Test et vérification

Objectifs du module

- Distinguer les concepts de faute, d'erreur et de défaillance et comprendre la notion d'enchaînement entre la défaillance et la faute
- Connaître et maîtriser les concepts du test logiciel
  - Test fonctionnel/structurel, test unitaire, test de non régression, cas de test, oracle, critères de couverture, développement dirigé par les tests
- Savoir écrire de *bons* tests
  - Maîtriser des outils de test unitaire et de couverture du code
  - Connaître différentes techniques de test et savoir les appliquer
  - Être en mesure de juger de la qualité des tests
- Savoir expliquer ce qui différencie le test des autres approches de vérification formelle
- Connaître les principes de la sémantique axiomatique à la Floyd-Hoare
  - Triplet de Hoare, règles de déduction, invariant de boucle

Au menu

Au menu

Vérification & Validation (V&V)

- Vérification – *Are we building the product right ?*
  - Construisons-nous bien le produit ?
  - Autrement dit, est-ce que le logiciel fonctionne correctement ?
  - Le logiciel doit correspondre à sa spécification
- Validation – *Are we building the right product ?*
  - Construisons-nous le bon produit ?
  - Autrement dit, est-ce que le logiciel réalise les fonctions attendues ?
  - Le logiciel doit répondre aux attentes de l'utilisateur
  - Nécessite la participation des utilisateurs et est en général faite par des équipes spécifiques qui ne développent pas

Bugs

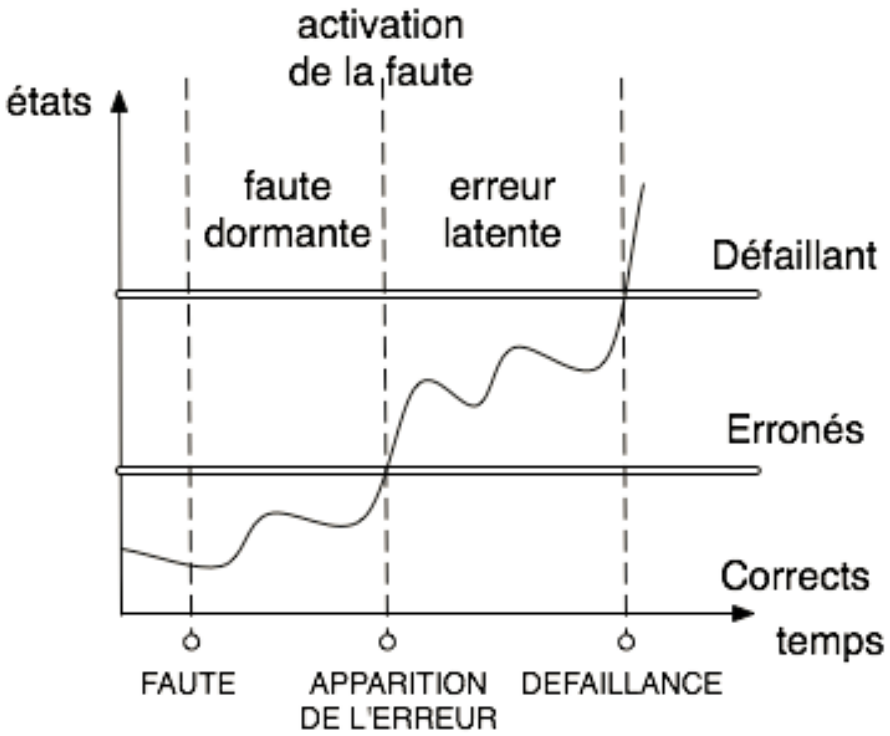
- Locaux
  - Code redondant
  - Condition erronée
  - Initialisation erronée
  - Omission
  - Manque de vérification
  - Division par zéro
  - Approximations
  - *etc.*
- Globaux (émergeant d'interactions)
  - Hypothèses erronées
  - Erreur dans la réutilisation de logiciels
  - Problème de concurrence (*e.g.*, *race condition*)
  - Interactions improbables entre le matériel, le logiciel et l'utilisateur
  - *etc.*

Faute, erreur, défaillance (1)

Une faute du développeur introduit une erreur dans le système qui provoquera sa défaillance à l'exécution

- Une *faute* est une décision inappropriée ou erronée, faite par un développeur lors de la conception/implémentation du système, qui peut conduire à l'introduction d'une erreur
- Une *erreur* est la manifestation d'une ou plusieurs fautes dans le système (imperfection dans un des aspects du système) qui peut contribuer à la survenance d'une défaillance
  - État atteint par le système qui n'avait pas été spécifié
  - État que le système n'était pas supposé atteindre
- Une *défaillance* est un comportement inacceptable présenté par le système (différence entre le comportement attendu et le comportement observé du système)
  - Le comportement du système n'est plus conforme à sa spécification
  - La fréquence des défaillances reflète la fiabilité du système

Faute, erreur, défaillance (2)



- Une faute ne produit pas toujours une erreur : faute dormante
- Une erreur n'entraîne pas toujours une défaillance : erreur latente

- Si un système peut être considéré comme un ensemble de composants, la conséquence de la défaillance d'un composant est une faute (interne) pour le système qui le contient, et aussi une faute (externe) pour les composants qui interagissent avec lui

Exemple

- Système : le code dans un nœud (commutateur, routeur, ...) d'un réseau de télécommunications
- Dans ce code, le programmeur a écrit `i = 0;` au lieu de `i = 1;` qui était l'instruction correcte
  - Il y a donc une faute dans le système
- À un moment particulier, l'ordinateur exécute cette instruction, et, suite à un certain calcul, un tampon est dimensionné à 10 au lieu d'être dimensionné à 100
  - Il y a donc une erreur dans le système
- Les conditions d'utilisation du réseau font qu'exceptionnellement, ce jour-là, la charge est telle que le tampon reçoit un trafic trop important, il y a alors trop de pertes (plus que les niveaux maximaux spécifiés)
  - Il y a donc une défaillance du système

Pourquoi la chasse aux *bugs* ?

- Augmenter la confiance en ce qui a été programmé
- Assurer la qualité des logiciels
  - Imposée par des lois/normes/assurances
    - ★ *e.g.*, la norme DO-178C pour l'avionique
- Les *bugs* font partie intégrante de l'informatique
- Les *bugs* sont-ils fréquents ?
  - Mettez à jour les applications de votre *smartphone*...
- Mais en fait... les *bugs*, est-ce si grave ?

Des *bugs* logiciels aux conséquences désastreuses (1)

- Aérospatiale
  - 1962 Perte d'itinéraire de la sonde *Mariner 1* au lancement
    - Cause : erreur de transcription d'une équation (problème de spécification)
  - 1996 Auto-destruction d'*Ariane 5* (vol 501), 37 secondes après décollage
    - Cause : conversion flottant 64 bits trop grand vers entier 16 bits (réutilisation d'un composant d'*Ariane 4*)
    - Coût : 370 millions de dollars
  - 2004 Blocage du robot *Mars Rover Spirit*
    - Cause : trop de fichiers ouverts en mémoire flash
- Médecine
  - 1985-87 5 morts par irradiations massives dues à la machine de radiothérapie *Therac-25*
    - Cause : conflit d'accès aux ressources entre deux parties logicielles (problème de synchronisation)
    - Coût : plusieurs vies humaines

Des *bugs* logiciels aux conséquences désastreuses (2)

- Télécommunications
  - 1990 Crash à grande échelle du réseau *AT&T*
    - Cause : toute unité défaillante alertait ses voisines, mais la réception du message d'alerte causait une panne du récepteur (effet domino)
- Énergie
  - 2003 Panne d'électricité aux États-Unis et au Canada (*General Electric*), 55 millions d'habitants plongés dans le noir
    - Cause : mauvaise gestion d'accès concurrents aux ressources dans un programme de surveillance
    - Coût : 6 milliards de dollars
- Finance
  - 2012 Bourse de Tokyo paralysée par un *bug* logiciel
- Automobile
  - 2014-15 Campagnes de rappel de millions de véhicules (*Toyota*, *Ford*, *Land Rover*) en raison d'un *bug* logiciel

Des *bugs* logiciels aux conséquences désastreuses (3)

- Informatique
  - 1994 *Bug FDIV du Pentium* (Intel) dans l'unité de calcul en virgule flottante (FPU)
    - Cause : algorithme de division erroné
  - 1978–95 Faille dans le protocole d'authentification de *Needham-Schroeder*
    - Cause : attaque *man in the middle*
  - 2006–08 Clés générées par *OpenSSL* et données cryptées non sûres, impactant les applications l'utilisant (*e.g.*, `ssh`)
    - Cause : générateur de nombres aléatoires cassé
  - 2012–14 *Bug Heartbleed* dans *OpenSSL* permettant de récupérer les clés secrètes des serveurs, les noms et mots de passe des utilisateurs, ainsi que le contenu échangé par ces derniers sur les serveurs compromis
    - Cause : oubli de la validation d'une variable contenant une longueur

Coût des *bugs*

- Pour l'utilisateur : coûts économiques, humains, environnementaux...
- Pour le fournisseur, le coût de la correction des *bugs* augmente de façon importante avec l'avancée dans le cycle de vie du logiciel
  - En phase d'implémentation : coût 1
  - En phase d'intégration (*bug* de conception) : coût 10
  - En phase de recette (*bug* de spécification) : coût 100
  - En phase d'exploitation : coût > 1000

  $\Rightarrow$ Plus un *bug* est découvert tard, plus il coûte cher !
- 10 milliards de dollars par an en tests rien qu'aux États-Unis
- Les phases de test peuvent être plus longues que les phases de spécification, conception et implémentation réunies
  - En moyenne 30% du développement d'un logiciel standard
  - Plus de 50% du développement d'un logiciel critique (parfois $> 90\%$)

Pourquoi la chasse aux *bugs* ?

- Les *bugs* sont-ils évitables ?
  - *Errare humanum est*
  - Complexité et taille croissantes des logiciels

  $\Rightarrow$ Besoin de méthodes pour avoir des assurances de correction !

Comment détecter et corriger des *bugs* ?

- Il suffirait d'écrire un « super-compilateur » qui, en plus de compiler, détecterait les *bugs*
  - Comment décrire le comportement attendu du programme à vérifier ?
  - Comment être certain que ce super-compilateur est lui-même correct ?
  - Est-il réellement possible d'écrire un tel super-compilateur ?

Limites théoriques

Théorème de Rice

Toute propriété non triviale des langages récursivement énumérables est indécidable.

- En clair, il n'est pas possible de programmer un « super-compilateur » qui, à la compilation, prendrait en entrée un programme source et :
  - Détecterait les instructions non atteintes
  - Détecterait les assertions fausses
  - *etc.*
- Notion d'indécidabilité
  - Une propriété indécidable signifie qu'elle ne pourra jamais être prouvée dans le cas général (pas de procédé systématique/algorithme)
  - Quelques exemples :
    - ★ L'exécution d'un programme termine (problème de l'arrêt)
    - ★ Deux programmes calculent la même chose
    - ★ Un programme n'a pas d'erreur

La chasse aux *bugs* n'est pas facile

- Il n'est donc pas possible de détecter les *bugs* automatiquement
- Et pourtant, c'est une nécessité !
- Pour les détecter, il existe différentes approches complémentaires :
  - Test
  - Preuve de programme
  - Vérification de modèle (*model-checking*)
  - *etc.*

Test (1)

« Tester peut démontrer la présence d'erreurs, mais pas leur absence », Dijkstra, Notes on Structured Programming, 1972

- Peut trouver des *bugs*, mais pas garantir leur absence
  - Ce n'est pas parce qu'un ensemble de tests réussit qu'un logiciel fonctionne bien
- Repose sur l'exécution du système réel, quelles que soient sa taille et sa complexité
- Un programme a un nombre infini (ou gigantesque) d'exécutions possibles

```
Lire un entier positif N
Tant que N > 1 faire ...
```

  - Si un entier est codé sur 32 bits, $2^{31} \approx 2$ milliards de valeurs !
  - Explosion combinatoire

Test (2)

- Un jeu de tests n'examine qu'un nombre fini (petit) d'exécutions possibles
  - Exhaustivité impossible
- Trouver des heuristiques
  - Approcher l'infini (ou le gigantesque) avec le fini (petit)
  - Tester les exécutions les plus « représentatives »
- Est la méthode la plus utilisée pour assurer la qualité des logiciels
  - Facile à appréhender
  - Plus proche du code
  - Moins coûteux à mettre en œuvre
- Est utilisé pour analyser toutes sortes d'artefacts logiciels (code, modèles, spécifications, *etc.*)

Preuve de programme

- Garantit (par construction) l'absence de *bugs*
  - Preuve formelle, fondée sur la modélisation logique et le raisonnement
  - Exhaustif
  - Approche descendante (*top-down*)
- Pas entièrement automatique, demande de l'expertise
  - Rappel : la plupart des propriétés d'un programme sont indécidables...
- Est utilisée sur des parties spécifiques de logiciels critiques (*e.g.*, compilateurs certifiés)
- Exemples de langages de spécification et d'outils d'aide à la preuve : Isabelle/HOL, B, Coq, *etc.*

Vérification de modèle (*model-checking*)

- Intermédiaire entre le test et la preuve de programme
- Travaille sur un modèle comportemental formel (automate) de système pour en vérifier des propriétés
- Donne des garanties
  - Exploration exhaustive des états du système
  - Vérifications automatiques de propriétés sur les exécutions du modèle : absence d'interblocage (*deadlock*), propriétés de sûreté (*safety*) et de vivacité (*liveness*), *etc.*
- Ne fonctionne que sur des abstractions du système
  - Distance entre le modèle et le système réel ?
- Est limitée par la taille du système (explosion d'états)
- Est utilisée dans la vérification matérielle et logicielle
  - Détecter de façon automatique les *bugs* dans les circuits, dans les protocoles de communication...
  - S'applique bien en phase de conception, ou après modélisation
- Exemple d'outils : Promela/SPIN, SMV, Java PathFinder, *etc.*

Au menu

Définitions du test

« Tester, c'est exécuter le programme dans l'intention d'y trouver des anomalies ou des défauts », Myers, The Art of Software Testing, 1979

« Le test est un processus manuel ou automatique, qui vise à établir qu'un système vérifie les propriétés exigées par sa spécification, ou à détecter des différences entre les résultats engendrés par le système et ceux qui sont attendus par la spécification », norme IEEE-STD729, 1983

Techniques de test

- Test statique
  - Relecture/revue de code
  - Analyse automatique : vérification de propriétés, règles de codage...
- Test dynamique
  - Exécution du programme avec des valeurs spécifiques en entrée et observation du comportement

Avec quoi on teste ?

- Une spécification qui exprime ce qu'on attend du système
  - Un cahier des charges (en langue naturelle)
    - ★ Fonctionnalités du logiciel
    - ★ Facteurs de qualité : portabilité, évolution, robustesse, *etc.*
    - ★ Contraintes : performances temporelles, taille du logiciel, environnement d'exécution, *etc.*
    - ★ Interfaces avec l'environnement
  - Des règles de codage
  - Des commentaires dans le code
  - Des contrats sur les opérations (*e.g.*, Eiffel, JML)
  - Un modèle UML
  - Une spécification formelle (automate, modèle B, *etc.*)
- Un code source
- Un exécutable

Qu'est-ce qu'on teste ? Quelles propriétés ?

Critères d'évaluation (qualité du logiciel)

- Fonctionnalités
- Sécurité/intégrité
- Facilité d'utilisation
- Cohérence
- Maintenance
- Efficacité
- Robustesse
- Sûreté de fonctionnement
- Portabilité
- *etc.*

Comment on teste ?

Accessibilité

- Test fonctionnel (test boîte noire)
  - Utilise la description des fonctionnalités du programme (*i.e.*, une spécification formelle ou informelle)
  - Aucune connaissance des détails d'implémentation n'est nécessaire
  - Ne se fonde que sur les résultats produits (et non pas sur l'implémentation qui les produit)
- Test structurel (test boîte blanche)
  - Utilise la structure interne du programme (analyse du code source)
  - Prend en compte le mécanisme qui produit les résultats
  - Exemples :
    - ★ Tests des représentations des données, des structures de données utilisées
    - ★ Tests de contrôle de gestion explicite de la mémoire (*e.g.*, en C/C++, lecture ou écriture de mémoire libérée, libération multiple de mémoire, lecture et utilisation de mémoire non initialisée, fuite mémoire...)

Complémentarité du test fonctionnel et structurel

- Détection de fautes différentes
  - Test fonctionnel : détecte plus facilement les oublis ou les erreurs par rapport à la spécification
  - Test structurel : détecte plus facilement les erreurs de programmation
- Exemple :

```
// addition de deux entiers modulo 100000
int sum(int x, int y) {
  if (x == 600) && (y == 500)
    return x - y; // bug1
  else
    return x + y; // bug2
}
```

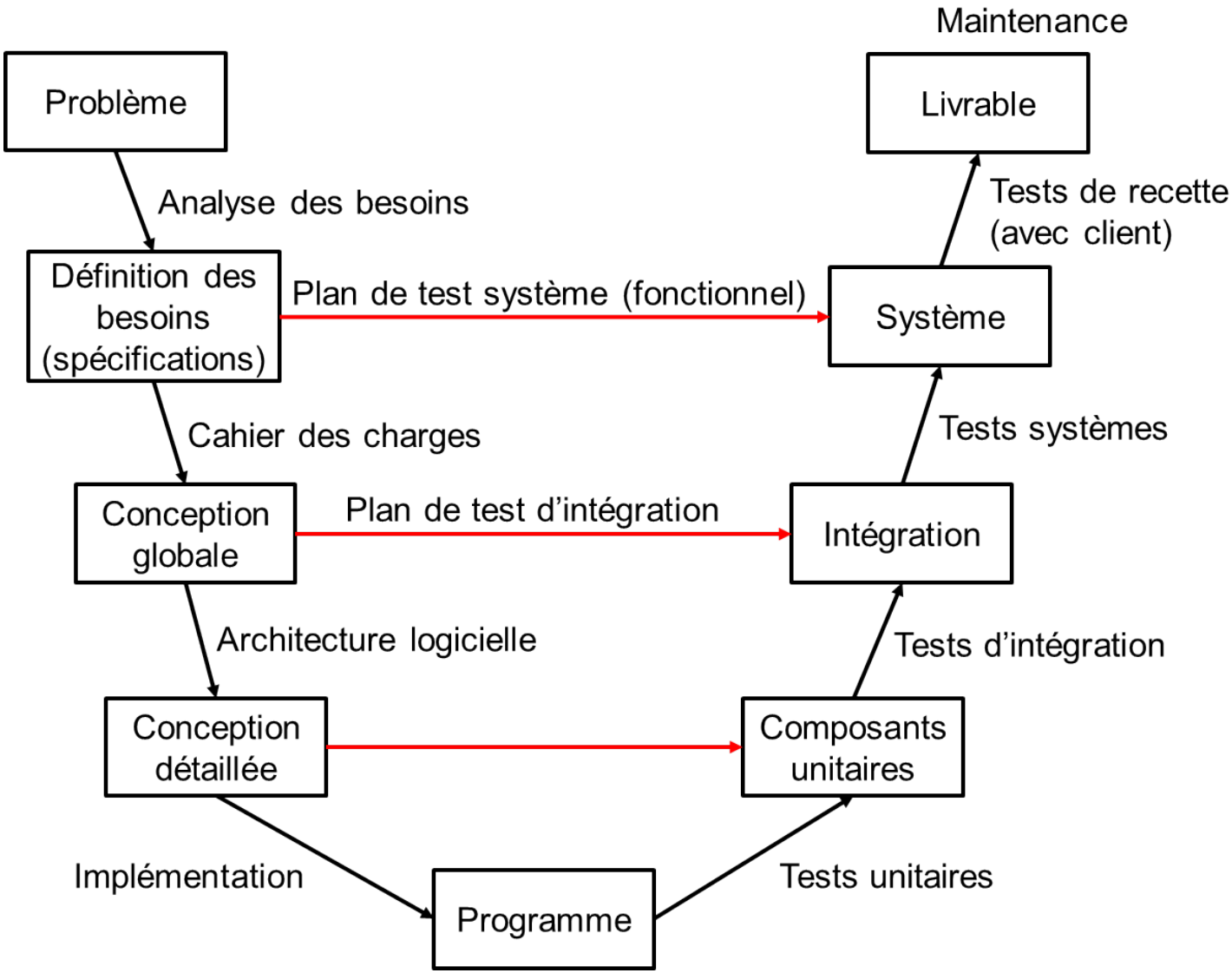
  - Test fonctionnel : détecte l'erreur par rapport à la spécification (*bug1* difficile, *bug2* facile)
  - Test structurel : détecte l'erreur pour les valeurs $x = 600$ et $y = 500$ (*bug1* facile, *bug2* difficile)
    - ★ En examinant ce qui a été réalisé, on ne prend pas forcément en compte ce qui aurait dû être fait

Plusieurs niveaux/étapes

Stratégie (cycle de vie du logiciel)

- Test unitaire
- Test d'intégration
- Test système
- Test de non régression
- Test de recette (avec client)

Cycle en V


Problème
Analyse des besoins
Définition des besoins (spécifications)
Plan de test système (fonctionnel)
Cahier des charges
Conception globale
Plan de test d'intégration
Architecture logicielle
Conception détaillée
Implémentation
Programme
Tests unitaires
Composants unitaires
Tests d'intégration
Intégration
Tests systèmes
Système
Tests de recette (avec client)
Livrable
Maintenance

Test unitaire

- Défini par rapport à une seule composante de code à la fois (*i.e.*, une unité compilable)
  - Par exemple, une fonction, une classe, un module ou un composant
- Validation d'un seul comportement à la fois de façon isolée (*i.e.*, indépendamment des autres)
  - Une unité peut avoir beaucoup de tests unitaires associés
- Le plus fréquent et le plus facile à mettre en œuvre
- Exemple GPS : algorithme de calcul d'itinéraire sur des exemples de graphes construits a la main
- Nombreux *frameworks* disponibles (*e.g.*, JUnit pour Java)

Test d'intégration

- Validation de la composition des modules *via* leur interface (communications entre modules, appels de procédures...)
- Choisir un ordre pour intégrer et tester les différents modules du système
  - Cas simple
    - ★ Il n'y a pas de cycle dans les dépendances entre modules
    - ★ Les dépendances forment un arbre et on peut intégrer simplement de bas en haut
  - Cas plus complexe
    - ★ Il y a des cycles dans les dépendances entre modules
    - ★ Très fréquent dans les systèmes à objets
    - ★ Il faut des heuristiques pour trouver un ordre d'intégration
- Exemple GPS : lecture des données depuis la base de données ou encore communications avec l'IHM
- Différents *frameworks* disponibles (*e.g.*, EasyMock) s'utilisent en complément/avec les *frameworks* de tests unitaires
  - Principales techniques : bouchon de tests (*stubs*), *mock objects* pour simuler les modules non disponibles

Test système

- Validation de la globalité du système à partir de l'interface (test fonctionnel) avec son environnement d'exécution (matériel, système d'exploitation, machine virtuelle, bibliothèques, serveurs...)
  - Fonctionnalités offertes
  - Qualité du système
    - ★ Charge, ergonomie, sécurité, performances, *etc.*
- Exemple GPS : utilisation du logiciel sur des scénarios réalistes et complets
- Avec UML
  - Diagrammes de cas d'utilisation
  - Diagrammes de séquence

Test de non régression

- Type de test transversal (unitaires, intégration, système, *etc.*)
- Vérifier que des modifications apportées au logiciel n'ont pas introduit de nouvelles erreurs
  - Autrement dit, vérifier que ce qui marchait marche encore
- Dans la phase de développement et de maintenance du logiciel après :
  - Refactorisation
  - Ajout/modification/suppression de fonctionnalités
  - Correction d'une faute
- Rejouer automatiquement les tests
  - Tous ?
  - Détection des tests obsolètes ?

Riche zoologie

- Quelques autres types et appellations de tests[1] : *accessibility testing, ad hoc testing, agile testing, alpha/beta testing, arc testing, back-to-back testing, big-bang testing, bottom-up testing, branch testing, compatibility testing, compliance testing, concurrency testing, configuration testing, confirmation testing, conversion testing, cover testing, data driven testing, data integrity testing, decision condition testing, decision table testing, dynamic/static testing, exhaustive testing, exploratory testing, incremental testing, isolation testing, maintenance testing, model-based testing, mutation testing, pair testing, path testing, profile testing, reliability testing, sanity test, security testing, site acceptance testing, etc.*

1. *Standard glossary of terms used in Software Testing. Erik van Veenendaal, 2005*

Aide au génie logiciel

- Les tests peuvent servir :
  - À l'élaboration d'un logiciel (l'avant)
  - Au développement d'un logiciel (le pendant)
  - À la validation d'un logiciel (l'après)

Complément à l'énonciation des besoins

- Les tests permettent de préciser les besoins et les demandes d'un client (même au niveau d'un contrat)
  - Ils sont spécifiés *avant* la mise en œuvre (voire même *implémentés avant*)
- En spécifiant des tests, on découvre également de nouveaux besoins
- Il s'agit de tests unitaires ou système, en général en boîte noire, qui explicitent et valident les besoins fonctionnels et non fonctionnels
- Plus ces tests sont détaillés, plus ils aideront au développement du logiciel
- Dans le cahier des charges, seuls des tests non triviaux et qui apportent une information seront ajoutés

Aide au développement d'un logiciel

- Les tests sont implémentés *pendant* la mise en œuvre, afin de valider au fur et à mesure ce qui est développé
- Ils aident à se prémunir d'une dégradation de l'état d'un logiciel
  - En effet, à chaque modification d'un programme, il y a une forte probabilité d'intégrer de nouvelles erreurs...
- Enfin, ils sont effectués en présence du client afin de valider l'application complète

Vocabulaire du test

- Objectif de test
  - Comportement du système à tester
- Données de test
  - Données à fournir en entrée au système de manière à déclencher un objectif de test
- Résultats d'un test
  - Conséquences ou sorties de l'exécution d'un test (*e.g.*, affichage à l'écran, modification des variables, envoi de messages...)
- Cas de test
  - Données d'entrée et résultats attendus associés à un objectif de test
- Oracle
  - Décision de la réussite de l'exécution d'un test
  - Comparaison entre le résultat attendu et le résultat obtenu

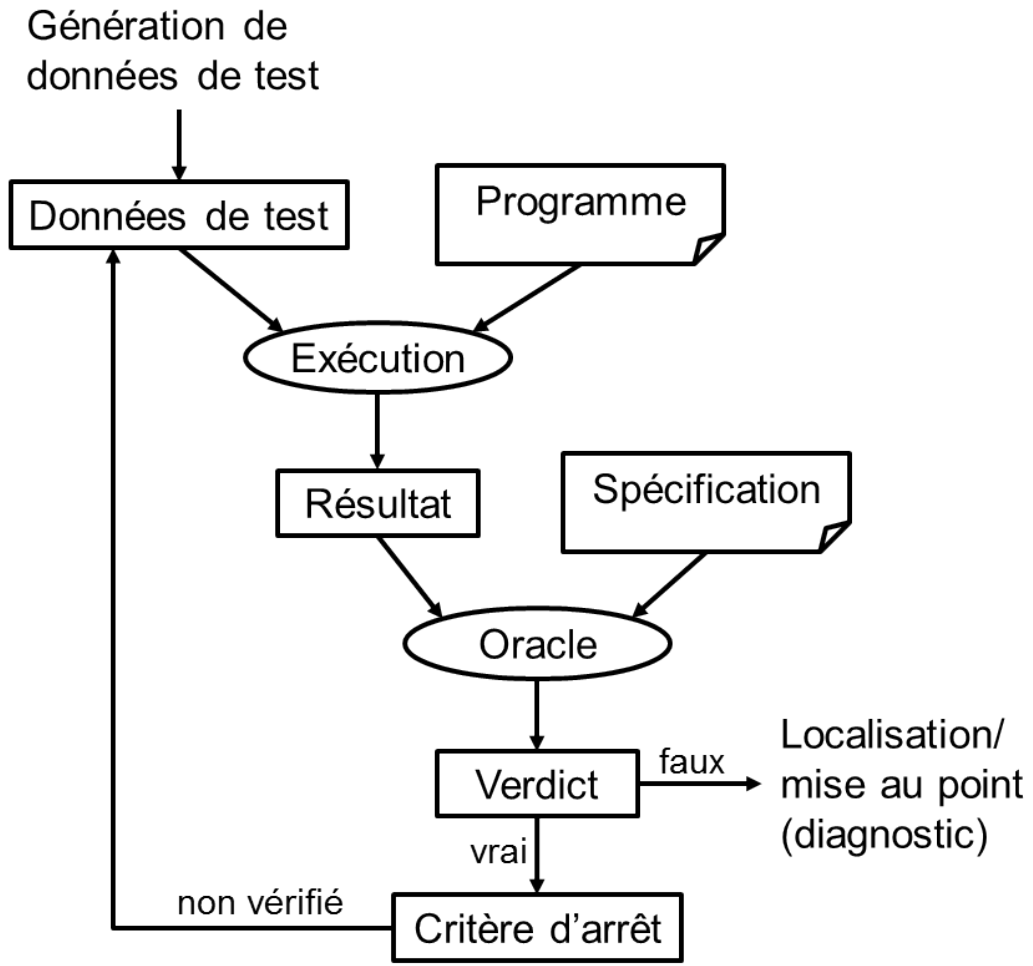
Exemples de cas de test pour le tri d'un tableau d'entiers

Objectif de test	Donnée de test	Résultat attendu	Résultat du test
tableau vide	[]	[]	[...]
tableau à 1 élément	[5]	[5]	[...]
tableau >= 2 éléments, déjà trié	[1, 2, 3, 4]	[1, 2, 3, 4]	[...]
tableau >= 2 éléments, non trié	[11, 20, 22, 17]	[11, 17, 20, 22]	[...]

- Oracle : résultat attendu = résultat du test ?

Processus de test

1. Choisir les comportements à tester (*i.e.*, les objectifs de test)
2. Choisir des données de test permettant de déclencher ces comportements + décrire le résultat attendu pour ces données (*i.e.*, les cas de test)
3. Exécuter les cas de test sur le système + collecter les résultats
4. Comparer les résultats obtenus aux résultats attendus pour établir un verdict



Problème du test

- Soit D le domaine d'entrée d'un programme P spécifié par la spécification S, on voudrait pouvoir dire :
  - Soit D le domaine de P : $\forall x \in D, P(x) = S(x)$
- Test exhaustif impossible dans la plupart des cas
  - Domaine D trop grand, voire infini
  - Trop long et coûteux
- On cherche alors un ensemble de données de test T tel que :
  - $T \subset D$
  - Si $\forall x \in T, P(x) = S(x)$ alors $\forall x \in D, P(x) = S(x)$
- Critère d'arrêt pour la génération de données de test
  - {données de test} = T

Génération de test

- Génération déterministe
  - « à la main »
- Génération automatique aléatoire
- Génération automatique aléatoire contrainte
  - Mutation
  - Test statistique
- Génération automatique guidée par les contraintes
- Génération de test avancée (à partir des exigences, de modèles : *Model Based Testing*)
- Reste à savoir quand on a suffisamment testé
  - Critères de test structurels, fonctionnels
  - Analyse de mutation
- Choisir le bon niveau pour le test

Critère d'arrêt et notion de couverture

- Critère d'arrêt = conditions objectives, mesurables, permettant de considérer qu'une série de tests est suffisante
- Test fonctionnel (boîte noire)
  - Pour arrêter les tests, on se fonde sur des critères externes :
    - ★ Couverture des exigences
    - ★ Couverture des fonctions des interfaces
    - ★ Taux de défaillances par jour inférieurs à un seuil
- Test structurel (boîte blanche)
  - Pour arrêter les tests, on se fonde sur des critères internes :
    - ★ Couverture de la structure du système : chaque composant, chaque dépendance entre composants
    - ★ Couverture du code
- Le test structurel permet d'être plus précis dans l'élaboration du critère d'arrêt des tests

Oracle (1)

- Fonction qui évalue le résultat d'un cas de test
- Plus formellement, soient :
  - Un programme $P : Dom(P) \rightarrow Ran(P)$
  - Une spécification $S : Dom(P) \rightarrow Ran(P)$
  - Une donnée de test $X \in Dom(P)$
  - Un oracle $O : Dom(P) \times Ran(P) \rightarrow bool$
    - ⋆ $O(X, P(X)) = true$ ssi $P(X) = S(X)$
- Comment comparer $P(X)$ et $S(X)$?
  - Plus S est formalisé, plus on peut automatiser

Oracle (2)

- Oracle manuel
  - On « regarde » le résultat et un humain évalue s'il est bon
- Construire le résultat exactement attendu
  - Comparer le résultat obtenu avec le résultat construit (diff)
- Assertions
  - Dans le code (programmation défensive)
  - Aux interfaces (*design by contract*)
  - Dans les cas de test (*e.g.*, JUnit)

Test fonctionnel

- Spécification formelle
  - Modèle B, Z
  - Automate, système de transitions
- UML
  - Diagramme de cas d'utilisation (*use cases*)
  - Diagramme de classes (+ contrats)
  - Diagramme de séquence
  - Diagramme d'états
- Description en langue naturelle

Test structurel

- À partir d'un modèle du code
  - Modèle de contrôle (conditionnelles, boucles...)
  - Modèle de données
  - Modèle de flot de données (définition, utilisation...)
- Utilisation importante des parcours de graphes
  - Critères fondés sur la couverture du code

Problème de l'oracle

- La décision de la réussite de l'exécution d'un test peut être complexe
  - Types de données sans prédicat d'égalité
  - Système non déterministe : résultat possible mais pas celui attendu
  - Heuristique : approximation du résultat optimal attendu

Problème de l'oracle

Exemples

- Trouver le maximum d'un tableau
  - Donnée de test : [4, 2, 8, 1]
  - Résultat attendu : 8
  - Oracle : égalité entre entiers OK
- Calculer l'itinéraire le plus rapide entre deux villes
  - Donnée de test : Paris – Bordeaux
  - Résultat attendu : ...A10...
  - Oracle : égalité des chemins ? Non
- Problème du sac à dos (résolu avec une heuristique)
  - Oracle : résultat raisonnablement éloigné du résultat optimal ? ? Non

Problème de l'oracle

Exemples

- Trouver le maximum d'un tableau
  - Donnée de test : [4, 2, 8, 1]
  - Résultat attendu : 8
  - Oracle : égalité entre entiers OK
- Calculer l'itinéraire le plus rapide entre deux villes
  - Donnée de test : Paris – Bordeaux
  - Résultat attendu : ...A10...
  - Oracle : trajet de 5h12 (quel que soit l'itinéraire choisi) OK
- Problème du sac à dos (résolu avec une heuristique)
  - Oracle : résultat = résultat optimal + 5% OK

Problème de l'oracle

- En général, résultat attendu = ensemble de conditions si plusieurs solutions sont possibles, mais qu'une énumération est impossible
- Risque : échec d'un programme conforme si la définition est trop stricte par rapport au résultat attendu
  - Faux positifs
  - Par exemple, le calcul d'itinéraire dans lequel on impose un chemin

Faux positif et faux négatif

- Validité des tests
  - Les tests n'échouent que sur des programmes incorrects
  - Un faux positif fait échouer un programme correct
- Complétude des tests
  - Les tests ne réussissent que sur des programmes corrects
  - Un faux négatif fait passer un programme incorrect
- Validité des tests indispensable, complétude impossible en pratique
  - Toujours s'assurer que les tests sont valides

Tester est difficile

- Aspects psychologiques
  - Processus « destructif » : un bon test est un test qui trouve une erreur
  - Plus facile de trouver les erreurs des autres que les siennes
- Sélection de tests
  - La qualité du test dépend de la pertinence du choix des données de test
  - Il ne faut pas rater un comportement fautif
  - Mais les cas de test sont coûteux (conception, exécution, stockage...)
- Enjeux
  - Choisir des données de test suffisamment variées pour espérer trouver des erreurs
  - Éviter les données de test redondantes ou non pertinentes

$\Rightarrow$ Nécessite de maîtriser des techniques et des outils

$\Rightarrow$ Demande de l'invention, du travail, de la compréhension des mécanismes sous-jacents, de l'expertise

Critères de test fonctionnel

- Le test fonctionnel n'utilise que la description fonctionnelle du programme (*i.e.*, la spécification)
- Plusieurs informations
  - Domaine d'entrée
  - Scénarios
  - Cas d'utilisation
  - Existant
  - Invariants
  - *etc.*

Domaine d'entrée

- Plusieurs niveaux
  - Type des paramètres d'une méthode
  - Pré-condition sur une méthode
  - Ensemble de commandes sur un système
  - Grammaire d'un langage
  - *etc.*
- Impossible de tout explorer, il faut délimiter
  - Analyse partitionnelle
  - Test aux limites
  - Test combinatoire
  - Test aléatoire
  - Graphe causes – effets
  - *etc.*

Test de domaine

- Les tests de domaine de définition vérifient le comportement par rapport aux valeurs/données manipulées
- Dans le cas simple d'une fonction :

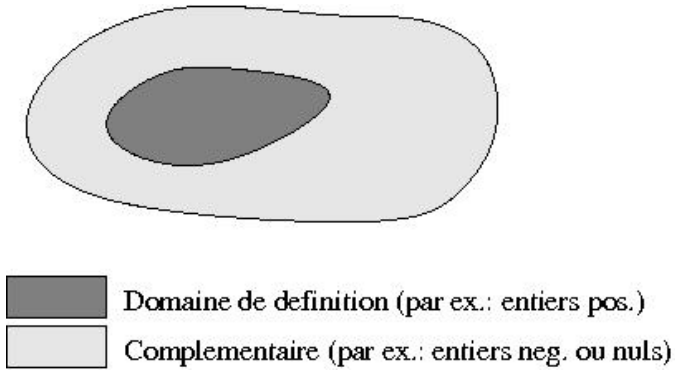
$$\begin{array}{rccc} f: & \mathcal{S} & \rightarrow & \mathcal{B} \\ & x & \mapsto & f(x) \end{array}$$

  - Tests du bon fonctionnement par rapport à $\mathcal{S}$ et $\mathcal{B}$
- Par exemple, la fonction factorielle :

$$\begin{array}{rccc} fact: & \mathbb{N}^* & \rightarrow & \mathbb{N}^* \\ & x & \mapsto & fact(x) \end{array}$$

  - Tests du bon fonctionnement par rapport aux entiers positifs

Test positif et test négatif



- Les tests positifs (ou tests nominaux) se font sur le domaine de définition
  - Ils vérifient les comportements attendus
  - Exemple : `fact` de 1 à 10
- Les tests négatifs (ou tests de robustesse) se font sur le complémentaire de définition
  - Ils vérifient les comportements hors des limites de fonctionnement
  - Exemple : `fact` de -10 à 0
  - Les complémentaires de définition impliquent souvent des erreurs non détectées et silencieuses

Test aux limites

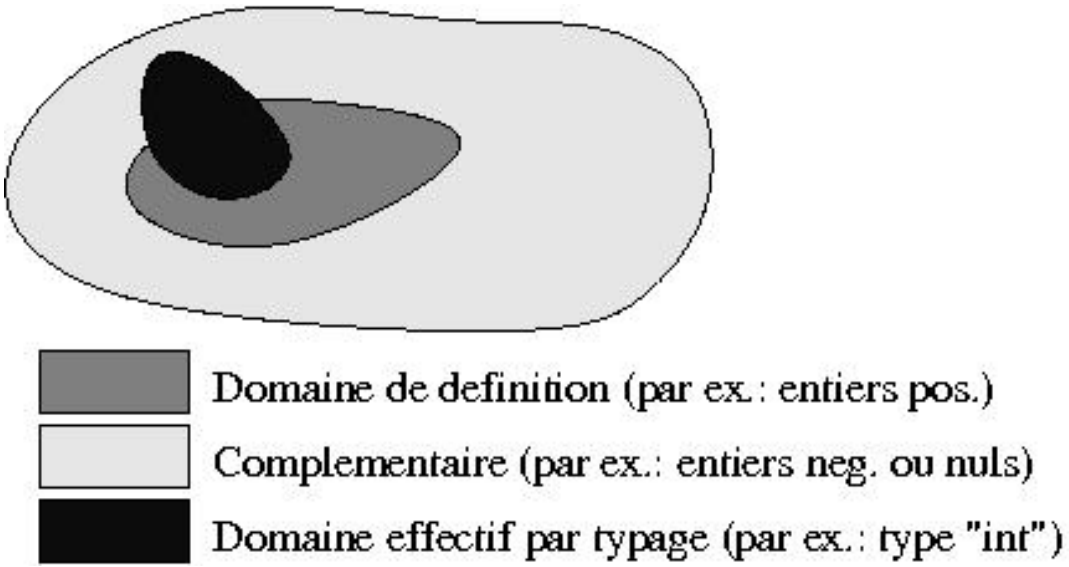
- Intuition : les frontières des domaines sont très propices aux erreurs
- Exemples de frontières :
  - Les minimums et maximums des valeurs admises
  - Les cas particuliers d'un domaine (*e.g.*, une structure vide)
  - Les points particuliers d'une structure (*e.g.*, les extrémités d'un tableau, la racine d'un arbre)
- Le test aux limites vérifient le comportement des valeurs de la frontière des domaines
- Pour chaque donnée en entrée :
  1. Déterminer les bornes du domaine
  2. Prendre des valeurs sur les bornes et juste un peu autour
     - ⋆ Il produit à la fois des cas de test nominaux et de robustesse
- Par exemple, pour l'intervalle $[1, 100]$: $1, 100, 2, 99, 0, 101$
- Souvent utilisé avec l'analyse partitionnelle
  - Les valeurs aux limites peuvent être des valeurs aux frontières des partitions

Domaine effectif

- Les domaines de définition « idéaux » sont souvent contraints par des éléments extérieurs :
  - Contraintes de matériel
  - Contraintes de représentation interne des données
  - Contraintes de typage du langage utilisé
  - *etc.*
- Les domaines effectifs sont souvent différents des domaines idéaux !
- Les tests de domaines doivent en tenir compte

Contraintes de typage

- Les systèmes de type associés à des langages comme C, C++ ou Java, permettent certaines vérifications automatiques des domaines de définition et fixent la représentation interne des données
- Contraintes et compromis entre domaines idéaux et domaines effectifs
  - Par exemple, `int` pour $\mathbb{Z}$ ou même $\mathbb{N}$, `double` pour $\mathbb{R}$



Test du domaine de `fact`

- Exemple de comportement particulier lié à un domaine effectif très différent du domaine idéal
  - Fonction factorielle :

$$\begin{array}{rccc} fact: & \mathbb{N}^+ & \rightarrow & \mathbb{N}^+ \\ & x & \mapsto & fact(x) \end{array}$$

  - Implémentation possible de cette fonction en Java :

```
int fact(int n) {
  return n > 1 ? n*fact(n-1) : 1;
}
```

et donc :

$$\begin{array}{rccc} fact: & int & \rightarrow & int \\ & x & \mapsto & fact(x) \end{array}$$

Test du domaine de `fact`

```
fact (-10) ---> 1
...
fact(0) ---> 1
fact(1) ---> 1
fact(2) ---> 2
fact(3) ---> 6
fact(4) ---> 24
...
fact(15) ---> 2004310016
fact(16) ---> 2004189184
fact(17) ---> -288522240
...
fact(31) ---> 738197504
fact(32) ---> -2147483648
fact(33) ---> -2147483648
fact(34) ---> 0
```

Analyse partitionnelle

- Il s'agit de décomposer/simplifier le domaine en utilisant une connaissance *a priori*
- Les domaines d'entrée sont à partitionner en classes d'équivalences C_i
  - Identifier des classes d'équivalence pour chaque donnée
  - Les classes d'équivalence forment une partition du domaine de chaque donnée en entrée
    - ★ $\bigcup C_i = Dom(P) \wedge \forall i, j, C_i \cap C_j = \emptyset$ si $i \neq j$
  - Choisir une donnée dans chacune
    - ★ Le choix de valeurs aux limites est une heuristique solide de choix de données d'entrée au sein des classes d'équivalence
- Une classe d'équivalence correspond à un ensemble de données de test supposées tester le même comportement, c'est-à-dire provoquer la même erreur
  - $\forall C_i, \forall d, d' \in C_i : P(d)$ correct $\Leftrightarrow P(d')$ correct
- Exemple de subdivision possible du domaine pour `fact` :

$$[-\infty, 0]; [0, 1]; [1, 15]; [15, 16]; [16, 31]; [31, 32]; [32, \infty]$$

Règles de partitionnement des domaines

- Si la donnée 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
- Le choix des classes d'équivalence est important
  - Risque de ne pas révéler une erreur

Test aléatoire

- Il s'agit d'approximer un domaine par un échantillonnage aléatoire
- En général, on suppose une distribution uniforme
  - Échantillonnage fondé sur des fonctions "random" classiques
- Sinon, il est nécessaire de simuler une distribution statistique associée aux données
  - Estimation de la confiance du test
- Remarque : une telle distribution est parfois difficile à estimer (*e.g.*, une IHM pour utilisateurs variés)

Exemples de tests aléatoires

- Tests aléatoires sur une structure de données : génération de suites aléatoires d'opérations sur cette structure
  - Création automatique de tableaux de tailles diverses, contenant des éléments divers
  - Modifications aléatoires variées de ces tableaux par insertion et suppressions
- Tests aléatoires sur un réseau : génération aléatoire d'événements, de requêtes et de connexions
- Remarque : les tests aléatoires demandent souvent de l'implémentation

Test à scénarios

- Les tests à scénarios sont une suite d'opérations/sous-tests liée à un scénario ou un cas d'utilisation de besoin, et qui mène à des résultats spécifiques
  - Par exemple, validation des interactions avec une base de données :
    1. Tester le mécanisme de *login*
    2. Tester les moyens de recherche d'informations
    3. Tester les affichages d'informations
    4. Tester les mécanismes de transfert
    5. Tester la nature des transferts
    6. Tester les mécanismes de *logout*
- Ils peuvent être suffisamment sophistiqués pour nécessiter un langage de script de description des tests
  - Par exemple, tests d'une interface graphique à l'aide de séquences spécifiques scriptées d'opérations (la boucle d'événements est remplacée par l'interprétation de scripts)

Test avec existant

- Certains tests de validation peuvent utiliser de l'existant ou des valeurs connues/calculables par des moyens externes
  - Résultats d'une fonction numérique comparés à des résultats calculés auparavant de manière symbolique
  - Utilisation d'anciens prototypes
  - Utilisation de programmes qui font partie de l'existant

Test avec invariants

- Certains tests de validation peuvent exploiter des invariants spécifiques
  - Une fonction *rev* qui renverse une liste est telle que $rev^2 = id$ ou $rev^3 = rev$
  - Les paires de fonctions de compression/décompression, cryptage/décryptage, cod/cod^{-1} sont telles que $cod(cod^{-1}) = id$
  - Une fonction qui engendre un ensemble statistiquement équivalent à un autre
  - Une fonction qui agit sur des graphes planaires doit laisser invariante la caractéristique d'Euler

Critères de test structurel

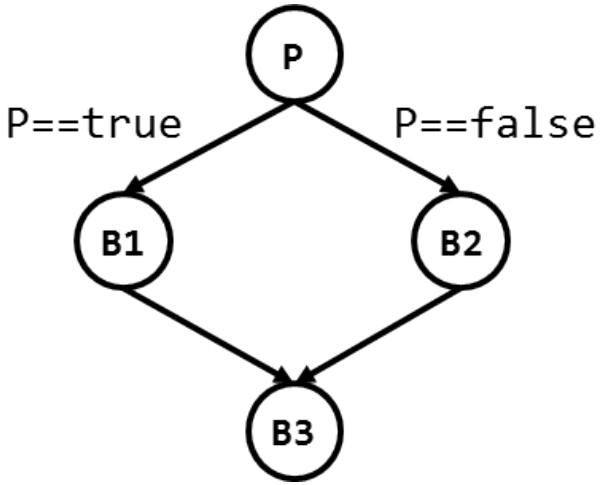
- Les tests sont construits en exploitant :
  - La structure de l'architecture (test d'intégration)
  - La structure du programme (test unitaire structurel)
- Quel est le rôle de la structure (par rapport à une boîte noire) ?
  - Offrir des informations sur la manière dont le système est réalisé (*i.e.*, le comment)
- Les éléments de la structure doivent être couverts par les tests
  - Critères de couverture du code
  - Abstraire la structure pour obtenir un critère formel

Graphe de flot de contrôle (1)

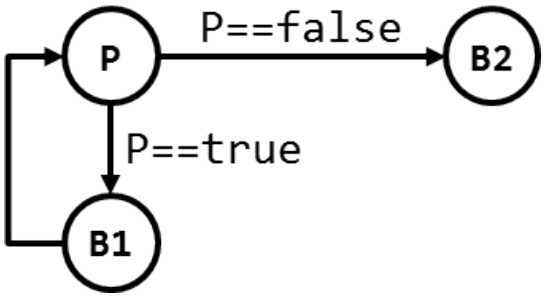
- But
  - Représenter tous les chemins d'exécution potentiels
- Graphe orienté
  - Avec un nœud d'entrée E et un nœud de sortie S
- Sommets
  - Blocs élémentaires du programme : séquence maximale d'instructions séquentielles
  - Prédicats des conditionnelles/boucles
  - Nœuds de jonction « vide » associé à un nœud prédicat
- Arcs
  - Enchaînements d'exécution possibles entre deux sommets

Graphe de flot de contrôle (2)

- Conditionnelle : si P alors $B1$ sinon $B2$ fin $B3$

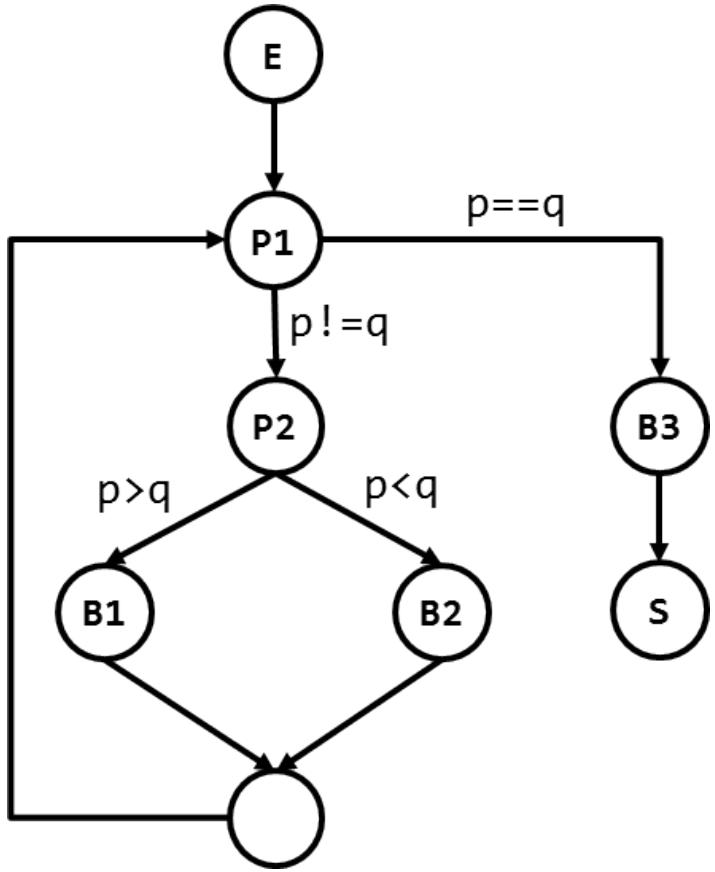


- Boucle : tant que P faire $B1$ fin $B2$



Exemple

```
int pgcd(int p, int q) {
  while (p != q ) { // P1
    if (p > q) // P2
      p = p - q; // B1
    else
      q = q - p; // B2
  }
  return p; // B3
}
```



Couverture de code

- Mesure quantitative utilisée pour décrire le taux de code source testé d'un programme
- Souvent combinée à l'utilisation de tests unitaires
  - Permet d'aboutir à un ensemble réaliste de tests de non régression

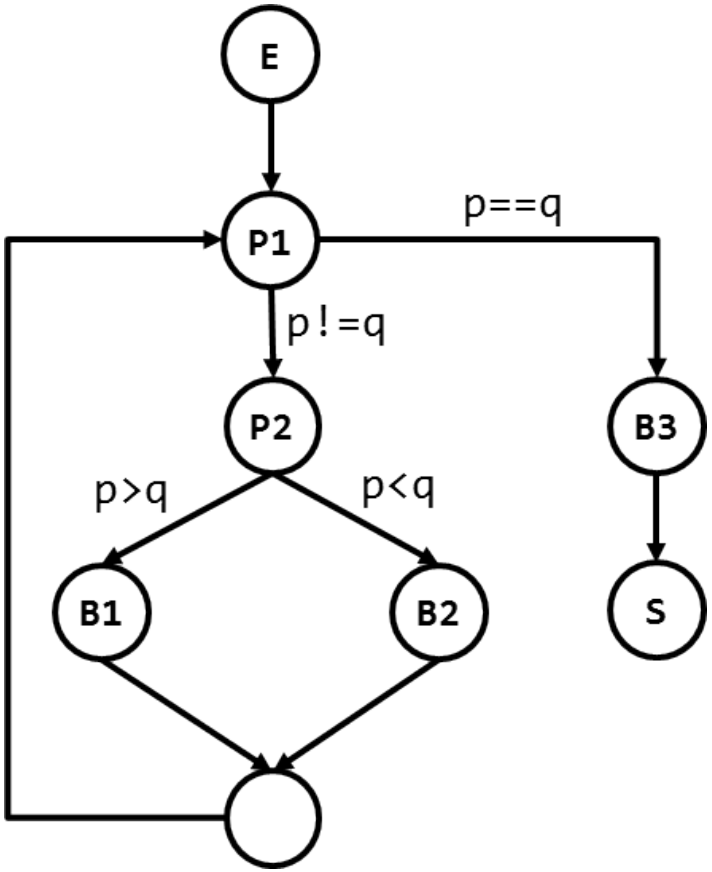
Principaux critères de couverture (1)

- *Statement Coverage*
  - Quelles instructions ont été exécutées ?
  - Couverture des nœuds du graphe de flot de contrôle (*all-nodes*)
- *Branch Coverage*
  - Quelles parties d'un bloc conditionnel (*i.e.*, `if/else`, `switch/case`) ont été exécutées ?
- *Condition Coverage*
  - Quelles valeurs de décision (*e.g.*, `i < num && !skip`) ont été rencontrées à l'exécution ?
  - Couverture des arcs du graphe de flot de contrôle (*all-edges*)
  - Décomposer les conditionnelles dans le graphe de flot de contrôle
  - Critère MC/DC (*Modified Condition/Decision Criterion*)
    - ★ Permet de contrôler le problème d'explosion combinatoire
    - ★ Pour satisfaire MC/DC, il faut, pour chaque condition, générer 2 cas de test tels que la décision change alors que toutes les autres conditions sont fixées

Principaux critères de couverture (2)

- *Function Coverage*
  - Quelles fonctions/méthodes ont été exécutées ?
- *Entry/Exit Coverage*
  - Toutes les possibilités d'invocation et de sortie d'une fonction/méthode ont-elles été utilisées ?
- *Loop Coverage*
  - Aucune, une seule, plusieurs exécutions successives d'une boucle ?
- *Path Coverage*
  - Tous les chemins d'exécution d'une partie du code ont-ils été empruntés ? (*all-paths*)
    - ★ Critère de couverture maximal
  - Tous les k-chemins
    - ★ Les chemins qui passent au maximum k fois dans le corps des boucles
  - Tous les chemins élémentaires
    - ★ 0 à 1 passage dans les corps de boucles

Exemple



- Tous les nœuds
  - (E, P1, P2, B1, P1, B3, S)
  - (E, P1, P2, B2, P1, B3, S)
- Tous les arcs
  - Idem
- Tous les chemins élémentaires
  - Idem + (E, P1, B3, S)
- Tous les 2-chemins
  - Idem +
  - (E, P1, P2, B1, P1, P2, B2, P1, B3, S)
  - ...

Complexité cyclomatique

- $CC = E - N + 2P$ où :
  - E est le nombre d'arcs
  - N est le nombre de nœuds
  - P est le nombre de nœuds de sortie
- C'est le nombre de points de décision + 1 (`if`, `for`, *etc.*)
- Cette mesure indique le nombre minimum de cas de test à écrire
- Dans l'exemple, $CC = 9 - 8 + 2 = 3$

Attention

- *Statement Coverage* $\not\Rightarrow$ *Condition Coverage*

```
void foo(int x) {
  System.out.println("Hello");
  if (x <= 0)
    System.out.println("␣world");
  System.out.println("!");
}
```

Attention

- *Statement Coverage* $\not\Rightarrow$ *Condition Coverage*

```
void foo(int x) {
   System.out.println("Hello");
   if (x <= 0)
      System.out.println("␣world");
   System.out.println("!");
}
```

  - Si foo(-1) est appelée dans un test unitaire, 100% de *Statement Coverage*, mais 50% de *Condition Coverage*

Path Coverage

Propriété

Path Coverage $\Rightarrow$ *Statement / Condition / Branch / Entry/Exit / Loop Coverage*

- Comme le nombre de chemins d'un programme est généralement très grand, voire infini (attention aux boucles), en pratique nous cherchons à écrire des tests permettant de couvrir certains éléments plus limités, tels que les instructions ou les branches
- Attention, la signification du pourcentage de code testé dépend du critère de couverture choisie
  - Par exemple, tester 75% des chemins possibles est plus utile que 75% des instructions !

Indécidabilité de l'exécutabilité d'un chemin

- Il n'existe aucun algorithme général qui permet de décider si un chemin est exécutable ou non
- La présence de chemins non exécutables est souvent le signe d'un code mal écrit, voire erroné

Limite du critère *Statement Coverage*

- Exemple

```
read(x);
if (x != 0) {
   x = 1;
}
y = 1/x;
```

Limite du critère *Statement Coverage*

- Exemple

```
read(x);
if (x != 0) {
   x = 1;
}
y = 1/x;
```

  - Pour $x = 1$: 100% de *Statement Coverage*
  - Pour $x = 0$: *Statement Coverage* incomplet, mais montre une erreur !

- La satisfaction d'un critère de couverture arbitraire n'implique en rien l'absence d'erreur

Limite du critère *Condition Coverage*

- Exemple

```
// tab est un tableau d'entiers positifs
read(inf, sup);
int sum = 0;
for (int i = inf; i <= sup; i++) {
  sum += tab[i];
}
System.out.println("Result: " + 1/sum);
```

Limite du critère *Condition Coverage*

- Exemple

```
// tab est un tableau d'entiers positifs
read(inf, sup);
int sum = 0;
for (int i = inf; i <= sup; i++) {
  sum += tab[i];
}
System.out.println("Result: " + 1/sum);
```

  - Pour $tab = [1, 2, 3]$, $inf = 0$ et $sup = 2$: 100% de *Condition Coverage*
  - Pour $tab = [1, 2, 3]$, $inf = 1$ et $sup = 0$: *Condition Coverage* incomplet, mais montre une erreur !

Ce que ne capture pas ce modèle

- Aspects données
  - Couverture des dépendances entre variables
    - ★ Flot de données
  - Couverture des domaines d'entrée
    - ★ Pertinence des données de test

Conclusion

- Les critères structurels et fonctionnels sont complémentaires
  - Au niveau unitaire
    - ★ Commencer par les tests fonctionnels
    - ★ Compléter par des tests structurels
- Aucun critère n'est meilleur qu'un autre
  - Tous complémentaires !
- Ils permettent de :
  - Quantifier la qualité des tests
  - Savoir ce que l'on teste et donc de générer automatiquement des tests

Au menu

JUnit

- *Framework* de test unitaire pour Java
- Open source : `www.junit.org`
- Créé par Erich Gamma et Kent Beck
- Origine : *Xtreme Programming* (développement dirigé par les tests)
- Intégré dans plusieurs IDE
  - Eclipse, NetBeans, IntelliJ, *etc.*
- Décliné dans beaucoup de langages : xUnit
  - NUnit pour .Net, PyUnit pour Python, *etc.*
- Différences notables entre JUnit 3.8 et JUnit 4
  - Important de connaître les deux !

Ce que JUnit offre

- Des facilités pour le test unitaire d'applications Java
  - Assertions pour exprimer les oracles
  - Méthodes d'initialisation et de finalisation
- Le lancement automatique de suites de test
- Le formatage du diagnostic
  - Savoir quels tests ont échoué, planté, réussi
- Des tests de non régression

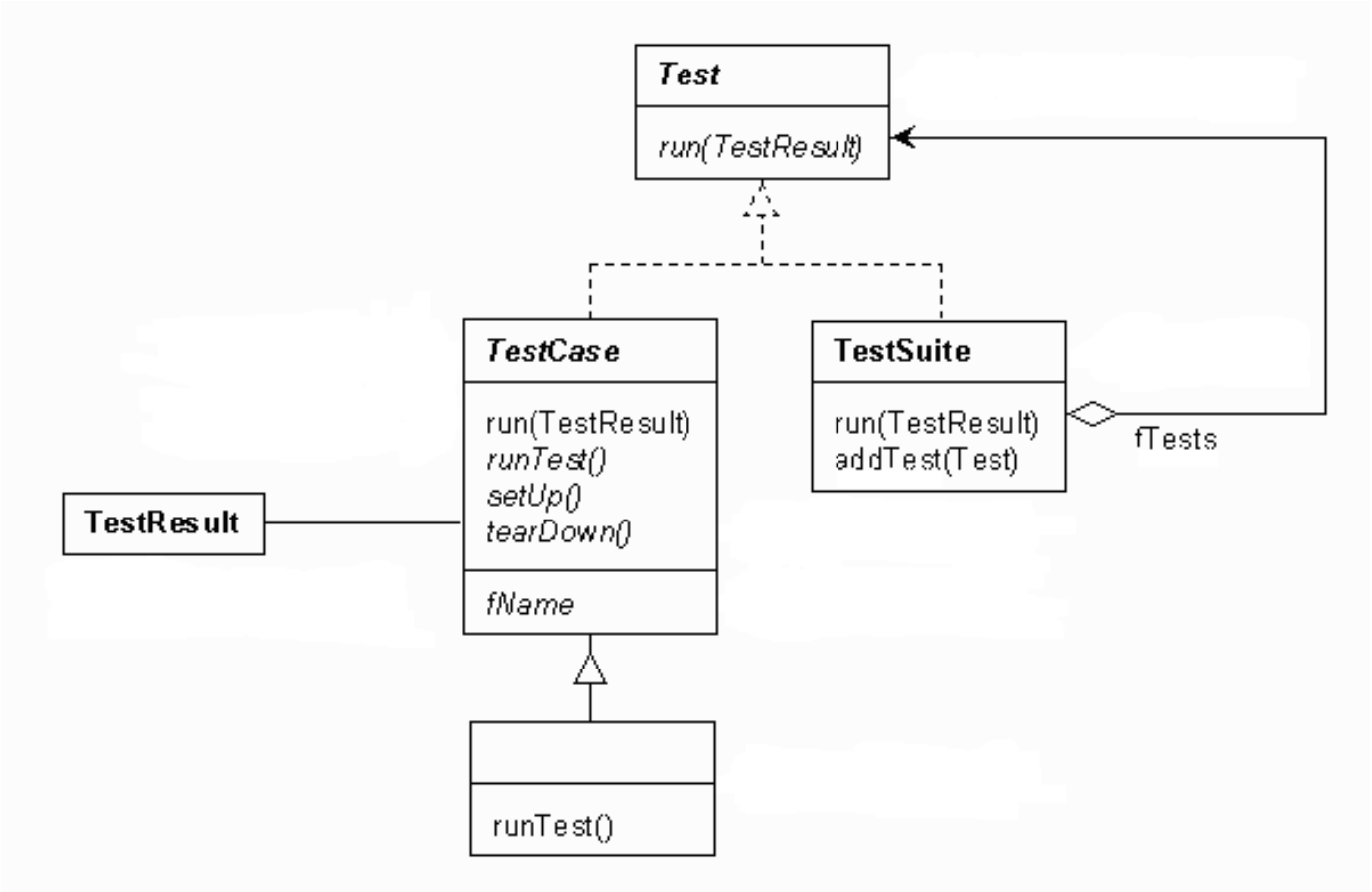
Ce que JUnit n'offre pas

- L'écriture des tests à votre place
- Des recettes pour trouver les bons tests

Rappel : *framework*

- Un *framework* est un ensemble d'interfaces et de classes qui collaborent, mais qui ne s'utilisent pas directement
  - C'est le code du *framework* qui appelle celui du reste de l'application
- C'est une application « à trous » qui offre la partie commune des traitements et est spécialisée par chaque utilisateur selon leurs besoins
  - Il faut implémenter/spécialiser les types fournis

Architecture de JUnit 3.8



- Utilisation
  - Définir les tests
    - ★ En JUnit 3.8, une classe de test hérite de la classe `TestCase`
  - S'en remettre à JUnit pour leur exécution
  - Ne pas appeler explicitement les méthodes de test

Rappel : assertions

- Une assertion est une expression booléenne supposée vraie
- Mot clé `assert` à partir de Java 1.4
  - `assert` *expr_bool*
    - ★ Lève une exception de type `AssertionError` si *expr_bool* est fausse
  - `assert` *expr_bool* : *expr_si_violation*

Assertions de JUnit

- Elles servent à décrire l'oracle du cas de test (plus de `println`)
- Ce sont des méthodes statiques qui lèvent une `AssertionError` si l'assertion est violée
- Différentes assertions :
  - `assertTrue(...)`, `assertFalse(...)`
  - `assertEquals(Object` *`expected`*`, Object` *`actual`*`)`
  - `assertSame(Object` *`expected`*`, Object` *`actual`*`)`
  - `assertNull(...)`
  - `assertEquals(double expected, double actual, double delta)`
  - `assertArrayEquals(Object[] expected, Object[] actual)`
  - ...
  - `assertThat(`*`value, matcher statement`*`)` : nouveauté version 4.4
    - `is(...)`, `not(...)`, `either(...).or(...)`, `each(...)`, `hasItem(...)`
    - Exemple : `assertThat(x, is(not(5))`
    - Messages d'erreur plus clairs

Rappel : test unitaire

- Tester une unité (*i.e.*, une classe) isolément du reste du système
- Test du point de vue client
  - Les cas de tests appellent les méthodes depuis une classe extérieure
  - On ne peut donc tester que ce qui est publique
- Écrire au moins un cas de test par méthode publique
- Choisir un ordre pour le test
  - Quelles méthodes sont interdépendantes ?
- Problème pour l'oracle
  - Encapsulation : les attributs sont privés ou protégés
  - Difficile de récupérer l'état d'un objet
- Penser aux tests au moment du développement
  - Prévoir des accesseurs en lecture sur les attributs privés/protégés
  - Des méthodes pour accéder à l'état de l'objet

Organisation du code des tests

- Créer des classes de test
  - Y insérer des méthodes de test
  - Configurer l'environnement de test
- Créer une suite de tests associée à ces classes de test
  - Lancer cette suite de tests et observer les résultats

Classe de test

- Une classe de test pour une classe testée (ou une méthode testée)
  - Il peut y avoir plusieurs classes de test pour une même classe testée
- Elle contient les méthodes de test
  - C'est une collection de cas de test (sans ordre)
- Elle peut contenir des méthodes particulières pour configurer l'environnement de test

Méthodes de test

- Elles s'intéressent à une seule unité de code/un seul comportement
  - Une méthode de test = un cas de test
- Elles font appel à une ou plusieurs méthodes de la classe du système à tester
- Corps de la méthode
  - Configuration initiale
  - Une donnée de test
    - ★ Un ou plusieurs paramètres pour appeler la méthode testée
  - Un oracle permettant un verdict automatique (*i.e.*, les assertions)
    - ★ Il faut construire le résultat attendu
    - ★ Ou vérifier des propriétés sur le résultat obtenu
- Elles sont indépendantes les unes des autres
- Elles seront appelées par JUnit, dans un ordre quelconque

Environnement de test

- Aussi appelé « l'état du monde »
- Les méthodes de test ont besoin d'être appelées sur des instances
- En général, les instances sont déclarées comme des variables d'instance de la classe de test
- La création des instances, et plus globalement la mise en place de l'environnement de test, sont laissées à la charge des méthodes d'initialisation

Exemple

```
/** Classe qui represente une liste de chaines de caracteres */
public class StringList {
  private int count;
  private Node firstNode = null;
  private Node lastNode = null;
  private Node currentNode = null;
  public StringList(){
    this.count = 0;
  }
  public int size() {...}
  public String item() {...}
  public void next() {...}
  public void previous() {...}
  public void add (String s) {...}
  public void remove() {...}
}
```

- Créer une classe de test qui manipule des instances de `StringList`
- Écrire au moins 6 cas de test (1 par méthode publique)
- Pas d'accès aux attributs privés

JUnit 3.8

- *Package* `junit.framework`
- Paramétrage par spécialisation
- Utilisation de conventions de nommage

Classe de test

- Elle hérite de la classe `TestCase`
- Elle peut utiliser les `assertXXX` sans import
  - La classe `TestCase` hérite de la classe `Assert`

```
import junit.framework.*;

/** Test de la classe StringList */
public class TestStringList extends TestCase {
   // declaration des instances
   private StringList list;
   // configuration de l'environnement de test
   ...
   // methodes de test
   ...
}
```

Préambule (*fixture*) : la méthode `setUp`

- Elle permet de mettre en place l'environnement de test
- Elle est appelée avant chaque méthode de test
  - Rappel : l'exécution est pilotée par le *framework* et non le testeur
- Elle permet de factoriser la construction de « l'état du monde »
- Elle est protégée, mais pas statique

```
protected void setUp() throws Exception {
  super.setUp();
  this.list = new StringList();
}
```

Postambule (*fixture*) : la méthode `tearDown`

- Elle permet de défaire l'environnement de test
- Elle est appelée après chaque méthode de test
- Elle permet de défaire « l'état du monde »
- Elle est protégée, mais pas statique

```
protected void tearDown() throws Exception {
  super.tearDown();
}
```

Méthodes de test (1)

- Elles commencent obligatoirement par le mot `test`
- Elles sont publiques, sans paramètre ni type de retour
- Elles contiennent des assertions (oracle)

```
/** Test de la methode add avec deux elements */
public void testAdd2(){
   list.add("premier");
   list.add("second");
   assertTrue(list.size() == 2);
   assertTrue(list.item() == "second");
}
```

  - Un test se documente !
    - ★ Javadoc

Méthodes de test (2)

- Elles peuvent lever des exceptions

```
/** Test de la methode remove sur une liste vide */
public void testRemoveWithEmptyList(){
   try {
      list.remove();
      fail();
   } catch (EmptyListException e) {
      // OK
   }
}
```

Suite de tests : classe TestSuite

- Une suite de tests permet de rassembler des méthodes de test pour enchaîner leur exécution
- Constructeurs
  - `TestSuite()`
  - `TestSuite(Class)` construit une suite de test contenant tous les cas de test correspondant à une méthode commençant par *test* dans la classe passée en paramètre
  - `TestSuite(Class[])`
- Méthodes
  - `addTest(Test)` pour ajouter un cas de test donné
  - `addTestSuite(Class)` pour ajouter une autre suite de tests

```
public static Test suite() {
  TestSuite suite = new TestSuite();
  suite.addTestSuite(TestStringList.class)
  //suite.addTest(new TestStringList("testAdd2"));
  return suite;
  //return new TestSuite(TestStringList.class);
}
```

Lancement des tests

- On utilise un `TestRunner`
  - Graphique (intégré à Eclipse) : « *keep the bar green to keep the code clean* »
    - `java junit.swingui.TestRunner`
  - Textuel : affichage des résultats sur la console
    - Depuis un main : `junit.textui.TestRunner.run(`*`ma suite de tests`*`)`
    - Depuis une ligne de commande : `junit.textui.TestRunner` *`ma suite de tests`*

Verdicts

- Ils sont définis grâce aux assertions placés dans les cas de test
  - *Pass* (vert)
    - ★ Pas d'erreur détectée, ni d'exception levée
  - *Fail* (rouge)
    - ★ Échec, le résultat attendu n'est pas celui obtenu (une exception de type `AssertionFailedError` a été levée)
  - *Error* (rouge)
    - ★ Le test n'a pas pu s'exécuter correctement (*e.g.*, autre exception levée)

Exemple d'exécution en mode console

```
>java -cp .;junit.jar junit.textui.TestRunner TestStringList
JUnit version 3.8
....E
Time: 0.057
There was 1 failure:
1) testAdd2(TestStringList)
java.lang.AssertionError: Result expected:<true> but was:<false>
...
FAILURES!!!
Tests run: 6,  Failures: 1
```

Comment ça marche derrière

- L'interface `Test` possède une méthode `run()` qui prend en paramètre un `TestResult` servant à stocker le résultat
- Le `TestRunner.run()` appelle cette méthode pour la suite de test
- Pour chaque méthode de test :
  - Exécution du `setUp()`
  - Exécution de la méthode commençant par `test`
    - ★ Pour cela, JUnit utilise l'introspection
  - Exécution de la méthode `tearDown()`

  Puis affichage formaté du `TestResult`

JUnit 4

- Fonctionne avec Java 5
- Utilisation intensive des annotations
- Plus de *runner* graphique (laissé à la charge de l'IDE)
- Nouvelle architecture : *package* `org.junit`
- Beaucoup de nouvelles fonctionnalités
  - Tests paramétrés, *timeouts*, *etc.*

Classe de test

- `import org.junit.Test;`
- `import static org.junit.Assert.*;`
- Aucun héritage !

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

/** Test de la classe StringList */
public class TestStringListV4 {

  // declaration des instances
  private StringList list;
  // configuration de l'environnement de test
  ...
  // methodes de test
  ...
}
```

Méthodes annotées `@Before` ou `@After`

- `import org.junit.Before;` (idem pour `After`)
- Elles sont publiques (et non plus protégées)
- Elles sont appelées avant/après chaque méthode de test
- Possibilité d'annoter plusieurs méthodes (ordre d'exécution indéterminé)

```
@Before
public void init() {
  this.list = new StringList();
}
```

Méthodes annotées `@BeforeClass` et `@AfterClass`

- `import org.junit.BeforeClass;` (idem pour `AfterClass`)
- Elles sont publiques et statiques
- Elles sont appelées avant/après la première/dernière méthode de test
- Une seule méthode pour chaque annotation

Méthodes de test (1)

- Elles ont un nom quelconque
- Elles sont annotées `@Test`
- Elles sont publiques, sans paramètre ni type de retour
- Elles peuvent lever des exceptions
  - `@Test(expected=MyException.class)` pour indiquer le type de l'exception attendue
  - `@Test(timeout=10)` pour indiquer que le test échoue (*fail*) si la réponse n'arrive pas avant le *timeout* (en ms)
- Elles sont annotées `@Ignore` pour être ignorées

Méthodes de test (2)

```
@Test
/** Test de la methode add avec deux elements */
public void add2() {
   list.add("premier");
   list.add("second");
   assertTrue(list.size() == 2);
   assertTrue(list.item() == "second");
}

@Test(expected = EmptyListException.class)
/** Test de la methode remove sur une liste vide */
public void removeWithEmptyList() {
   list.remove();
}
```

Suite de tests

- Utilisation des annotations
- Classe vide annotée `@RunWith(Suite.class)` pour utiliser le *runner* par défaut `org.junit.runners.Suite`
- `@RunWith(Class)` pour changer de *runner*
- `@SuiteClasses(Class[])` pour indiquer comment former la suite de tests

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

@RunWith(Suite.class)
@SuiteClasses({TestStringListV4.class, ...})
public class TestAllV4 { }
```

Exemple d'exécution en mode console

- Le *runner* par défaut exécute tous les tests (non ignorés) de la classe

```
>java -cp .;junit.jar org.junit.runner.JUnitCore TestStringList
JUnit version 4.11
.I.
Time: 0.057

OK (6 tests)
```

Compatibilité ascendante et descendante

- JUnit 4 peut exécuter directement des tests JUnit 3.8
- Des tests JUnit 4 peuvent être exécutés avec JUnit 3.8
  - Adaptateur `junit.framework.JUnit4TestAdapter`
  - Ajouter dans chaque classe de test

```
public static junit.framework.Test suite() {
  return new JUnit4TestAdapter(TestStringListV4.class);
}
```

Anti-patterns de test

- Assertions manquantes
- Assertions multiples
  - Pas une bonne idée
  - Les tests doivent rester courts
    - ★ Plus facile à comprendre
    - ★ Plus facile d'identifier les causes des erreurs détectées par les tests
    - ★ Moins de risque d'erreur dans les tests
  - Utiliser `setUp` si besoin de partage de code entre parties à tester
- Utilisation du mauvais `assert`
- Test trop compliqué
  - Il ne faut pas avoir à tester le test pour se convaincre qu'il est bon
- Dépendances externes
  - Éviter les dépendances vers l'environnement
    - ★ SGBD, fichiers, bibliothèques, *socket*, *etc.*
  - Il ne faut qu'il y ait des éléments trop compliqués à mettre en place
    - ★ Sinon les tests ne seront pas exécutés

Eclipse

- JUnit est bien intégré à Eclipse
- Pour créer un test : `New...` puis choisir `JUnit Test Case`
  - Choisir la version de JUnit
  - La version utilisée de JUnit peut être modifiée en allant dans l'onglet `libraries` des propriétés du projet
- Pour exécuter un test (ou une suite de tests) : `Run as...` puis choisir `JUnit`
- Organisation du projet
  - `bin` (exécutables)
  - `src` (sources) contenant des *packages*
  - `test` contenant les mêmes *packages*
- Cela permet de livrer ou non les tests et évite les imports dans les tests

Couverture de code avec EclEmma

- Un outil de couverture de code permet de vérifier dynamiquement (*i.e.*, à l'exécution) quelles parties du code source à tester ont effectivement été exécutées (*i.e.*, *couvertes* par le(s) test(s))
- Installation : *Help > Eclipse Marketplace...*
- Comment ça marche : il instrumente le *bytecode* chargé au lancement de la JVM et génère les informations de couverture de code
  - *Coverage Configurations* : sélectionner seulement le dossier `src`
  - *Coverage View* : taux de couverture
  - Annotations dans le code source : couverture complète, couverture partielle et non couvert
- Quels sont les critère de couverture proposés par EclEmma ?