Est-ce que le logiciel réalise les fonctions attendues ? - adaptation vis à vis des besoins des utilisateurs, réponse aux attentes, concerne les utilisateurs.
- Vérification : Est-ce que le logiciel fonctionne correctement ? - correction interne du logiciel, absence de défauts, concerne les développeurs

Cycle de vie du logiciel

Processus de développement anti-cascade

Le changement est une constante des projets logiciels

- Diviser le projet en incréments, définis a priori, sous partie fonctionnelle cohérente du produit final



- Procéder par itération : chaque itération affine la réalisation

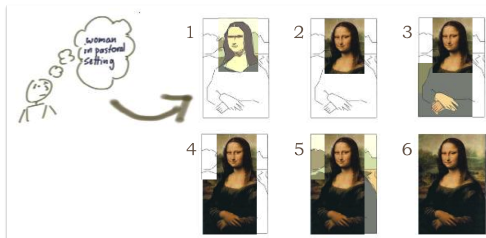
crédit image http://www.agileproductdesign.com/blog/dont_know_what_i_want.html

Cycle de vie du logiciel

Processus de développement anti-cascade

Le changement est une constante des projets logiciels

Développement itératif et incrémental :

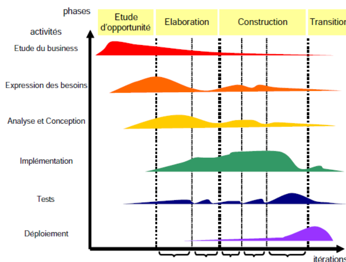


- Un sprint = une itération produit un incrément
- incrémental : chaque incrément est testé et ajoute de nouvelles fonctionnalités
- itératif : chaque incrément affine les fonctionnalités existantes

crédit image `http://itsadeliverything.com/revisiting-the-iterative-incremental-mona-lisa`

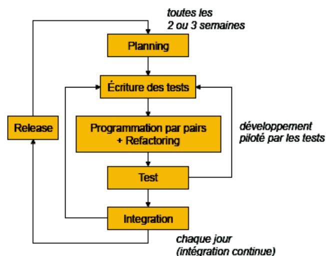
Processus de développement itératifs

- Exemples de méthodes :
  - Cycle en b (N.D. Birrel et M.A. Ould, 1985)
  - Cycle en spirale (B. Boehm, 1988)
  - Cycle en O (P. Kruchten, 1991)
  - Processus unifié (J. Rumbaugh, I. Jacobson et G. Booch, 1999)
  - Méthodes agiles



Source Rational Software Corporation

UP



XP : Amélioration continue (refactoring), Tests unitaires & TDD, Programmation par pairs, ...

Méthodes agiles

Quelles activités pouvons nous abandonner tout en produisant des logiciels de qualité ?

Comment mieux travailler avec le client pour nous focaliser sur ses besoins les plus prioritaires et être aussi réactifs que possible ?

- Création de l'Agile Alliance et de son manifeste en 2001 : 4 valeurs et 12 principes (http ://agilealliance.org)
- Itérations très courte (2 semaines à 2 mois)
- Liens forts avec le client
- S'opposent à la procédure et à la spécification à outrance
- Adaptation aux changements fréquents
- Promotion de l'auto-organisation versus la hiérarchie verticale

Conclusion sur processus de développement logiciel

- Pas de modèle idéal : la meilleure méthode est celle adaptée au contexte
  - Type de logiciel
  - Ampleur du projet
  - Equipe de développement
- Références sur les méthodes agiles
  - http://www.agileproductdesign.com/index.html
  - http://agilarium.wikispaces.com/
  - Vidéo retour d'expérience de 2 ans de travail avec le mélange des différentes déclinaisons de l'agilité : UP, XP, Kanban et Scrum, en particulier au sein des Laboratoires Boiron : http://clacote.free.fr/agilite.html

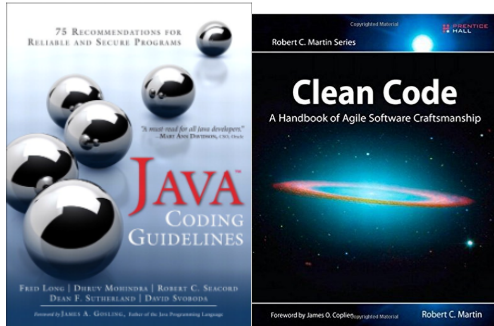
Conclusion sur processus de développement logiciel

- Pas de modèle idéal : la meilleure méthode est celle adaptée au contexte
  - Type de logiciel
  - Ampleur du projet
  - Equipe de développement
- Références sur les méthodes agiles
  - `http://www.agileproductdesign.com/index.html`
  - `http://agilarium.wikispaces.com/`
  - `http://clacote.free.fr/agilite.html`
- Importance des tests : cf. suite du cours
- Importance des outils autour du développement logiciel : cf. transparent suivant

Outils autour du développement logiciel

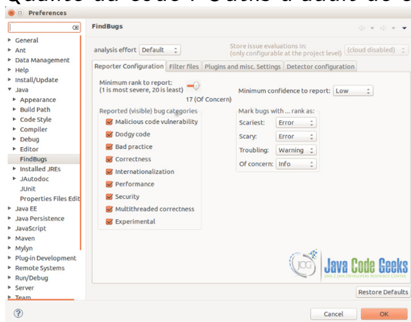
Au-delà du code :

- Documentation : Générateurs de documentation
- Partage des sources : Gestionnaire de versions
- Suivi de bugs / évolution : Gestionnaire de tickets
- Qualité du code : Outils d'audit de code
  - livres du CERT Centre gouvernemental de veille, d'alerte et de réponse aux attaques informatiques `https://www.securecoding.cert.org/confluence/display/java/SEI+CERT+Oracle+Coding+Standard+for+Java`



Outils autour du développement logiciel

- Documentation : Générateurs de documentation
- Partage des sources : Gestionnaire de versions
- Suivi de bugs / évolution : Gestionnaire de tickets
- Qualité du code : Outils d'audit de code

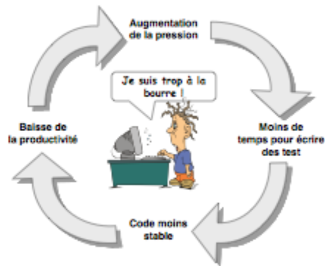


- Analyseur statique de code
- http://findbugs.sourceforge.net/
- 4 niveaux de bugs (Scariest,Scary,Troubling,Of concern), confidence, catégories de bugs

Plan

1. Introduction
2. Test logiciel
3. Méthodes de test
4. Outils pour le test unitaire

Problème du test



Modalités de test

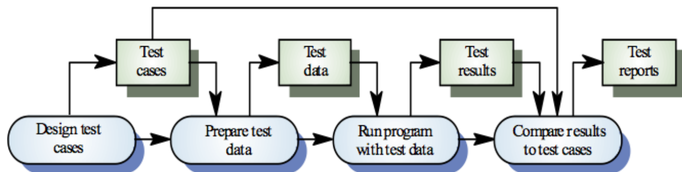
- Test statique : test “par l’humain” avec revue de code, de spécifications, de documents, ...
- Test dynamique : test par l’exécution du système pour s’assurer d’un fonctionnement correct

Actuellement, le test dynamique est la méthode la plus utilisée et représente jusqu’à 60% de l’effort complet de développement d’un produit logiciel.

Définitions

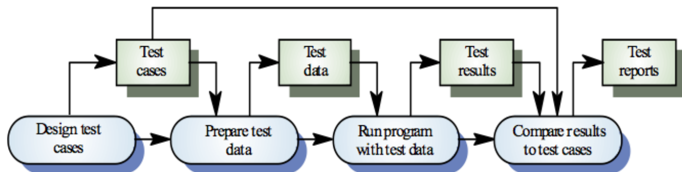
- *Tester, c'est exécuter le programme dans l'intention d'y trouver des anomalies ou des défauts* - G. Myers (The Art of Software testing)
- *Le test est l'exécution ou l'évaluation d'un système ou d'un composant par des moyens automatiques ou manuels, pour vérifier qu'il répond à ses spécifications ou identifier les différences entre les résultats attendus et les résultats obtenus* - IEEE
  → notion d'Oracle : résultats attendus d'une exécution du logiciel
- *Testing can reveal the presence of errors but never their absence* – Edsgar W. Dijkstra. *Notes on structured programming*. Academic Press, 1972.
  → test exhaustif impossible à réaliser, le test est une méthode de vérification partielle

L'activité de test



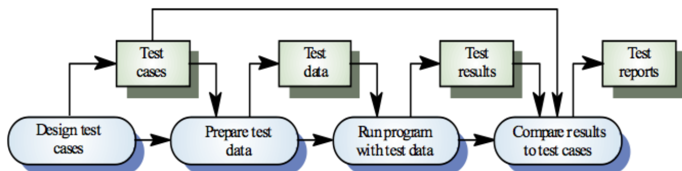
1. Identification des scénarios à tester
    - définir les actions à effectuer ou procédure à suivre pour réaliser un test

L'activité de test



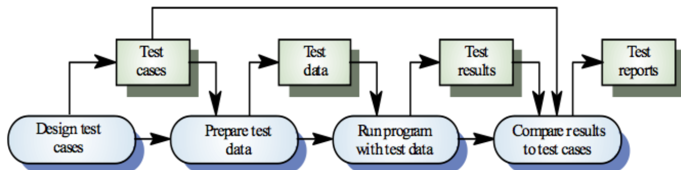
1. Identification des scénarios à tester
2. Détermination des oracles de chaque scénario
   - Le scénario de test produit un résultat qui doit être évalué de manière manuelle ou automatique (oracle)
   - Oracle : mécanisme permettant de décider la réussite d'un scénario de test
     - Exemple d'oracle de test pour un quicksort : implanter un tri plus simple, faire appel à une fonction de tri de la bibliothèque standard, vérifier que le tableau en sortie est trié et mêmes éléments

L'activité de test



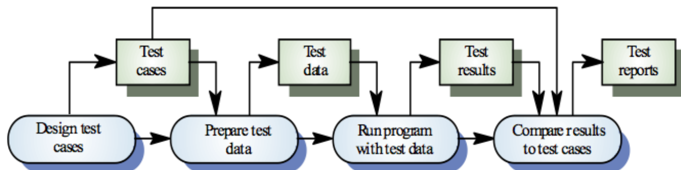
1. Identification des scénarios à tester
2. Détermination des oracles de chaque scénario
3. Génération des Données de tests (DT) de chaque scénario
   - associer à chaque entrée d'un programme une valeur choisie dans son domaine de définition afin d'exercer un scénario de test. (par exemple, $DT1 = \{a = 2, z = 4.3\}$)
     - Les données de tests (DT) doivent permettre d'avoir un échantillon représentatif de toutes les entrées possibles

L'activité de test



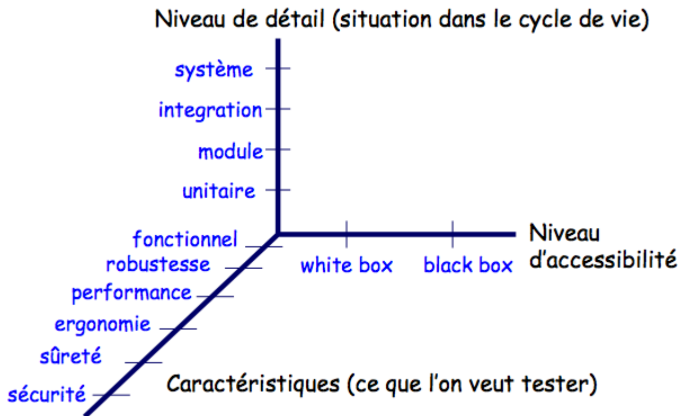
1. Identification des scénarios à tester
2. Détermination des oracles de chaque scénario
3. Génération des DT de chaque scénario
4. Execution des cas de tests
   - association d'un scénario de test, des DT le déclenchant et d'un oracle décidant de sa réussite.

L'activité de test



1. Identification des scénarios à tester
2. Détermination des oracles de chaque scénario
3. Génération des DT de chaque scénario
4. Execution des cas de tests
5. Comparaison des résultats obtenus aux oracles
6. Emission d'un rapport de test

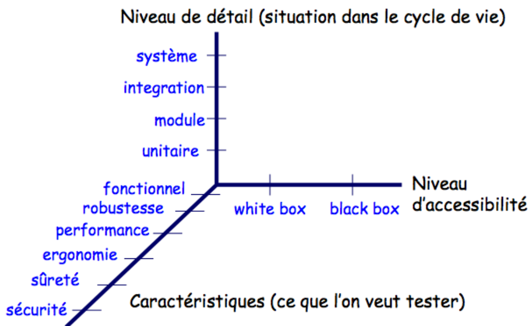
Types de tests



Types de tests : niveau de détail

- Tests unitaires : vérification des composants en isolation
- Tests d'intégration : s'assurer que les interfaces des composants sont cohérentes entre elles et que le résultat de leur intégration permet de réaliser les fonctionnalités prévues.
- Tests de validation/recette : s'assurer que le système complet, matériel et logiciel, correspond bien aux spécifications
- Tests Alpha : faire tester par le client sur le site de développement
- Tests Bêta : faire tester par le client sur le site de production
- Tests de non-régression : vérification que les corrections ou évolutions dans le code n'ont pas créées d'anomalies nouvelles

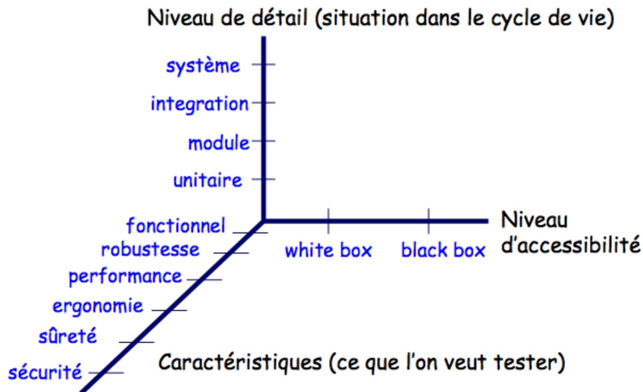
Types de tests : caractéristiques



- Test nominal ou fonctionnel : les cas de test correspondent à des données d'entrée valide (*test-to-pass*)
- Test de robustesse : les cas de test correspondent à des données d'entrée invalide (*test-to-fail*)
- Test de performance : montée en charge (*load testing*), demandes de ressources anormales (*stress testing*)

Types de tests : Niveau d'accessibilité

cf. section suivante



Plan

1. Introduction
2. Test logiciel
3. Méthodes de test
   - Méthodes de test fonctionnel
   - Méthodes de test structurel
4. Outils pour le test unitaire

Méthodes de test fonctionnel

1. Introduction

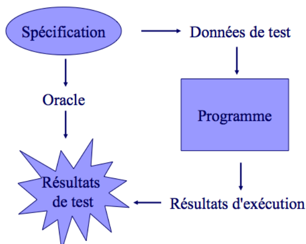
2. Test logiciel

3. Méthodes de test
   - Méthodes de test fonctionnel
   - Méthodes de test structurel

4. Outils pour le test unitaire
   - JUnit
   - Doublures d'objets

Méthodes de test fonctionnel ou boîte noire

- Cas de tests issus de la spécification du logiciel (cas d'utilisation) → à partir d'entrées définies on vérifie que le résultat final convient
- Aucune connaissance de l'implémentation
- Détection plus facile des erreurs d'omission et de spécification



Méthodes de test fonctionnel

- test exhaustif impossible à réaliser
- les DT doivent permettre d'avoir un échantillon représentatif de toutes les entrées possibles.

$\rightarrow$ comment choisir les données de tests ?

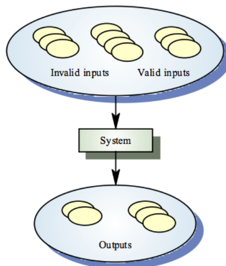
1. Approche aléatoire
2. Analyse partitionnelle des domaines des données d'entrée
3. Tests aux limites
4. Tests combinatoires - algorithmes n wise

1. Test aléatoire

- Principe : utiliser une fonction aléatoire pour sélectionner les DT sur leurs domaines de définition
- Facilement automatisable mais besoin d'un oracle valable pour tous les cas de tests
- Difficultés à produire des comportements très spécifiques
- Aucune garantie de bonne couverture de l'ensemble des entrées
- Référentiel pour les autres techniques

2. Analyse partitionnelle

- Diminuer le nombre de cas de tests par calcul de classes d'équivalence
- Une classe d'équivalence correspond à un ensemble de données de tests supposées tester le même comportement, i.e. activer le même défaut.
- Partitionner le domaine d'entrée en un nombre fini de classes d'équivalence et sélectionner au moins un test dans chaque.



2. Analyse partitionnelle : Méthode

Trois phases :

1. Pour chaque donnée d'entrée, définir des classes d'équivalence valides et invalides sur les entrées

- Si la donnée d'entrée est un intervalle, construire une classe d'équivalence valide (valeur dans intervalle) et 2 classes invalides (valeur inf / sup)
- Si la donnée est un ensemble de valeurs, construire une classe valide avec valeurs dans l'ensemble, une classe avec ensemble vide, une classe invalide avec trop de valeurs
- Si la donnée est une obligation ou une contrainte, construire une classe valide avec la contrainte respectée, une classe invalide avec la contrainte non-respectée

2. Analyse partitionnelle : exemple

1. Pour chaque donnée d'entrée, définir des classes d'équivalence valides et invalides sur les entrées

Tester un programme calculant la valeur absolue d'un entier à partir d'une entrée au clavier (donc une chaîne de caractères) supposée fournie en notation décimale.

classe	validité
chaîne vide	invalide
plusieurs mots	invalide
pas un décimal	invalide
décimal positif	valide
décimal négatif	valide

2. Analyse partitionnelle : exemple

1. Pour chaque donnée d'entrée, définir des classes d'équivalence valides et invalides sur les entrées
2. Définir un oracle par classe
3. Choisir au moins une DT par classe d'équivalence, qui sera associée à l'oracle correspondant pour obtenir un cas de test

Tester un programme calculant la valeur absolue d'un entier à partir d'une entrée au clavier (donc une chaîne de caractères) supposée fournie en notation décimale.

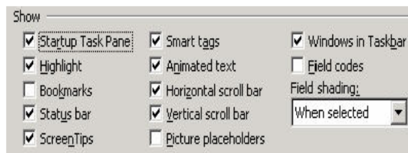
classe	validité	représentant	oracle
chaîne vide	invalide	""	échec
plusieurs mots	invalide	"1234 1234"	échec
pas un décimal	invalide	"56a"	échec
décimal positif	valide	"1234"	1234
décimal négatif	valide	"-1234"	1234

2. Analyse partitionnelle

1. Pour chaque donnée d'entrée, définir des classes d'équivalence valides et invalides sur les entrées
2. Définir un oracle par classe
3. Choisir au moins une DT par classe d'équivalence, qui sera associée à l'oracle correspondant pour obtenir un cas de test
    - technique *all single* : générer des jeux de tests utilisant au moins une fois une DT de chaque classe d'équivalence de chaque paramètre
    - technique *all pairs* : générer des jeux de tests utilisant au moins une fois chaque valeur de paire de DT de classe d'équivalence

3. Test combinatoire : approche pairwise

- Les combinaisons de valeurs de domaines d'entrée donnent lieu à une explosion combinatoire
- Exemple : Options d'une boite de dialogue



$\rightarrow 2^{12} \times 3 = 12288$

- Tester un fragment des combinaisons de valeurs qui garantissent que chaque combinaison de 2 variables est testée
- L'idée sous-jacente : la majorité des fautes sont détectées par des combinaisons de 2 valeurs de variables

3. Test combinatoire : approche pairwise

Exemple

4 variables avec 3 valeurs possibles chacune

OS	Réseau	Imprimante	Application
XP	IP	HP35	Word
Linux	Wifi	Canon900	Excel
Mac X	ATM	Canon-EX	Pwpoint

81 combinaisons, 9 paires

	OS	Réseau	Imprimante	Application
Case 1	XP	ATM	Canon-EX	Pwpoint
Case 2	Mac X	IP	HP35	Pwpoint
Case 3	Mac X	Wifi	Canon-EX	Word
Case 4	XP	IP	Canon900	Word
Case 5	XP	Wifi	HP35	Excel
Case 6	Linux	ATM	HP35	Word
Case 7	Linux	IP	Canon-EX	Excel
Case 8	Mac X	ATM	Canon900	Excel
Case 9	Linux	Wifi	Canon900	Pwpoint

Exercice : Analyse partitionnelle

Proposez une analyse partitionnelle du domaine pour les méthodes suivantes (définir les classes valides, proposer des DT) :

1. Une méthode retourne le maximum entre deux nombres.
2. Une méthode retourne vrai si on entre un nombre pair et faux sinon.
3. Une fonction calculant la valeur absolue d'un entier.
4. Une fonction calculant la distance entre deux points du plan $(d((x,y),(x',y')) = \sqrt{(x-x')^2 + (y-y')^2})$.

Exercice : Analyse partitionnelle (correction)

Proposez une analyse partitionnelle du domaine pour les méthodes suivantes (classes valides) :

1. Une méthode retourne le maximum entre deux nombres.

$$D1(a,b) = \{a > b\}; D2(a,b) = \{a < b\}; D3(a,b) = \{a == b\}$$

$$(5,2),(2,5),(5,5)$$

2. Une méthode retourne vrai si on entre un nombre pair et faux sinon.
3. Une fonction calculant la valeur absolue d'un entier.
4. Une fonction calculant la distance entre deux points du plan $(d((x,y),(x',y')) = \sqrt{(x-x')^2 + (y-y')^2})$.

Exercice : Analyse partitionnelle (correction)

Proposez une analyse partitionnelle du domaine pour les méthodes suivantes (classes valides) :

1. Une méthode retourne le maximum entre deux nombres.
2. Une méthode retourne vrai si on entre un nombre pair et faux sinon.

$$D1(d) = \{d\%2 == 0\}, D2(d) = \{d\%2 == 1\}$$

$$(2), (3)$$

3. Une fonction calculant la valeur absolue d'un entier.
4. Une fonction calculant la distance entre deux points du plan $(d((x, y), (x', y')) = \sqrt{(x - x')^2 + (y - y')^2})$.

Exercice : Analyse partitionnelle (correction)

Proposez une analyse partitionnelle du domaine pour les méthodes suivantes (classes valides) :

1. Une méthode retourne le maximum entre deux nombres.
2. Une méthode retourne vrai si on entre un nombre pair et faux sinon.
3. Une fonction calculant la valeur absolue d'un entier.

$$D1(d) = \{d < 0\}, D2(d) = \{d > 0\}, D3(d) = \{d == 0\}$$

4. Une fonction calculant la distance entre deux points du plan $(d((x, y), (x', y')) = \sqrt{(x - x')^2 + (y - y')^2})$.

Exercice : Analyse partitionnelle (correction)

Proposez une analyse partitionnelle du domaine pour les méthodes suivantes (classes valides) :

1. Une fonction calculant la distance entre deux points du plan ($d((x, y), (x', y')) = \sqrt{(x - x')^2 + (y - y')^2}$).
    1. Considérer la relation entre x et x' ($<$, $=$ ou $>$: 3 classes d'équivalences pour x) et celle entre y et y' (3 classes d'équivalences pour y) : partitionnement en $3^2 = 9$ classes représentant les combinaisons possibles.
    2. Considérer le signe de x, x', y et y' (< 0, $= 0$ ou > 0) : partitionnement en $3^4 = 81$ classes.
    3. Combinaison des deux partitionnements : 9×81 combinaisons (un certain nombre impossibles i.e. $x < x'$, $x = 0$ et $x' < 0$).

4. Tests aux limites

Solliciter les entrées se trouvant aux frontières des classes d'équivalence.

1. Calculer les classes d'équivalences
2. Pour chacune, générer une valeur médiane et une ou plusieurs valeurs aux bornes

- Pour un domaine de type intervalle : on garde les 2 valeurs des limites, et les 4 valeurs pour les limites + / - le plus petit delta possible
- Pour un ensemble ordonné de valeurs, on choisit le premier, le second, l'avant dernier et le dernier
- 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 tests pour des nombres de valeurs hors limites invalides.

Le test aux limites produit à la fois des cas de test nominaux et de robustesse

4. Tests aux limites : exemple

Solliciter les entrées se trouvant aux frontières des classes d'équivalence.

1. Calculer les classes d'équivalences
2. Pour chacune, générer une valeur médiane et une ou plusieurs valeurs aux bornes

Test aux limites d'une fonction qui attend "oui" ou "non"

classe	validité	représentant avec limites
"oui"	valide	"oui"
"non"	valide	"non"
autre	invalide	" " ou "hello" ou "ouii"

4. Tests aux limites : exemple

Solliciter les entrées se trouvant aux frontières des classes d'équivalence.

1. Calculer les classes d'équivalences
2. Pour chacune, générer une valeur médiane et une ou plusieurs valeurs aux bornes

Test aux limites d'une fonction manipulant des numéros de département

classe	validité	représentant avec limites	Large couverture
$[1; 95]$	valide	1,48,95	1,2,48,94,95
$[minInt; 1[$	invalide	-3000,0	-3000,-1,0
$]95; maxInt]$	invalide	96,1000	96,97,1000

Exercice : Tests aux limites

Proposez des tests aux limites pour une fonction qui incrémente un entier.

Exercice : Tests aux limites

Proposez des tests aux limites pour une fonction qui incrémente un entier.

- entier minimal représentable $-intMax$,
- entier $-intMax + 1$
- très grands entiers $intMax - 1$ et $intMax$
- -1, 0 et $+1$.

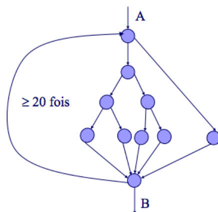
Méthodes de test structurel

1. Introduction
2. Test logiciel
3. Méthodes de test
   - Méthodes de test fonctionnel
   - Méthodes de test structurel
4. Outils pour le test unitaire
   - JUnit
   - Doublures d'objets

Niveau d'accessibilité

Approches structurelles ou test boîte blanche

- Données de test sont produites à partir d'une analyse du code
- Examen de la structure du programme (basé sur le flot de contrôle ou de données)
- Établissement des tests en fonction de critères de couverture
- Tests uniquement pour un code déjà écrit



Méthodes de test structurel

Création de cas de test sur la base de chemins d'exécution dans le code couvrant certains éléments.

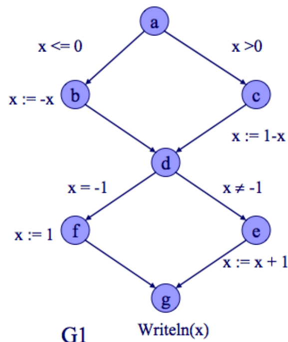
Graphe de flot de contrôle

- 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;
writeln(x)
```

- Son graph de contrôle G1 admet 4 chemins de contrôle :

$$G1 = a(b + c)d(e + f)g$$

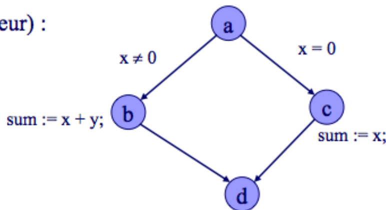


Méthodes de test structurel : couverture de *tous les noeuds*

- Les DT doivent permettre de couvrir au moins une fois chaque noeud (instruction)

Soit le programme P5 (somme avec erreur) :

```
function sum (x,y : integer) : integer;
begin
if (x = 0) then sum := x
else sum := x + y
end;
```

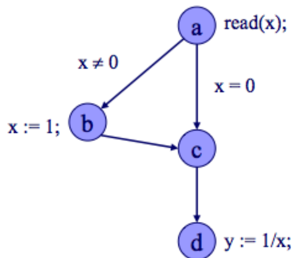


- Critère de tous les noeuds satisfait par : abd (DT1 = {x=1,y=1}) et acd (DT2 = {x=0,y=1})
- L'erreur est détectée par l'exécution de acd

Méthodes de test structurel : couverture de *tous les noeuds*

Soit le programme P6 (avec erreur) :

```
read(x);
...
if (x <> 0) then x := 1;
...
y := 1/x;
```

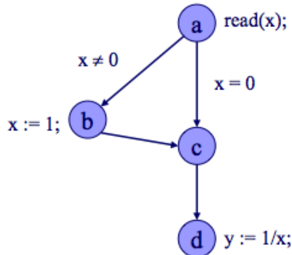


- Critère de tous les noeuds satisfait par : $abcd$ avec `DT1 = {x=1}`
- La couverture de tous les noeuds permet-elle de détecter l'erreur ?

Méthodes de test structurel : couverture de *tous les noeuds*

Soit le programme P6 (avec erreur) :

```
read(x);
...
if (x <> 0) then x := 1;
...
y  := 1/x;
```



- Critère de tous les noeuds satisfait par : $abcd$ avec `DT1 = {x=1}`
- L'erreur n'est pas détectée

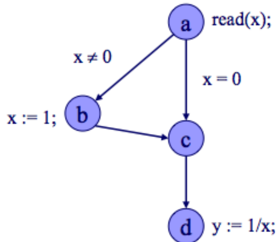
Le critère *tous les noeuds* est insuffisant pour détecter une majorité d'erreurs de programmation.

Méthodes de test structurel : couverture de *tous les arcs*

- Couvrir au moins une fois chaque arc du graphe de contrôle
- Si le critère *tous les arcs* est réalisé, le critère *tous les noeuds* est satisfait

Soit le programme P6 (avec erreur) :

```
read(x);
...
if (x <> 0) then x := 1;
...
y  := 1/x;
```



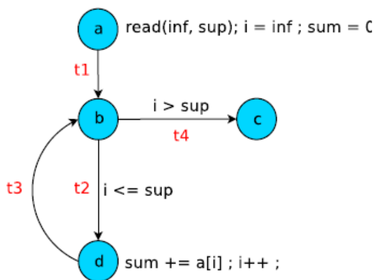
- Critère de tous les arcs satisfait par : $abcd$ et acd avec {`DT1 = {x=1}, DT2={x=0}`}
- L'erreur est détectée

Méthodes de test structurel : couverture de *tous les arcs*

```
/* a : tableau d'entiers > 0 */
read(inf,sup);
i = inf;
sum = 0;

while (i <= sup) {
  sum += a[i];
  i++;
}

printf("%d", 1/sum);
```



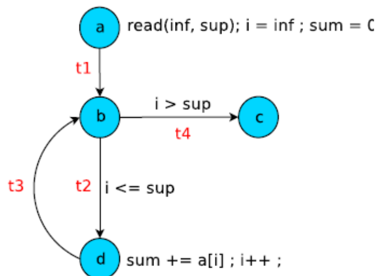
- Critère de tous les arcs satisfait par `DT1 = {a[3]={50,60,60},inf=0,sup=2}`
- La couverture de tous les arcs permet-elle de détecter l'erreur ?

Méthodes de test structurel : couverture de *tous les arcs*

```
/* a : tableau d'entiers > 0 */
read(inf,sup);
i = inf;
sum = 0;

while (i <= sup) {
  sum += a[i];
  i++;
}

printf("%d", 1/sum);
```



- Critère de tous les arcs satisfait par `DT1 = {a[3]={50,60,60},inf=0,sup=2}`
- Le chemin $t1t4$ révèle un défaut non détecté par la couverture de tous les arcs

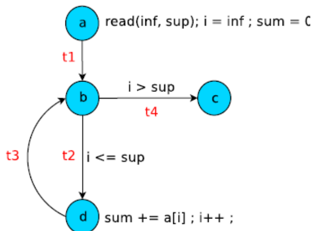
Méthodes de test structurel : couverture de *tous les chemins indépendants*

- Parcourir tous les arcs dans chaque configuration possible
- Pour prouver que des chemins sont indépendants, il suffit de trouver un arc qui n'est pas contenu dans les deux chemins
- Le nombre de chemins peut devenir très grand : on cherche des couvertures incomplètes

```
/* a : tableau d'entiers > 0 */
read(inf,sup);
i = inf;
sum = 0;

while (i <= sup) {
  sum += a[i];
  i++;
}

printf("%d", 1/sum);
```



- `DT1 = {a[3]={50,60,60},inf=0,sup=2}` couvre tous les arcs mais pas tous les chemins.
- `DT1,DT2={inf=1, sup=0}` couvre tous les chemins

Méthodes de test structurel : exercice

Soit le programme suivant :

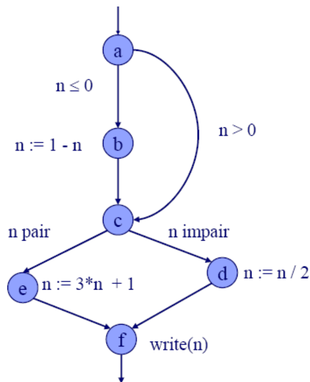
```
If n ≤ 0 then
   n := 1-n
end;
If n impair
Then n := n / 2
Else n := 3*n + 1
end ;
Write(n);
```

1. Donner le graph de flot de contrôle associé à ce graph
2. Calculer les DT pour les critères :
   - *tous-les-noeuds*
   - *tous-les-arcs*
   - *tous-les-chemins-indépendants*

Méthodes de test structurel : exemple

Soit le programme suivant :

```
If n ≤ 0 then
n := 1-n
end;
If n impair
Then n := n / 2
Else n := 3*n + 1
end ;
Write(n);
```



- *tous-les-noeuds* : {DT1={n=0},{DT2={n=-1}}
- *tous-les-arcs* : {DT1={n=0},{DT2={n=1}}
- *tous-les-chemins-indépendants* : {DT1={n=-1},{DT2={n=-2},{DT3={n=1},{DT4={n=2}}

Difficultés du test

- Test exhaustif impossible à réaliser
  - En test fonctionnel, l'ensemble des données d'entrée est en général infini ou de très grande taille
  - En test structurel, le parcours du graphe de flôt de contrôle conduit à une forte explosion combinatoire

  $\rightarrow$ le test est une méthode de vérification partielle de logiciels

  $\rightarrow$ la qualité du test dépend de la pertinence du choix des données de test
- Choix des cas de tests :
  - Couvrir au mieux l'espace d'entrées avec un nombre réduit d'exemples
  - les zones sujettes à erreurs nécessitent une attention particulière
- Difficultés d'ordre psychologique : le test est un processus destructif : un bon test est un test qui trouve une erreur

Plan

1. Introduction
2. Test logiciel
3. Méthodes de test
4. Outils pour le test unitaire
   - JUnit
   - Doublures d'objets

Outils du test unitaire

Réalisation des TU :

- manuelle : débugger, méthode main dans chaque classe et traces
- automatique : *frameworks* dédiés à l'automatisation des TU : séparation claire du code et des tests, analyse automatisée des résultats, facilite les tests de non-régression donc la maintenance corrective, évolutive, le refactoring.

Outils du test unitaire

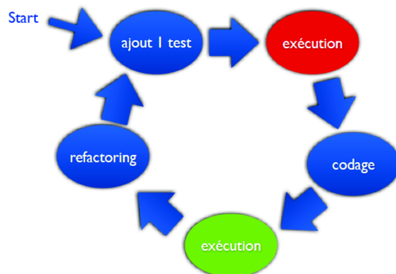
Utilité des TU automatisé

- primordiaux dans certaines méthodologies agiles (XP, TDD)
- améliorer la qualité, fiabilité, conception du code notamment :
  - la granularité des méthodes : il est plus facile des tester des méthodes courtes que de longues méthodes
  - Une classe avec un couplage fort vers d'autres classes est difficile à tester : réduit le couplage entre les objets
- automatise les tests de non-régression

Test et méthodes agiles

TDD *Test Driven Development*

- Méthode de développement dans les méthodes agiles
- Préconise l'écriture des tests avant le développement du code :
  - Développer les tests en premier
  - Développer le code correspondant
  - Refactoriser
- Utilise les outils du TU



Mise en oeuvre du TU automatique

3 règles

- tester le plus possible : afin d'augmenter les chances de découvrir des bugs
- tester le plus tôt possible : plus les tests sont fait tôt plus les bugs sont rapidement détectés
- tester le souvent possible : en les automatisant et si possible en les intégrant dans un processus d'intégration continue

Principes des TU

- le test doit être le plus petit et le plus simple possible
- les TU doivent être automatisés

Outils du test unitaire automatique

Framework d'automatisation des TU

Ecriture et automatisation des TU :

- Java : JUnit, TestNG
- PHP : PHPUnit, atoum, SimpleTest
- C++ : CppUnit
- .NET : NUnit

Outils d'analyse de couverture de code

Mesure de la couverture de code d'un jeu de tests :

- Java : Cobertura, Emma
- PHP : Xdebug

Outils d'automatisation des tests IHM

- Selenium pour les applications web

Outils du test unitaire automatique

JUnit + Eclemma

Java - CursorableLinkedList.java - Eclipse SDK

File Edit Source Refactor Navigate Search Project Run Window Help

JUnit

Finished after 34,898 seconds

Runs: 13009/13009 Errors: 0 Failures: 0

Failures | Hierarchy

- junit.framework.TestSuite
  - junit.framework.TestSuite
    - TestBagUtils
    - org.apache.commons.collections.TestClos
    - org.apache.commons.collections.TestColle
    - TestBufferUtils
    - TestEnumerationUtils
    - org.apache.commons.collections.TestFact
    - TestListUtils
    - TestMapUtils
    - org.apache.commons.collections.TestPred
    - TestSetUtils
    - org.apache.commons.collections.TestTran
    - TestArrayStack
    - TestBeanMap
    - org.apache.commons.collections.TestBina
    - TestBoundedFifoBuffer
    - TestBoundedFifoBuffer2
    - TestCursorableLinkedList
    - TestDoubleOrderedMap
    - org.apache.commons.collections.TestExte
    - TestFastArrayList
    - TestFastArrayList1
    - TestFastHashMap
    - TestFastHashMap1
    - TestFastTreeMap
    - TestFastTreeMap1

Failure Trace

CursorableLinkedList.java

```java
public boolean addAll(int index, Collection c) {
    if(c.isEmpty()) {
        return false;
    } else if( size == index ||  size == 0) {
        return addAll(c);
    } else {
        Listable succ = getListableAt(index);
        Listable pred = (null == succ) ? null : succ.prev();
        Iterator it = c.iterator();
        while(it.hasNext()) {
            pred = insertListable(pred,succ,it.next());
        }
        return true;
    }
}
```

Problems | Javadoc | Declaration | Console | Coverage

TestAllPackages (31.10.2006 15:04:14)

Element	Coverage	Covered Lines	Total Lines
java - commons-collections	79,5 %	10927	13738
org.apache.commons.collections	74,1 %	3842	5183
ArrayStack.java	86,5 %	32	37
BagUtils.java	86,7 %	13	15
BeanMap.java	72,4 %	155	214
BinaryHeap.java	87,6 %	127	145
BoundedFifoBuffer.java	93,2 %	82	88
BufferOverflowException.java	55,6 %	5	9
BufferUnderflowException.java	88,9 %	8	9
BufferUtils.java	30,8 %	4	13
ClosureUtils.java	93,9 %	31	33
CollectionUtils.java	92,4 %	293	317
ComparatorUtils.java	8,6 %	3	35
CursorableLinkedList.java	85,4 %	444	520

Writable | Smart Insert | 149 : 28

Mise en oeuvre du TU automatique

Tests en isolation

Le résultat d'un test ne doit pas dépendre de l'exécution des tests précédents :

- utilisation de design patterns
- utilisation d'objets de type *mock* ou doublures d'objets
- éviter de faire appels aux ressources externes dans les cas de tests
- ...

JUnit

1. Introduction

2. Test logiciel

3. Méthodes de test
   - Méthodes de test fonctionnel
   - Méthodes de test structurel

4. Outils pour le test unitaire
   - JUnit
   - Doublures d'objets

JUnit

- open-source : `www.junit.org`
- framework écrit en java
- intégration dans Eclipse, Netbeans, ...

JUnit offre :

- des assertions pour exprimer les oracles
- basé sur des annotations (Java 5)
- la sélection des tests à lancer
- la création de suite de tests
- la possibilité de lancer tous les tests d'un coup
- affichage graphique
- ne permet que le TU

Structure du projet

Les tests ne sont pas dans le source du projet !

Le répertoire de travail contient :

- `src` (sources) contenant packages applicatifs
- `test` (tests) avec les même packages
- `classes` (exécutables)

Avantages :

- permet de livrer ou non les tests
- garde pour les tests la structure en package de l'application

Assertions

Servent à décrire l'oracle du cas de test :

- Une assertion est une expression booléenne supposée vraie
- Méthodes statiques de `org.junit.Assert`
- Lèvent une erreur `AssertionFailedError` si l'assertion n'est pas vérifiée

- `assertTrue(String msg, boolean test)`, `assertFalse(String msg, boolean test)`
- `assertEquals(String msg, Object expected, Object actual)`
- `assertSame(Object expected, Object actual)`, `assertNotSame(Object expected, Object actual)`
- `assertNull(Object o)`, `assertNotNull(Object o)`

Voir aussi Assertj `http://joel-costigliola.github.io/assertj/`

Classe de Test

- Classe contenant des cas de test
- Par convention, la classe de test qui teste la classe `modele.Calcul` (répertoire `src`) a pour nom `modele.TestCalcul` (répertoire `test`)
- Une méthode de la classe de test représente un cas de test
- Par convention, une méthode qui teste la méthode `getLevel` a pour nom `testGetLevel`
- Une méthode de test est
  - annotée par @Test
  - publique, type de retour `void`
  - pas de paramètre, peut lever une exception
  - contient forcément un oracle (assertion ou vérification de levée d'exception)

Classe de Test : exemple

Counter
- count : int
+ Counter() + Counter(int) + int increment() + int decrement + int getValue() + Counter add(Counter) + Counter sub(Counter)

```java
import org.junit.Test;
import static org.junit.Assert.*;
public class TestCounter {
   // ...
   @Test
   public void testAdd() {
      Counter c1 = new Counter (10);
      Counter c2 = new Counter (12);
      Counter c3 = c1.add(c2);
      assertTrue(c3.getValue() == c1.getValue()
            + c2.getValue());
    }
   }
```

L'exécuteur de tests Junit attrape les objets de type `AssertionFailedError` et indique :

- le test passe : barre verte
- le test échoue : barre rouge (*Failures* petite croix bleue)
- erreur : barre rouge (*Errors* petite croix rouge)

Fixture ou objets fixes

Méthode annotée par @Before :

- Par convention, appelée `setUp()`
- publique, `throws Exception`
- appelée avant chaque appel d'une méthode de test
- sert à factoriser la construction des objets

Counter
- count : int
+ Counter() + Counter(int) + int increment() + int decrement + int getValue() + Counter add(Counter) + Counter sub(Counter)

```
import org.junit.Test;
import static org.junit.Assert.*;
public class TestCounter {
  private Counter c1;
  private Counter c2;
   @Before
   public void setUp() throws Exception {
      c1 = new Counter (10);
      c2 = new Counter (12);
   }
   @Test
   public void testAdd() {
      Counter c3 = c1.add(c2);
      assertTrue(c3.getValue() == c1.getValue()
            + c2.getValue());
    }
   }
```

Fixture ou objets fixes

```
import org.junit.*;
import static org.junit.Assert.*;

public class TestCounter {

    @BeforeClass
    public static void setUpClass() {
        ...
    }

    @AfterClass
    public static void tearDownClass() {
        ...
    }

    @Before
    public void setUp() {
       ...
    }

    @After
    public void tearDown() {
       ...
    }

    @Test
    public void m1() {
        ...
    }
    @Test
    public void m2() {
        ...
    }
}
```

Code exécuté une seule fois avant d'exécuter toutes les méthodes de test de la classe

Code exécuté une seule fois après avoir exécuté toutes les méthodes de test de la classe

Code exécuté avant chaque méthode de test de la classe

Code exécuté après chaque méthode de test de la classe

setUpClass()
setUp()
m1()
tearDown()
setUp()
m1()
tearDown()
tearDownClass()

Gestion des exceptions

Vérifier que des exceptions sont levées comme prévue :

- annotation `@Test(expected = ExceptionAttendue.class)`

Vérifier qu'une méthode ne dépasse pas un temps fixé :

- annotation `@Test(timeout=10)`

```
@Test (expected = IndexOutOfBoundsException.class)
public void testEmpty() {
   new ArrayList<Object>().get(0);
}
@Test (timeout=10)
public void testMethode() {
   ...
}
```

Suite de tests

Pour tester tous les tests des classes `CounterTest1` et `CounterTest2` :

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({CounterTest1.class, CounterTest2.class});

public class TestAll{
}
```

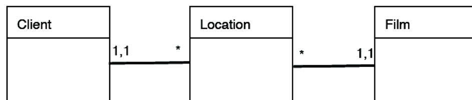
Doublures d'objets

1. Introduction
2. Test logiciel
3. Méthodes de test
   - Méthodes de test fonctionnel
   - Méthodes de test structurel
4. Outils pour le test unitaire
   - JUnit
   - Doublures d'objets

Test en isolation

Indépendance des tests

Le résultat d'un test ne doit pas dépendre de l'exécution des tests précédents pour s'assurer que l'échec d'un test n'est pas lié à une de ses dépendances.




```java
public class Location {
  private Film film;
  private Client client;
  ...
  public float montant (int duree)
    { if (client.getCat() ==
          PRIVILEGE)
        return film.prixJour()*(
             duree-1);
      else ...
  }}
```

```java
public class Film {
  private Categorie categorie;
  private String titre;
  ...
  public float prixJour(){
     switch(categorie){
       case Categorie.NOUVEAUTE:
         return categorie.prixBase()
              *...;
         ...
  }}
```

Si erreurs dans classe `Film`, tests sur méthode `montant` peuvent échouer à cause d'erreurs dans la méthode `prixJour`.

Test en isolation

Les doublures d'objets

Les objets permettant de simuler le comportement d'autres objets de façon maîtrisée :

- *stub* (bouchon) : classes qui renvoient en dur une valeur pour une méthode invoquée
- *spy* (espion) : classe qui vérifie l'utilisation qui en est faite après l'exécution
- *mock* (simulacre) : classe qui agit comme un *stub* et un *spy*

Test en isolation

Intérêts des Mocks

- Utiles pour les TU, pour simuler un objet qui n'est pas encore écrit, pour éviter d'invoquer des ressources longues à répondre, pour obtenir un état difficilement reproductible
- Maitrise des dépendances durant un test
- Réaliser les tests d'un objet de façon isolée et répétable.
- Vérifier les invocations qui sont faites sur un objet (nombres d'invocations, paramètres fournis, ordre d'invocations, ...)
- Mise en oeuvre via un framework :
  - Création dynamique d'objets mocks, généralement à partir d'interfaces.
  - Mockito `http ://code.google.com/p/mockito/`
  - JMock, EasyMock, ...

Mockito

Générateur automatique de doublures :

1. Création des mocks pendant la phase de création des objets du test
2. Description du comportement des objets (bouchonnage - stubbing) pendant la phase de création des objets du test
3. Lors de l'exécution du code à tester, mémorisation des interactions avec les mocks (observateur - vérification)
4. L'oracle peut interroger les mocks pour savoir comment ils ont été utilisés

Mockito

1- Création des mocks pendant la phase de création des objets du test

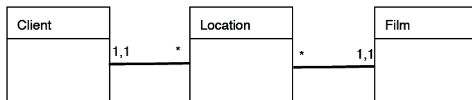
- `import static org.mockito.Mockito.*;`
- `Film mockfilm = Mockito.mock(Film.class) ;`
- `Film mockfilm = Mockito.mock(‘‘monfilm",Film.class) ;`
- Le mock peut appeler tous les appels de méthode de l'interface/classe `Film`.

Mockito

2- Description du comportement des objets (bouchonnage - stubbing) pendant la phase de création des objets du test

- Retour d'une valeur :
  `when((mockfilm.prixJour()).thenReturn(3.5) ;`
- Levée d'exception : `doThrow(new MonException()).when(loc).montant(10);`
- Stubbing avancé :
  `when((mockfilm.prixJour()).thenReturn(3.5).thenReturn(3)`
- `when(loc.montant(10)).thenThrow(new MonException())`

Mockito




```
public class Location {
 private Film film;
 private Client client;
 ...
 public float montant (int duree)
   { if (client.getCat() ==
        PRIVILEGE)
        return film.prixJour()*(
            duree-1);
     else ...
 }}
```

```
public class Film {
 private Categorie categorie;
 private String titre;
 ...
 public float prixJour(){
    switch(categorie){
     case Categorie.NOUVEAUTE:
      return categorie.prixBase()
          *...:
      ...
 }}
```

```
@Test
public void testMontant() {
 Film film = Mockito.mock(Film.class) ;
 Mockito.when(film.prixJour()).thenReturn(3.5) ;
 Client client = Mockito.mock(Client.class) ;
 Mockito.when(client.getCat()).thenReturn(PRIVILEGE) ;
 Location loc = new Location(film, client) ;
 Assert.assertEquals(3.5, loc.montant(2)) ;
}
```

Mockito

3- Lors de l'exécution du code à tester, mémorisation des interactions avec les mocks (observateur)

- L'oracle peut interroger les mocks pour savoir comment ils ont été utilisés
- `verify(film).prixJour()` lève une exception si la méthode `prixJour` n'est pas appelée exactement une fois

```
@Test
public void testMontant() {
 Film film = Mockito.mock(Film.class) ;
 Mockito.when(film.prixJour()).thenReturn(3.5) ;
 Client client = Mockito.mock(Client.class) ;
 Mockito.when(client.getCat()).thenReturn(PRIVILEGE) ;
 Location loc = new Location(film, client) ;
 loc.montant(2) ;
 Mockito.verify(film).prixJour() ;
}
```

Mockito

Vérifications avancées :

- `verify(mock, times(2)).methode()`
- `verify(mock, atLeastOnce()).methode();`
- `verify(mock, atMost(1)).methode();`
- `verify(mock, never()).methode();`
- `verifyNoMoreInteractions(mock);`
- `verifyZeroInteractions(mock)`

```
verify(loc, times(2)).montant(1);
```

Mockito

Argument matchers

- permettent de ne pas préciser une valeur d'argument
- `anyObject()` tout objet ou null
- `anyInt()` tout entier, Integer ou null
- `anyString()`
- ...

```
verify(loc, never()).montant(anyInt())
doThrow(new MonException()).when(loc).montant(anyInt()));
```

Mockito

Ordre d'appels :

- s'assurer qu'un mock est appelée avant un autre
- `InOrder inO = inOrder(firstMock,secondMock)`
- `inO.verify(firstMock).methode();`
- `inO.verify(secondMock).methode();`
- ...

Mock et injection de dépendance

- Si une méthode à tester utilise une dépendance directement instanciée :

```
public class ClasseA {
 public String maMethode(){
   // debut des traitements
   ClasseB classeB = new ClasseB();
   // suite des traitements utilisant classeB
   }}
```

- injection de dépendance

```
public class ClasseB implements InterfaceB { ...}

public class ClasseA {
 public String maMethode(){
   // debut des traitements
   InterfaceB classeB = creerClasseB();
   // suite des traitements utilisant classeB
   }
   protected InterfaceB creerClasseB() {
    return new ClasseB();}
}
```

Mock et injection de dépendance

- Pour le test, on réécrit la méthode qui instancie la dépendance

```
public class TestClasseA {
  @Test
  public void testMaMethode() {
    ClasseA classeA = new ClasseA(){
    // reecriture de la methode pour qu'elle renvoie un mock
    protected InterfaceB creerClasseB() {
      // renvoie une instance du mock de la classe B
      InterfaceB mockB = mock(ClasseB.class);
      return mockB;
    }} ;
    String resultat = classeA.maMethode();
   // evaluation des resultats du cas de test
   }}
```