

Le test logiciel

Matthias BRUN
Camille CONSTANT

10 janvier 2018

ESEO – DIS



La section sur le génie logiciel est inspirée des enseignements de :

- Olivier CAMP (Groupe ESEO, Angers) et
- Jérôme DELATOUR (Groupe ESEO, Angers).

Les ouvrages de référence pour le test logiciel sont donnés en annexe de cette présentation.

Plan

- 1 Contexte : le génie logiciel
- 2 Introduction au test logiciel
- 3 Mise en œuvre d'une activité de test
- 4 Techniques de vérification & validation
- 5 Annexes

Plan

1 Contexte : le génie logiciel

- Motivation
- Le génie logiciel
- Cycles de développement

Plan

1 Contexte : le génie logiciel

- Motivation
- Le génie logiciel
- Cycles de développement

Constat

Progression continue :

- du matériel informatique ;
- des besoins logiciels.

« Lorsqu'il n'y avait pas de machine, la programmation n'était pas un problème ; lorsqu'on a commencé à avoir quelques petits ordinateurs, la programmation est devenue un léger problème et maintenant que nous avons des ordinateurs énormes, la programmation est devenue un problème aussi énorme.

L'industrie de l'électronique n'a, dans cette mesure, résolu aucun problème, elle n'a fait qu'en créer – elle a créé le problème d'utiliser ses produits. »

Edsger W. Dijkstra, *Turing Award Lecture Notes*, 1972.

Constat

Progression continue :

- du matériel informatique ;
- des besoins logiciels.

Exemple de système complexe :

- Le système de navigation et d'attaque d'un *Rafale*
 - 2 millions de lignes de code en ADA ;
 - 50 équipements (numériques ou analogiques) ;
 - 12 calculateurs ;
 - Combinatoire des modes et des états élevée :
 - *Landing System* : plus de 10^{14} états possibles ;
 - *Side Displays Management* : plus de 2.10^8 états possibles.

Constat

Progression continue :

- du matériel informatique ;
- des besoins logiciels.

Symptômes :

- Coût de développement quasi-impossible à prévoir
(→ dépassement moyen de 70% du coût) ;
- Délai de livraison rarement respecté
(→ dépassement moyen de 50% du délai) ;
- Qualité des logiciels souvent déficiente
(→ 70% ne satisfont pas les besoins utilisateurs) ;
- Maintenance du logiciel coûteuse et difficile
(→ 60% du budget total et à l'origine de nouvelles erreurs).

Quelques chiffres

Des « origines » :

Logiciels payés mais jamais livrés	29%
Logiciels livrés mais jamais utilisés ou abandonnés	66%
Logiciels utilisables après corrections	3%
Logiciels livrés et utilisables en l'état	2%

source : GAO Survey, 1982.

... à « nos jours » :

Logiciels non livrés	23%
Logiciels livrés trop tard ou hors budget ou incomplets	49%
Logiciels livrés et utilisables en l'état	28%

source : Standish group, 2009.

Quelques chiffres

Corrélations de tailles...

Projet	Personnes	Durée	Succès
< 750 K\$	6	6 mois	55%
750 K\$ - 1,5 M\$	12	9 mois	33%
1,5 - 3 M\$	25	12 mois	25%
3 - 6 M\$	40	18 mois	15%
6 - 10 M\$	> 250	> 24 mois	8%

Quelques chiffres

Corrélations de tailles...

Entreprise	Surcoût	Temps (dep.)	Fonctionnalités
Grande	178%	6 mois	45%
Moyenne	182%	9 mois	66%
Petite	214%	12 mois	75%

« *The first 90 percent of the code accounts for the first 90 percent of the development time. The remaining 10 percent of the code accounts for the other 90 percent of the development time.* »

Tom Cargill, *The ninety-ninety rule*, Bell Labs.

Quelques anecdotes

Cocasses...

- Bug de l'an 1900
 - en 1992, dans le Minnesota, une femme de 104 ans reçoit une proposition pour un jardin d'enfant ;
- Bug de l'année bissextile
 - en 1988, vente de produits périmés d'un jour, le 29 février ;
- Bug de l'« utilisation »
 - le 10 avril 1990, un métro londonien est parti sans son conducteur (porte bloquée au démarrage...).

Quelques anecdotes

...et coûteuses.

- *Mariner I*, NASA, 1962 (coût > 100 millions de \$)
 - première sonde du programme *Mariner* (survol de Vénus), déviation imprévue de la trajectoire → ordre de destruction.
 - spécification :
 - lissage d'une dérivation de variable (\bar{a}) non transcrise
 - dérivée d'une variable (\dot{a})
 - réaction aux variations mineures comme aux écarts importants.
- City (4 ans de développement, 4 milliards de livres de perte) ;
- C-17 de McDonnell Douglas (80 μ p, 19 ordinateurs, 6 langages de programmation, 500 millions de \$ de dépassement) ;
- etc. (à suivre)

Quelques erreurs (classiques)

Les erreurs du programmeur (débutant) :

- Ne pas évaluer la qualité d'un logiciel (ça compile \Rightarrow c'est livrable) ;
- Ne pas évaluer la qualité d'un logiciel tant qu'il ne tourne pas (\rightarrow revue de code, métriques, etc.) ;
- Arrêt du travail quand le programme est livré
(\rightarrow maintenance, évolutions, etc.) ;
- Suffisance du code (\rightarrow documentation, gestion de versions, etc.) ;

Les erreurs du chef de projet (débutant) :

- Ajouter des programmeurs pour rattraper un retard
(projet évalué à 4 hommes/mois irréalisable en un 1 jour par 120 développeurs).

« Adding manpower to a late software project makes it later. »

Brooks Law, Turing Award, 1999.

Bilan

Complexité du logiciel

- de nombreuses fonctionnalités ;
- des objectifs parfois contradictoires (SdF vs. coûts) ;
- de nombreux acteurs ;
- produit immatériel, difficile à visualiser ;
- difficilement compréhensible par une seule personne.

Du point de vue du client, un logiciel doit :

- répondre à ses besoins ;
- être utilisé avec satisfaction dans toutes ses fonctionnalités ;
- être maintenable et évolutif ;
- être développé dans les limites du budget et des délais.

Plan

1 Contexte : le génie logiciel

- Motivation
- Le génie logiciel
- Cycles de développement

Le génie logiciel

Historique

- Les premiers logiciels étaient petits et développés par des programmeurs, souvent seuls, motivés et talentueux.
- Les premiers problèmes sont apparus dans les années 1960 avec des logiciels plus conséquents.
- Le terme « génie logiciel » (*software engineering*) est apparu en 1968/1969 lors d'une conférence de l'OTAN où il était question de la crise du logiciel.

La crise du logiciel

- Le coût du logiciel dépasse celui du matériel.
- Prise de conscience de la non-qualité des logiciels produits.
- Risques humains et économiques importants.

Le génie logiciel

Définitions

- Le génie logiciel permet *d'automatiser et de systématiser* la traduction de la pensée humaine en code machine.
- Le génie logiciel offre *des méthodes* afin de produire du logiciel *de qualité en respectant les contraintes de temps et de coût*.
- Le génie logiciel vise à *structurer et organiser* l'ensemble des activités liées à la réalisation de logiciels et à promouvoir *des niveaux de qualité croissants*.
- Le génie logiciel est *un ensemble d'outils, de méthodes et de techniques* visant à rationaliser la production du logiciel.

Le génie logiciel

Préoccupations

Le génie logiciel s'intéresse aux **procédés de fabrication** des logiciels afin de s'assurer que :

- les **fonctionnalités** répondent aux besoins des utilisateurs ;
- la **qualité** correspond au contrat initial ;
- les **délais** restent dans les limites prévues ;
- le **coût** reste dans les limites prévues.

(→ CQFD : Coût Qualité Fonctionnalités Délais)

La qualité

La qualité est l'ensemble des propriétés et caractéristiques d'un produit ou service qui lui confèrent l'aptitude à satisfaire des besoins exprimés ou implicites (conformité aux exigences).

L'assurance qualité est l'ensemble des actions préétablies et systématiques nécessaires pour donner la confiance appropriée en ce qu'un produit ou service satisfera aux exigences données à la qualité.

Le contrôle qualité est l'action de mesurer, examiner, essayer, passer au calibre une ou plusieurs caractéristiques d'un produit ou service et de les comparer aux exigences spécifiées en vue d'établir leur conformité.

ISO, *International Organization for Standardization*, ISO-8402.

La qualité

Exemple de démarche qualité

Cause d'échec	Voie de solution ^(*)
Surestimation de savoir-faire	Étude de faisabilité
Trop de confiance	Contrôle de processus
Mauvaise organisation	Respect de normes
Mauvaise spécification	Prototypage
Modifications excessives	Gestion de versions
Validation inadéquate	Test

(*) voies de solution non-exhaustives.

La qualité

Préoccupations (sûreté de fonctionnement - *dependability*)

Fiabilité (*reliability*)

Aptitude à accomplir une fonction requise pendant une durée donnée.

Disponibilité (*availability*)

Aptitude à être en état d'accomplir une fonction requise à un instant donné.

Maintenabilité (*maintainability*)

Aptitude à être maintenue ou rétablie dans un état dans lequel une fonction requise peut être accomplie.

Sécurité (*safety*)

Aptitude à éviter de faire apparaître des évènements critiques ou catastrophiques.

La qualité

Préoccupations (qualité de service - *quality of service* (QoS))

Durabilité (*durability*)

Aptitude d'une entité à demeurer en état d'accomplir une fonction requise dans des conditions données d'utilisation et de maintenance jusqu'à ce qu'un état limite soit atteint (exemple : fin de vie).

Continuabilité (*retainability*)

Aptitude d'un service, une fois obtenu, à continuer d'être fourni dans des conditions données et pendant la durée voulue.

Servibilité (*serveability*)

Aptitude d'un service à être obtenu à la demande d'un usager et à continuer d'être fourni pendant la durée voulue, avec des tolérances spécifiées et dans des conditions données.

La qualité

Préoccupations (sécurité - *security*)

Intégrité (*integrity*)

Aptitude à se protéger des altérations indésirées de l'information.

Confidentialité (*privacy*)

Aptitude à interdire la divulgation d'informations confidentielles.

Sécurité (*security*)

Aptitude à éviter les accès non autorisés.

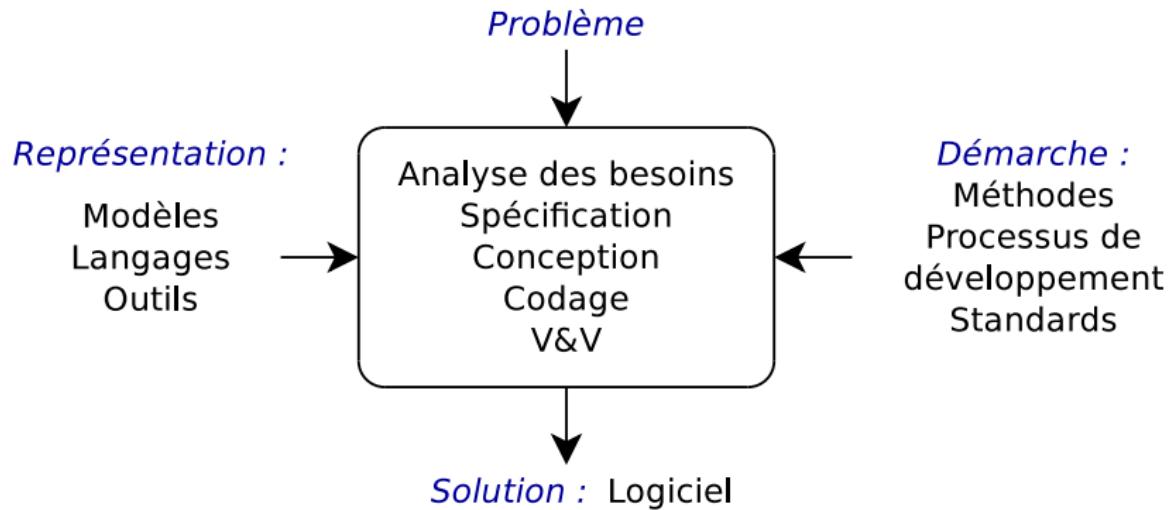
Préoccupations (test, etc.)

Testabilité (*testability*)

Aptitude à être testé et à localiser les fautes.

Langage de la qualité : norme IEC/CEI 9126, ISO 25000.

Démarche du génie logiciel



Phases de développement

Analyse des besoins

- dialogue avec le client (→ compréhension du problème) ;
- étude du domaine (→ besoins techniques, contraintes économiques).

Répond à la question : Pourquoi ?

Entrées : données fournies par les experts métier et les utilisateurs.

Méthodes : entretiens, questionnaires, étude de l'existant, etc.

Résultat : **cahier des charges** décrivant

- les qualités fonctionnelles attendues (en terme de services offerts) ;
- les qualités non fonctionnelles attendues (contrainte temporelle, facilité d'utilisation, etc.).

Phases de développement

Spécification

- formalisation des données, des fonctionnalités et des contraintes.

Répond à la question : Quoi ?

Entrées : cahier des charges, considérations techniques et faisabilité.

Méthodes : SADT, SART, MERISE, UML, etc.

Résultat : **document de spécification** décrivant

- le modèle du domaine (vocabulaire, contexte, frontière, acteurs) ;
- le modèle conceptuel (déploiement, fonctionnalités : CU, IHM, etc.).

Phases de développement

Conception

- description de l'architecture du logiciel ;
- définition de la structure du logiciel.

Répond à la question : Comment ?

Entrées : document de spécification, technologies de mise en œuvre.

Méthodes : MACH, MERISE, UML, etc.

Résultat : **document de conception** décrivant

- la conception générale (architecture, composants, relations, méthodes, comportement, contrats) ;
- la conception détaillée (classes, algorithmes, structures de données, MVC, entrées/sorties, persistances, communications, protocoles, multitâches, etc.).

Phases de développement

Codage

- programmation du logiciel.

Entrées : document de conception, environnement programmation.

Méthodes : programmation (génération automatique possible), standards de codage.

Résultat : **document d'implémentation** donnant

- le code source ;
- la documentation (code, installation, dépendances).

Livrable → code compilé pour un environnement donné.

Phases de développement

Validation & Vérification

V&V (ISO-9000-3)

Processus d'évaluation du logiciel pour s'assurer qu'il satisfait aux exigences spécifiées.

Validation : Est-ce que le logiciel réalise les fonctions attendues ?
→ faire le bon logiciel (*you built the right thing*)

Vérification : Est-ce que le logiciel fonctionne correctement ?
→ faire bien le logiciel (*you built it right*)

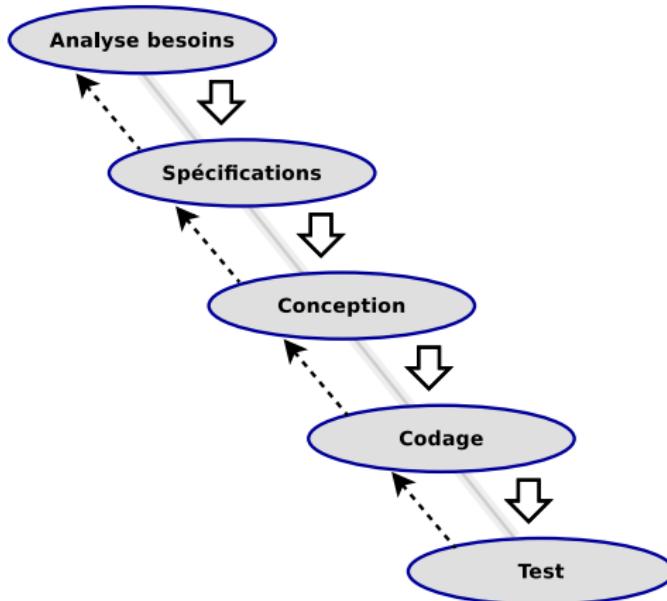
Méthodes : vérification formelle, test logiciel.

Plan

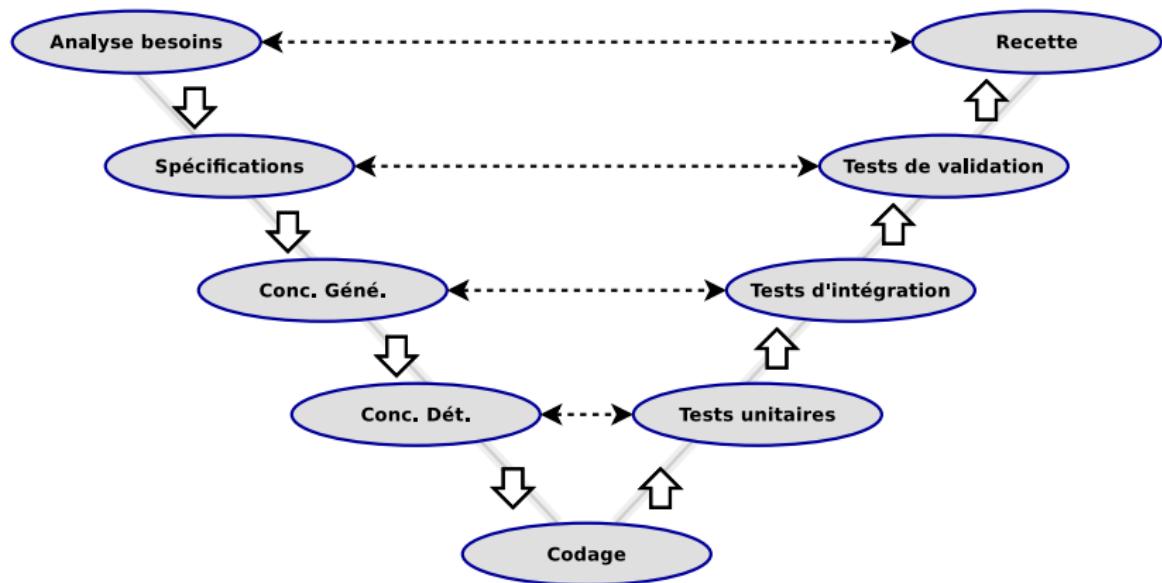
1 Contexte : le génie logiciel

- Motivation
- Le génie logiciel
- Cycles de développement

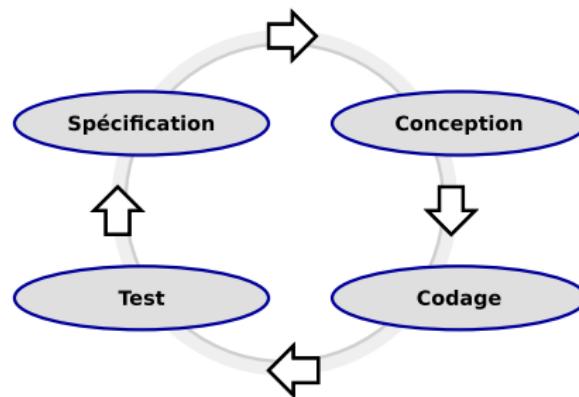
Modèle en cascade



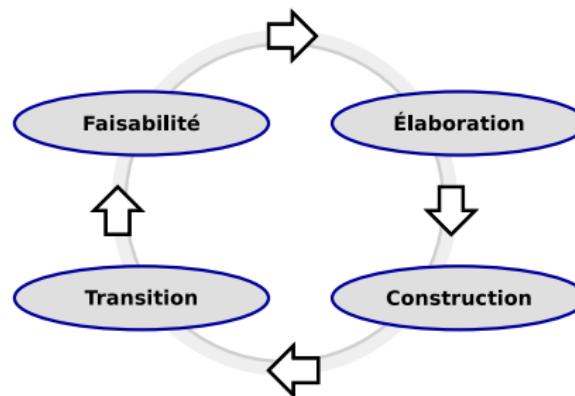
Modèle en V



Modèle itératif et incrémental



Modèle itératif et incrémental



« When to use iterative development ? »

« You should use iterative development only on projects that you want to succeed. »

Martin Fowler, dans *UML Distilled*.

Plan

2 Introduction au test logiciel

- Motivation
- Principe
- Types de test

Plan

2 Introduction au test logiciel

- Motivation
- Principe
- Types de test

Faute, Erreur, Défaillance

Défaillance (*failure*)

Cessation de l'aptitude d'une entité à accomplir une fonction requise.

IEC/CEI-271-1974, *International Electrotechnical Commission*.

Fautes, Erreur, Défaillance

Défaillance du système : le service délivré diffère de celui spécifié (attendu).

Erreur : partie du système anormale, à l'origine de la défaillance.

Faute : cause phénoménologique de l'erreur.

J.C. LAPRIE, *Guide de la sûreté de fonctionnement*, édition Cépadues, 1997.

Terminologie anglaise (IEEE 729)

Failure ~ Défaillance, Fault ~ Erreur, Error ~ Faute.

Faute, Erreur, Défaillance

Illustration des notions d'erreur et de défaillance :

```
int my_max(int t[], int size) {  
    int icour = 0; int imax = 0;  
    while( icour < size )  
    {  
        if ( t[icour] > t[imax] ) imax = t[icour];  
        icour++;  
    }  
    return t[imax];  
}
```

Faute, Erreur, Défaillance

Chaîne causale



Les coûts

Les **coûts** d'une défaillance

- Mariner I, Ariane 5, Patriot, Réseau AT&T...
- An 2000, coût 178 milliards de \$ (deux fois plus qu'en 1998).

Objectif

Anticiper les défaillances ← **déetecter les erreurs ou les fautes**
→ corriger les fautes.

Classification des erreurs en fonction de :

- leur nuisance (Beizer, 1984)
Mild, Moderate, Annoying, Disturbing, Serious, Very serious, Extreme, Intolerable, Catastrophic, Infectious.
- leur fréquence d'occurrence (Jorgensen, 2002)
One time only, Intermittent, Recurring, Repeatable.
- leur situation dans le cycle de développement (unitaire, intégration, etc.).
- ...

Les coûts de correction

Les **coûts** de correction d'une faute

<i>... lors d'une phase de :</i>	<i>Coût :</i>	
Spécification	1	● L'essentiel des fautes sont des fautes de traduction .
Conception	3 à 6	● 60% introduites en phase «amont» (lors de l'analyse du problème/des spécifications).
Réalisation	10	
Test	15 à 40	● 50% découvertes en exploitation (beaucoup de défaillances différentes pour une même faute).
Validation	30 à 70	
Production	1000	

Conclusion

Mieux vaut prévenir que guérir (*better safe than sorry*).

Plan

2 Introduction au test logiciel

- Motivation
- **Principe**
- Types de test

Validation & Vérification

V&V (ISO-9000-3)

Processus d'évaluation du logiciel pour s'assurer qu'il satisfait aux exigences spécifiées.

Validation :

Est-ce que le logiciel réalise les fonctions attendues ?

→ faire le bon logiciel (*you built the right thing*)

Vérification :

Est-ce que le logiciel fonctionne correctement ?

→ faire bien le logiciel (*you built it right*)

Validation & Vérification

Méthodes de V & V :

- **Vérification formelle** (analyse statique) :
Preuve formelle ou *model-checking* pour vérifier des propriétés.
- **Test statique** (analyse statique) :
Analyse de code, spécification ou conception pour détecter des fautes.
- **Test dynamique** (analyse dynamique) :
Exécution d'un programme pour détecter des erreurs.
- **Mesures de complexité** (analyse statique) :
Analyse de code pour apporter des métriques.

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

Vérification formelle

Principe

- Vérifier une propriété sur un modèle ($M \models \varphi$).

Avantages

- exhaustivité (propriété vérifiée sur tout le modèle) ;
- génération d'un contre-exemple si propriété violée (si possible).

Inconvénients

- nécessite un modèle formel du système ;
- vérification partielle/approchée
(complexité ou non-décidabilité du problème de vérification).

Test statique et dynamique

Principe

Déetecter un maximum :

- de fautes lors de l'analyse du système (analyse statique) ;
- d'erreurs à l'exécution du système (analyse dynamique).

Avantages

- test sur implémentation ou système réel ;
- modélisation partielle possible.

Inconvénients

- non-exhaustivité : correction du système non prouvée ;
- confiance en l'oracle ou la spécification.

Mesures de complexité

Principe

- Apporter des métriques sur l'implémentation du système.

Avantages

- indicateurs de qualité ;
- orientation de l'effort de V&V.

Inconvénients

- identification des indicateurs pertinents ;
- interprétation des métriques ;
- non suffisant (nécessite activité de V&V en complément).

Définitions du test

G. Myers, *The Art of Software Testing*, 1979

Tester c'est exécuter le programme dans l'intention d'y trouver des anomalies ou des défauts.

IEEE 729 (*Standard Glossary of Software Engineering Terminology*)

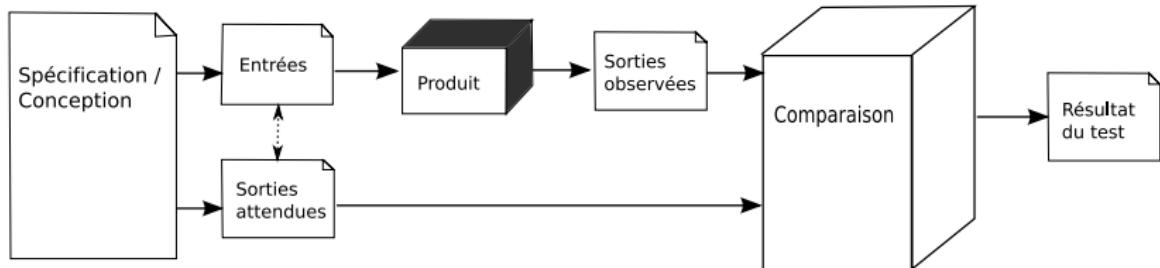
Le test est l'exécution ou l'évaluation d'un système ou d'un composant par des moyens automatiques ou manuels, pour :

- vérifier qu'il répond à ses spécifications ou
- identifier les différences entre les résultats attendus et les résultats obtenus.

Testing can reveal the presence of errors but never their absence.

Edsger W. Dijkstra, *Notes on structured programming*, AcademicPress, 1972.

Principe général du test



Processus de test

- 1 Élaboration des entrées (+ préconditions) et des sorties attendues.
- 2 Exécution du test (génération des sorties observées).
- 3 Observation des résultats (comparaison sorties attendues/observées).

Génération des jeux de tests

Données de Test (DT)

→ entrées du programme à tester.

Jeux de Test (JT)

→ ensemble des DT.

Pour la soumission des JT, un certain état du système doit souvent être atteint (pré-condition), ce qui implique généralement une certaine séquence d'actions à effectuer avant de soumettre les DT.

Scénario de Test (ou Cas de Test - *Test Case*)

→ regroupe deux séquences d'actions :

- une à effectuer pour satisfaire à une pré-condition ;
- une à effectuer pour utiliser les DT d'un JT.

Génération des jeux de tests

Exemple

Soit P un programme additionnant deux entiers entre 0 et 10.

L'ensemble des DT est $\{(0, 0), (0, 1), (0, 2), \dots (1, 10), \dots (10, 10)\}$

Exercice 2.1

Soit P' un programme additionnant deux nombres de 32 bits.

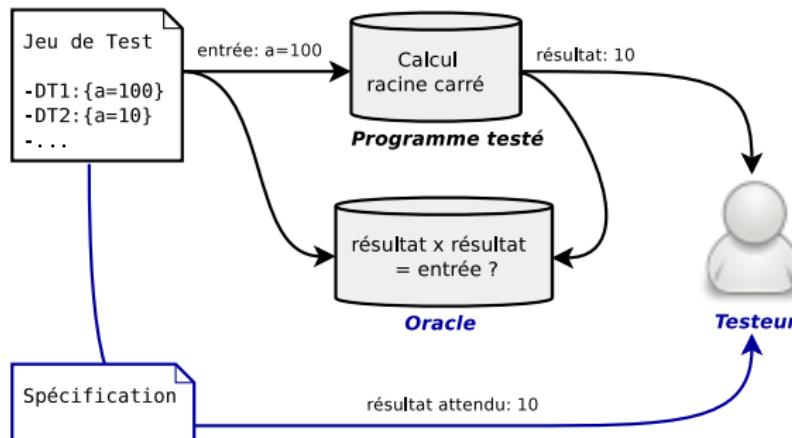
En supposant que l'on puisse exécuter 100 000 tests/seconde, combien faut-il de temps pour effectuer un test exhaustif ?

- ⇒ Le test exhaustif est en général impossible à réaliser.
- ⇒ Le test est une méthode de vérification partielle de logiciels.
- ⇒ La qualité du test dépend de la pertinence du choix des DT.

Observation des résultats

Objectif : l'entité testée fournit-elle le bon résultat ?

Observation : par un humain ou un automate (*oracle*).



Observation des résultats : oracle

Pour chaque test élémentaire t et pour un programme P , l'oracle O donne une des trois réponses suivantes :

- positif $\Rightarrow P$ satisfait t
- négatif $\Rightarrow P$ ne satisfait pas t
- indécidable $\Rightarrow P$ ne termine pas (par exemple)

Les notions liées aux oracles ne seront pas plus développées dans ce cours.

Le test logiciel

En résumé :

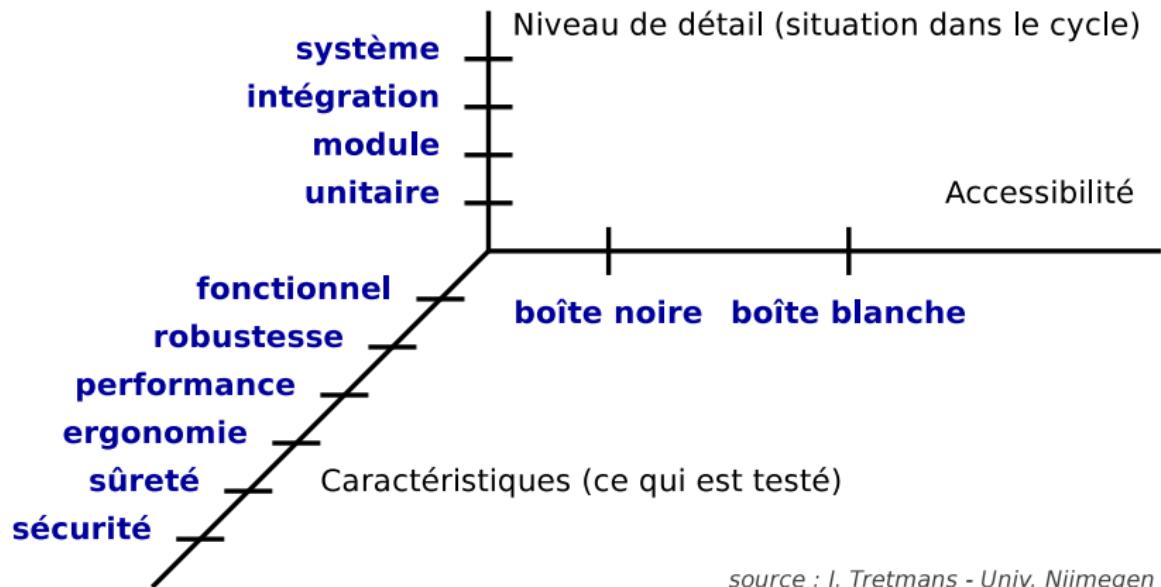
- Le test est l'une des activités de l'assurance qualité.
- Le test est une méthode de vérification et validation d'un programme.
- Tester, c'est seulement détecter des erreurs, ce n'est pas les corriger, ni diagnostiquer les causes.
- Le test n'a pas pour objectif de prouver qu'un programme est correct.
- Le test exhaustif est en général impossible à réaliser.
- La pertinence des tests repose sur l'élaboration des données de test.

Plan

2 Introduction au test logiciel

- Motivation
- Principe
- Types de test

Types de test



Niveau de détail : unitaire

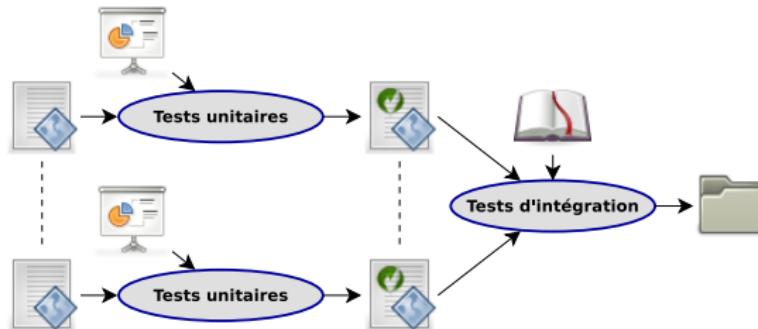


Le test unitaire vise à :

- contrôler la fiabilité des unités logicielles développées ;
- vérifier le respect de leurs conceptions ;
- identifier les erreurs logiques.

Exemples : tests de modules, de procédures, de classes, de méthodes.

Niveau de détail : intégration



Le test d'intégration vise à :

- démontrer la stabilité et la cohérence des interfaces et des interactions des unités logicielles ;
- vérifier le respect de la conception générale.

Exemple : tests de la composition de classes, de packages, de composants.

Niveau de détail : intégration

Les différentes approches d'intégration :

- intégration globale ;
- intégration continue ;
- intégration progressive incrémentale.

Niveau de détail : intégration

Les différentes approches d'intégration :

- intégration globale :
 - adaptée aux petites applications ;
 - modules peu dépendants ;
 - pas de stratégie, pas de planification.
- intégration continue
- intégration progressive incrémentale

Niveau de détail : intégration

Les différentes approches d'intégration :

- intégration globale
- intégration continue ;
- intégration progressive incrémentale.

Doublures : programmes de substitution

Lanceur (moniteur, simulateur, *driver*) :

fournit des données d'entrée à l'unité logicielle testée.

Bouchon (muet, *stub*, *mock*) :

récupère ou échange des données avec l'unité logicielle testée.

Niveau de détail : intégration

Les différentes approches d'intégration :

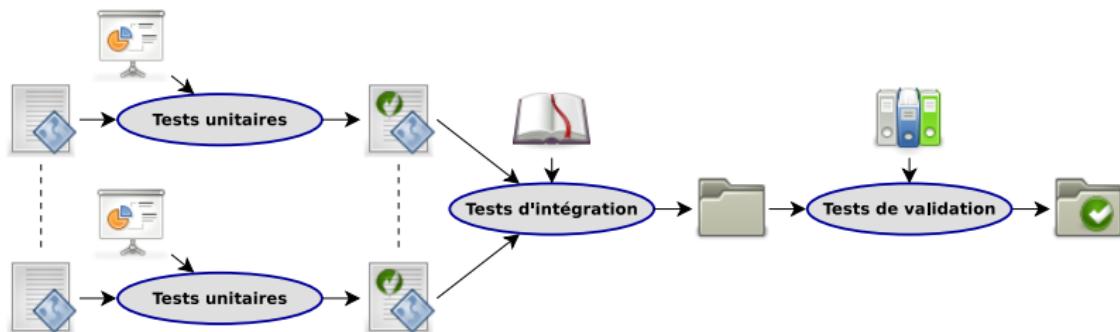
- intégration globale
- intégration continue :
 - intégration « au fur et à mesure » ;
 - modules peu dépendants ;
 - modules intégrés dès que disponibles ;
 - pas de planification des bouchons et des lanceurs.
- intégration progressive incrémentale

Niveau de détail : intégration

Les différentes approches d'intégration :

- intégration globale
- intégration continue
- intégration progressive incrémentale :
 - ordre d'intégration pré-spécifié ;
 - planification des bouchons et des lanceurs ;
 - intégration :
 - ascendante (min. bouchons/max. lanceurs) ;
→ disponibilité des entrées/sorties.
 - descendante (max. bouchons/min. lanceurs) ;
→ « squelette » du programme validé tôt.
 - guidée par les risques, les coûts, mixte.

Niveau de détail : système (validation)



Le test de validation (ou test système) vise à :

- valider l'adéquation aux spécifications du logiciel.
 - validation fonctionnelle : implantation correcte des exigences clients ;
 - validation solution : comportement correct vis-à-vis des cas d'utilisation.

Caractéristiques

Test nominal (*test-to-pass*)

Les cas de test correspondent à des données d'entrée valide.

Test de robustesse (*test-to-fail*)

Les cas de test correspondent à des données d'entrée invalide.

→ Tests nominaux effectués avant les tests de robustesse.

Tests de fonctionnalité

Les cas de test correspondent à des comportements attendus
(ex : test de conformité).

Test de performance (temps de réponse, stabilité)

→ test avec montée en charge (*load testing*).
→ test avec demandes de ressources anormales (*stress testing*).

Etc. (ergonomie, sûreté, sécurité...)

Caractéristiques

Préoccupations des tests

<i>Functionality</i> (fonctionnalité)	<i>Usability</i> (utilisation)	<i>Reliability</i> (fiabilité)	<i>Performance</i> (performance)	<i>Service</i> (service)	<i>Evolution</i> (évolution)
Fonctions	Compréhension	Maturité	Temps	Analyse	Adaptation
Résultats	Apprentissage	Tolérance	Comportement	Stabilité	Modification
Interopérabilité	Ergonomie	Réparation	Ressources	Tests	Cohabitation
etc.	etc.	etc.	etc.	etc.	etc.
<i>exigences non-fonctionnelles</i>					

Modèle *FURPSE*, ISO-9126.

Accessibilité

Boîte noire (*black box testing*)

Code source inaccessible, seule l'interface est accessible
(contrôle des entrées, observation des sorties).

Test fonctionnel

- Test de validation par rapport à la spécification.
- DT générées suivant les entrées possibles du système :
 - spécf. informelle (ex : langage naturel) sélection manuelle des DT ;
 - spécf. semi-formelle (ex : UML) sélection des DT guidée par la modélisation ;
 - spécf. formelle (ex : SDL) sélection des DT automatique envisageable.
- Vision globale.

Accessibilité

Boîte blanche (*white box testing*)

Code source accessible.

Test **structurel**

- DT générées à partir d'une analyse du code source.
- Vision locale.

- **statique** : test réalisé lors de l'examen du code source.
- **dynamique** : test réalisé lors de l'exécution du code source.

Accessibilité

Les tests fonctionnels et structurels sont utilisés de façon complémentaire.

Approche fonctionnelle :

- détecte plus facilement les fautes d'omission et de spécification,
- localise difficilement les fautes.

Approche structurelle :

- localise plus facilement les fautes commises,
- incapable de détecter des fonctions manquantes.

Examiner ce qui a été fait ne prend pas forcément en compte ce qui aurait dû être fait.

Non-régression

Tests de non-régression

Tests permettant de vérifier que les *corrections* ou *évolutions* effectuées sur le code n'ont créé aucune nouvelle anomalie.

- peuvent être automatisés.
- effectués suivant certains critères, basés par exemple :
 - sur les statistiques ;
 - sur le choix des fonctionnalités essentielles (selon criticité, modifications apportées, etc.) ;
 - sur les risques (selon stratégie de test).

Plan

- 3 Mise en œuvre d'une activité de test
- Cycle de développement
 - Plan de test
 - Cahier de test
 - Outilage

Plan

- 3 Mise en œuvre d'une activité de test
- Cycle de développement
 - Plan de test
 - Cahier de test
 - Outilage

Intégration du test dans le cycle de développement

La mauvaise manière de voir le test...

- test uniquement vu comme une exécution,
- peu de planification,
- peu d'attention à la sélection des DT.

...et la bonne :

Les activités liées au test

- interagissent avec toutes les activités de développement ;
- se déroulent tout au long du cycle de développement ;
- s'appuient sur des analyses pour élaborer des DT.

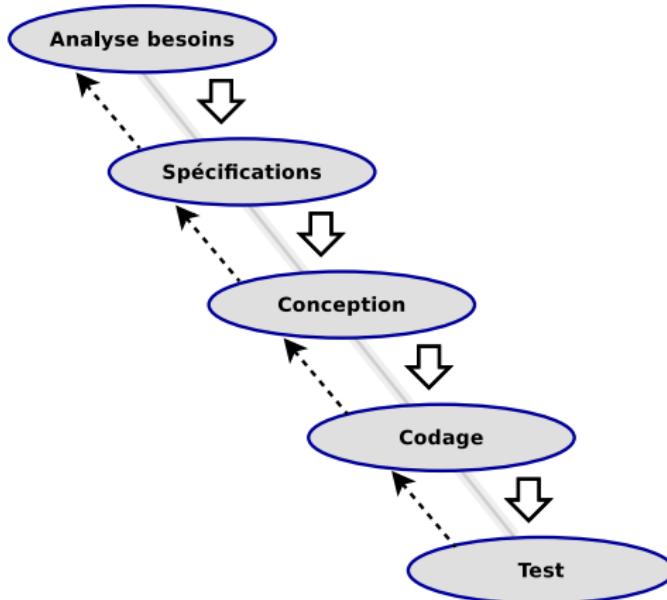
Difficultés du test dans le cycle de développement

Difficultés d'ordre psychologique ou « culturel »

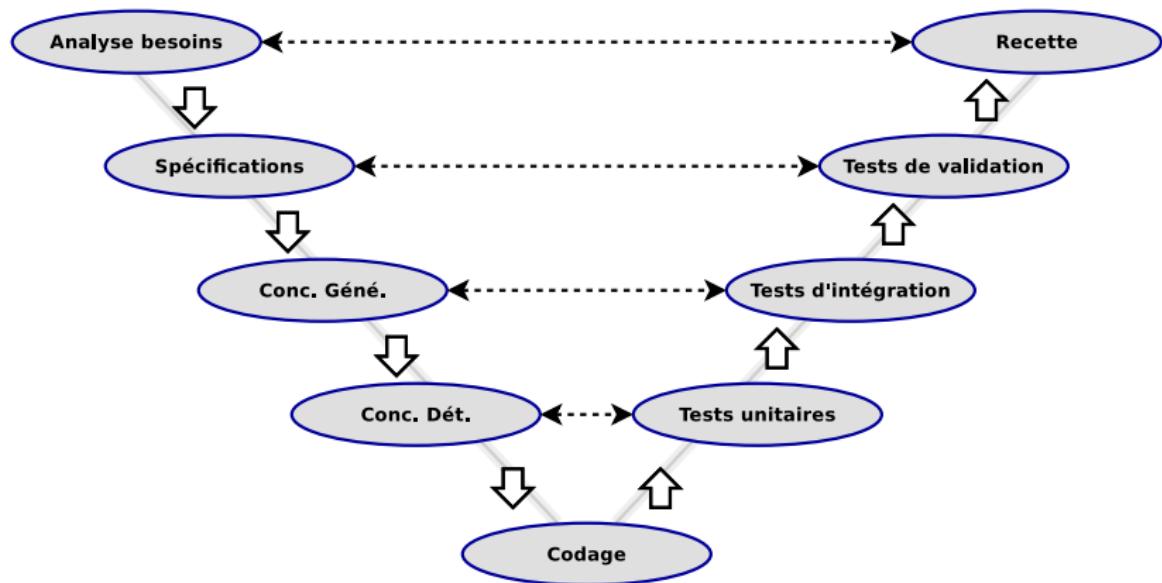
- Le test est un processus destructif :
un bon test est un test qui trouve une erreur
(l'activité de conception/programmation est un processus constructif)
- Les fautes peuvent être dues à des incompréhensions de spécifications ou de mauvais choix d'implantation.

L'activité de test s'inscrit dans le contrôle qualité, *en parallèle et en soutien* de la spécification/conception.

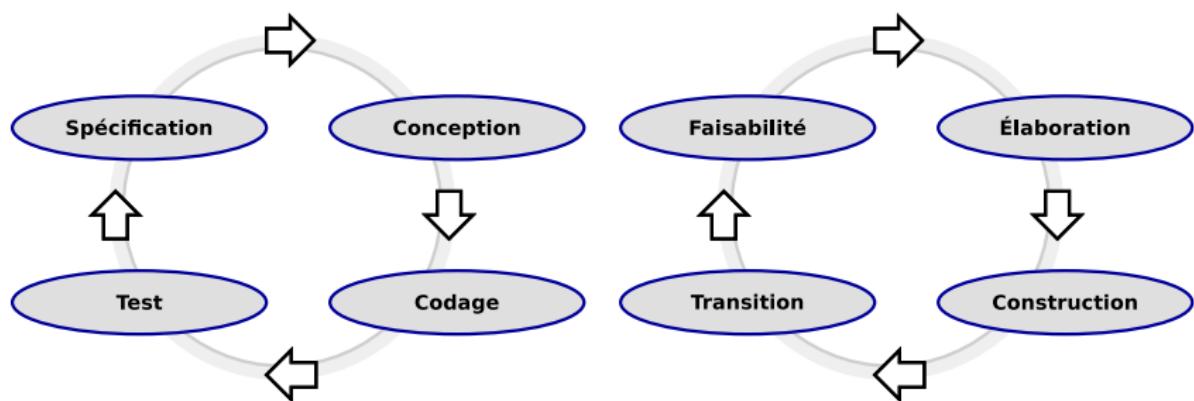
Intégration dans les cycles de développement (cf. Contexte)



Intégration dans les cycles de développement (cf. Contexte)



Intégration dans les cycles de développement (cf. Contexte)



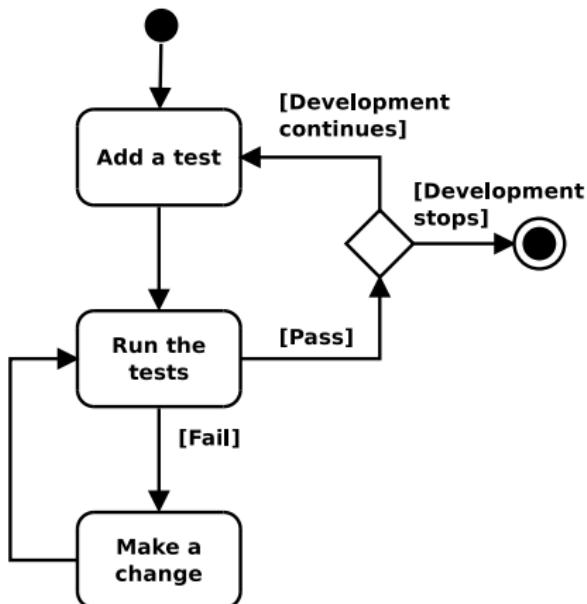
Développement dirigé par les tests

Test driven development

- ➊ Développer un test ;
- ➋ Développer le code correspondant ;
- ➌ Remanier le code tant que le test ne passe pas.

Utilisation des outils d'automatisation d'exécution des tests unitaires.

→ <http://www.agiledata.org/>



Développement dirigé par les tests

Méthodologie :

- ① identifier ce que doit faire le code à développer (ou partie de code) ;
- ② écrire un code de test unitaire avec un *Unit*
(pour valider le code à développer) ;
- ③ écrire le code à développer ;
- ④ exécuter ce code via le code de test ;
- ⑤ constater si le test unitaire passe ou ne passe pas.

Motivations :

- centrer la programmation sur les fonctionnalités attendues ;
- constater les fautes au plus tôt ;
- assurer une non-régression du code ;
- faciliter la confiance dans un code fourni
(indicateurs de validation intégrés au code)
(relecture du code de test plus rapide qu'une relecture du code du programme).

Plan

- 3 Mise en œuvre d'une activité de test
- Cycle de développement
 - Plan de test
 - Cahier de test
 - Outilage

Plan de test : principe

En début de projet

Définition d'un **Plan de Test** (*test plan*) :



- Objectifs de test ;
- Types de test ;
- Techniques de test ;
- Organisation des tests.

Plan de test : principe

Objectifs de test

- ce qui doit être testé (composants, fonctionnalités).
- contraintes (criticité, normalisation/standardisation, etc.).

Types de test

- TU, TI, TV (+ non-régression) ;
- caractéristiques (nominale, robustesse, etc.) ;

Techniques de test

- pour élaboration des DT (boîte noire / boîte blanche) ;

Organisation des tests

- processus de test ;
- planification des tests ;
- structure des documents à produire (cf. Cahier de Tests).

Plan de test : standard IEEE

IEEE 829-(1998/2008)

- **identifiant** (référence révision)
- **références** (documents mentionnés)
- **introduction** (objet du plan)
- **éléments concernés** (composants...)
- **fonctionnalités concernées**
- **fonctionnalités non-concernées**
- **stratégie** (règles, procédures, etc.)
- **critères de sortie des tests**
- **critères de suspension des tests**
- **documents produits**
- **activités restantes** (complémentaires)
- **environnement** (matériel, logiciel, humain)
- **équipe** (organisation, compétences)
- **responsabilités** (risques, validation, etc.)
- **ordonnancement** (planification)
- **planification des risques**
- **approbateurs** (du plan)
- **glossaire** (termes, acronymes)

→ Versions pour le client et pour l'équipe de test
(objectifs : information ou mise en œuvre).

Plan de test : application sur Monix

(cf. Annexe Monix)

Extrait du document de plan de test (Monix_plan_de_test_v1.0.pdf)
 (plan, extrait synthétique... à compléter)

① Introduction

- a) Objet ; Portée ; Copyright (...)
- b) Termes et abréviations (...)

② Références

Normes et standards : IEEE 829-2008

Documents de spécification : Monix_specification_v1.pdf

Documents de conception : Monix_conception_v1.pdf

Documents de test : Monix_cahier_de_tests_v1.pdf

(...)

③ Périmètre de test

- a) Composants à tester (*monix, morix ?*), à ne pas tester (*BDD, liaison TCP/IP ?*).
- b) Fonctionnalités à tester (*ajouter et enlever des produits à une vente, mise à jour du stock ?*).
- c) Fonctionnalités à ne pas tester (*aspects graphiques ?*).

Plan de test : application sur Monix (cf. Annexe Monix)

Extrait du document de plan de test (Monix_plan_de_test_v1.0.pdf)
 (plan, extrait synthétique... à compléter)

4) Processus et stratégie de test

- a)** Activité (*tests de validation, d'intégration, unitaires ? types de test ?*).
- b)** Techniques (*analyse partitionnelle, limites ?*).
- c)** Outils (JUnit, EasyMock, JMeter ?).
- d)** Critère d'acceptation des tests (*100% de couverture de code ?*).
- e)** Procédures de test (*unitaire : code/test, développement simultané/croisé ?*).
- f)** Gestion des anomalies (*je lève le doigt ?*).

5) Infrastructure de test

(réseau de l'école, station de travail personnelle ?).

6) Documents de test et livrables

(plan de test, cahiers de tests client/entreprise ?).

7) Responsabilités

(client : 0%, développeurs : 0%, testeurs : 100% ?).

8) Équipe de test

(I2-ASTRE : nom, prénom, compétences, rôle ?).

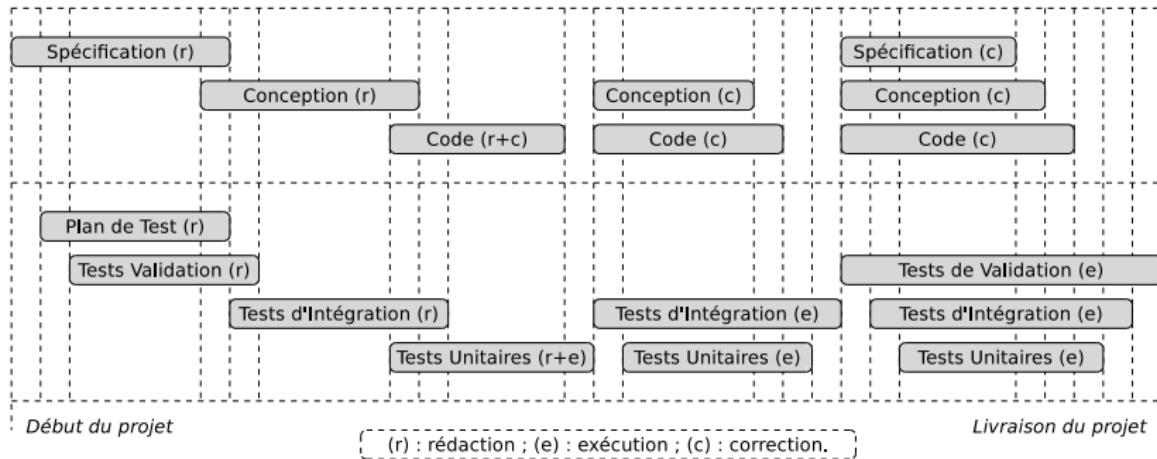
9) Planning prévisionnel

(2 heures ?).

Planification : exemple sur Monix (cf. Annexe Monix)

Extrait du document de planification (Monix_planification_v1.0.pdf)

Planning prévisionnel (vue générale, sur cycle en V) :



Plan

3 Mise en œuvre d'une activité de test

- Cycle de développement
- Plan de test
- Cahier de test**
- Outilage

Cahier de test

Tout au long du projet

Mise en œuvre d'un **Cahier de Test** :



- Tests prévus (*test case*) ;
- Procédures de test (*test procedure*) ;
- Résultats des tests (*test report*).

Pour chaque phase/objectif/préoccupation de test et conformément au plan de test.

Cahier de test

Organisation des fiches de test du cahier de test :

- Partie statique (description du test) ;
- Partie dynamique (capture résultats exécutions du test).

Cahier de test

Organisation des fiches de test du cahier de test :

- Partie statique (description du test) :
 - Référence du test ;
 - Référence de l'exigence, C.U., fonctionnalité, etc. ;
 - Type de test ;
 - Résumé ;
 - Contexte ;
 - Pré-conditions ;
 - Cas de test (scénario) ;
 - Jeux de test (DT et résultats attendus) ;
 - Automatisation mise en place.
- Partie dynamique (capture résultats exécutions du test).

Cahier de test

Organisation des fiches de test du cahier de test :

- Partie statique (description du test) ;
- Partie dynamique (capture résultats exécutions du test) :
 - Version du logiciel ;
 - Version du test ;
 - Testeur ;
 - Dates (prévisions, jeux, rejeux) ;
 - Résultats obtenus (✓, ✗, ?, trace d'exécution) ;
 - Procédure déclenchée.

Cahier de test : application sur Monix (cf. Annexe Monix)

Extrait du document de cahier de test (Monix_cahier_de_test_v1.0.pdf)
(principes synthétiques... à compléter)

① Introduction

- Objet, références, définitions, abréviations.

② Présentation des tests

- Objectifs des tests (*passer le temps ?*).
- Localisation des tests et configurations (*matériels, logiciels, squelettes/bouchons ?*).
- Déroulement des tests (*avec bouchon/squelette, système complet, sur cible ?*).
- Types de jeux de test (*nominaux, robustesse ?*).

③ Description des tests

- Pour chaque phase (validation, intégration, unitaire).
- Tests nominaux, robustesse.
- Réf. ; Résumé ; Pré-conditions ; Jeu ; Résultats attendus (internes/observables).
- Version ; Qui ; Quand (premier, rejeux) ; Résultat obtenu ; Procédure déclenchée.

Cahier de test : scénario de test de validation

Principe :

- À partir des spécifications (exigences, C.U., domaine, IHM, etc.) ;
- Répartition des points des C.U. :
 - Action(s) | Résultat(s) interne(s) | Résultat(s) observable(s).
- Formalisation avec :
 - Composants spécifiés (dans Domaine, IHM, etc.) ;
 - Termes métiers ingénierie (IHM, réseau, BDD, etc.) ;
 - Variables (à renseigner par la suite avec DT).

Intérêts :

- Vérification complétude et cohérence des spécifications ;
- Support pour automatisation des TV ;
- Support pour accompagnement conception ;
- Support pour localisation effort de TI et TU.

Plan

- 3 Mise en œuvre d'une activité de test
- Cycle de développement
 - Plan de test
 - Cahier de test
 - Outilage

Introduction à quelques outils

Dans l'esprit du génie logiciel, le test logiciel se doit d'être outillé.

Premiers outils à destination des testeurs-développeurs :

- Unit : JUnit, CUnit, EmbeddedUnit, TestNG, etc.
- Doubture : EasyMock, Mockito, JMockit, JEeasyTest, etc.
- Sollicitation réseau : JMeter.

Plan

3 Mise en œuvre d'une activité de test

- Cycle de développement
- Plan de test
- Cahier de test
- **Outilage**
 - Unit avec JUnit4
 - Mock avec EasyMock ou Mockito
 - Test serveur avec JMeter

JUnit : principe

Bibliothèque de test unitaire pour le langage Java, créée par Kent BECK et Erich GAMMA.

Deux types de fichiers de tests :

- **TestCase** : classes contenant des méthodes de test.
- **TestSuite** : pour exécuter des *TestCase* déjà définis.
- Sources : `src` et `test` même arborescence.

Lancement des tests :

- **TestRunner** : classes de lancement des tests.

Utilisation :

- Ajout du *jar* de JUnit (`junit.jar` ou `junit4.jar`) dans le *classpath*.

JUnit4 : TestCase

Création d'un test :

```
public class <nom de la classe à tester*>Test
```

Annotation des méthodes :

`@Test : méthodes de test`

`+ @Test (expected = Exception.class), @Test(timeout = X).`

`@Before / @After : exéc. avant/après chaque méthode de test.`

`@BeforeClass / @AfterClass : static exéc. avant/après ensemble.`

`@Ignore : méthode ignorée pour les tests + @Ignore("la raison").`

Méthodes de test :

`@Test`

```
public void <nom de la méthode de test>() {
```

Appel de la méthode à tester

**Vérif. comportement via `org.junit.Assert.assert*` ()
(assertion non vérifiée → défaillance).**

}

*. par convention

JUnit4 : TestCase

La classe org.junit.Assert :

- `assertEquals` :
tester l'égalité de deux objets ou de deux types primitifs
(*boolean, byte, char, double, float, int, long, short*)
(delta possible sur les *double* et les *float*).
- `assertFalse, assertTrue` :
tester une condition booléenne.
- `assertNotNull, assertNull` :
tester si une référence est nulle ou non.
- `assertNotSame, assertSame` :
tester si deux objets se réfèrent ou non au même objet.
- `fail` :
faire échouer un test.

→ Message d'erreur personnalisable en premier argument de chacune des méthodes.

JUnit4 : exemple sur Monix

```
package monix.modele.vente;

import org.junit.*;
import static org.junit.Assert.*;
import monix.modele.stock.*;

public class AchatTest
{
    private Produit produit;

    @Before
    public void setUp() throws Exception
    {
        this.produit = new Produit("A", "Libelle_A", new Double(10.50), 1);
    }

    @Test
    public void testIncrementeQuantite()
    {
        Integer origine = 1, ajout = 2, quantiteAttendue = 3;
        Achat achat = new Achat(this.produit, origine);

        achat.incrementeQuantite(ajout);
        assertEquals((int) quantiteAttendue, (int) achat.donneQuantite());
    }
}
```

JUnit4 : TestCase paramétré

Intérêt : Jouer différentes DT avec les méthodes de test[†].

Création d'un test paramétré :

classe de test : @RunWith(Parameterized.class)

déclarer des attributs privés de la classe pour instancier les DT.

déclarer un constructeur avec paramètres pour instancier les DT.

Création du JT :

```
@Parameters
public static Collection<Object[]> <nom méthode>() {
    final Object[][] data = new Object[][] {
        {dt1.1, dt1.2, dt1.3, etc.}
        {dt2.1, dt2.2, dt2.3, etc.}
        etc.
    };
    return Arrays.asList(data);
}
```

[†]. Attention : le JT est joué pour chaque méthode de test du cas de test.

JUnit4 : TestSuite

Principe :

Permettre de lancer plusieurs cas de test et/ou suites de tests, dans l'ordre indiqué des classes.

Classe de suite de tests :

```
@RunWith(Suite.class)
@SuiteClasses({
    <test case a>.class,
    <test case b>.class,
    <test suite x>.class,
    <test suite y>.class,
    etc.
})
public class <nom de la suite de test> {}
```

JUnit4 : TestRunner

Lancement d'un test en mode textuel :

```
java -classpath .:<path/to/>junit4.jar  
      org.junit.runner.JUnitCore <classe.de.test>
```

Une classe de test pouvant être un cas de test ou une suite de tests.

Lancement des tests généralement intégré dans les IDE.

JUnit4 : questions

- Comment tester une méthode privée ?
- Comment faire un *assert* sur un attribut privée ?

JUnit4 : et après ?

JUnit4 ne permet pas en particulier :

- de hiérarchiser les tests,
- d'organiser simplement les DT.

Perspectives :

- JUnit 5 (?)
- TestNG
- et autres frameworks complémentaires...

Plan

3 Mise en œuvre d'une activité de test

- Cycle de développement
- Plan de test
- Cahier de test
- **Outilage**
 - Unit avec JUnit4
 - Mock avec EasyMock ou Mockito
 - Test serveur avec JMeter

Doublure : principe

Plusieurs types de doublures (pour simuler le comportement d'un autre objet) :

- *dummy* (fantôme, bouffon) : objet « vide » sans fonctionnalité implantée.
- *stub* (bouchon) : classe qui renvoie une valeur en dur pour une méthode.
- *fake* (substitut, simulateur) : classe partiellement implantée
(qui renvoie, par exemple, toujours les mêmes réponses selon les paramètres fournis).
- *spy* (espion) : classe qui vérifie son utilisation après exécution.
- *mock* (simulacre) : classe qui agit comme un *stub* et un *spy*.

Un objet de type *mock* permet :

- 1 de simuler le comportement d'un autre objet ;
- 2 de vérifier les invocations qui sont faites de cet objet
(invocations des méthodes, paramètres fournis, ordre des invocations).

Mock : principe

Deux types de *mock*

- statique : classes Java écrites par le développeur ;
- dynamique : mis en œuvre via un *framework*.

→ avantage d'un *mock* dynamique : aucune classe implicite n'est écrite.

Solutions de *frameworks de mocking* :

- **proxy** : fourni à l'objet qui l'utilise via un constructeur ou un *setter* (nécessite un mécanisme d'injection de dépendance dans la classe à tester) (EasyMock, Mockito, etc.).
- **instrumentation** : *classloader* spécifique qui charge dynamiquement une classe à la place d'une autre (utilisation de la classe `java.lang.Instrument` de Java 1.5) (JMockit)
- **orienté aspect** : invocation d'une méthode d'un *mock* à la place de celle d'une implémentation (sans utilisation d'interface ni de mécanisme d'injection de dépendances) (JEasyTest, AMock, etc.).

Mock : principe

Les *frameworks de mocking* permettent généralement de :

- créer dynamiquement des objets *mocks* (à partir de classes ou interfaces) ;
- spécifier les méthodes invoquées (avec paramètres et valeur de retour) ;
- spécifier l'ordre des invocations de ces méthodes ;
- spécifier le nombre d'invocations de ces méthodes ;
- simuler des cas d'erreur en levant des exceptions ;
- valider des appels de méthodes ;
- valider l'ordre de ces appels.

Exemples de *frameworks de mocking* :

- EasyMock, Mockito, PowerMock, JMockIt, JMock, MockRunner, etc.

Mock : principe

Intérêts des *mocks* :

- Invoquer un composant qui n'existe pas encore ;
- Renvoyer des résultats déterminés (pour des tests unitaires automatisés) ;
- Obtenir un état difficilement reproductible (erreur d'accès réseau) ;
- Éviter d'invoquer des ressources longues à répondre (base de données) ;
- Etc.

Mise en œuvre d'un test avec un *mock* :

- 1 Définir le comportement du mock (méthodes, paramètres, exceptions, etc.).
- 2 Exécuter le test en invoquant la méthode à tester.
- 3 Vérifier les résultats du test.
- 4 Vérifier les invocations du ou des *mocks* (nombre et ordre des invocations).

EasyMock : principe

Bibliothèque de *mocking* pour le langage Java.

Création de *mocks* :

- à partir d'interfaces et de classes (versions ≥ 3.0).
- classe **EasyMock** : méthodes statiques.

Utilisation :

- Ajout du *jar* de EasyMock (*easymock.jar*) dans le *classpath*.
- Runner disponible : `@RunWith(EasyMockRunner.class)`

Remarques :

- Dépendance avec *cglib* et *objenesis* (jars dans le *classpath*).

EasyMock : principe

Création d'un *mock* : (alternative avec `@Mock`)

```
<interface|classe> mock =
    EasyMock.createMock(<interface|classe>.class)
```

Comportement d'un *mock* : (liste non exhaustive)

```
mock.<méthode>(<paramètres>)
EasyMock.expect(mock.<méthode>(<paramètres>))
    .andReturn(<retour associé>) (1 invocation)
    .andThrow(<exception associée>) (1 invocation)
    .andStubReturn() OU .andStubThrow () (0-n ? invocations)
```

Initialisation d'un *mock* :

```
EasyMock.replay(mock)
```

Vérification de l'invocation des méthodes :

```
EasyMock.verify(mock)
avec : EasyMock.createMock () (vérifie l'invocation des méthodes).
avec : EasyMock.createStrictMock () (+ ordre des invocations).
```

EasyMock : exemple sur Monix

(cf. Annexe Monix, cf. Code Monix)

```
package monix.modele.vente;

import monix.modele.stock.*;

import static org.junit.Assert.*;
import org.junit.*;
import org.easymock.EasyMock;

/**
 * Classe de test unitaire JUnit 4 de la classe Vente.
 *
 * <p>Utilisation d'un mock (simulacre) de stock (construit avec EasyMock).</p>
 *
 * @see monix.modele.vente.Vente
 */
public class VenteTest
{
    private Stock stockMock;

    @Before
    public void setUp() throws Exception {
        this.stockMock = EasyMock.createMock(Stock.class);
    }

    ...
}
```

EasyMock : exemple sur Monix

(cf. Annexe Monix, cf. Code Monix)

```
@Test
public void testAjouteAchatProduit_quantite() {
    Integer quantite = 2;
    Vente vente = new Vente(this.stockMock);

    EasyMock.expect(
        this.stockMock.donneProduit("A")
    ).andReturn(
        new Produit("A", "l.A", new Double(1), 2)
    );

    EasyMock.replay(this.stockMock);

    try {
        vente.ajouteAchatProduit("A", quantite);
        assertEquals(
            (int) (quantite),
            (int) (vente.donneAchats().get("A").donneQuantite())
        );
    } catch (AchatImpossibleException e) {
        fail("Vente.AchatProduit_catch_AchatImpossibleException");
    }
    EasyMock.verify(this.stockMock);
}
```

...

EasyMock : exemple sur Monix

(cf. Annexe Monix, cf. Code Monix)

```
@Test
public void testAjouteAchatProduit_increment() {
    Integer quantite = 2, increment = 3;
    Vente vente = new Vente(this.stockMock);

    EasyMock.expect(
        this.stockMock.donneProduit("A")
    ).andReturn(
        new Produit("A", "l.A", new Double(1), 2)
    );
    EasyMock.expectLastCall().times(2);
    EasyMock.replay(this.stockMock);

    try {
        vente.ajouteAchatProduit("A", quantite);
        vente.ajouteAchatProduit("A", increment);
        assertEquals(
            (int) (quantite + increment),
            (int) (vente.donneAchats().get("A").donneQuantite())
        );
    } catch (AchatImpossibleException e) {
        fail("Vente.AchatProduit_catch_AchatImpossibleException");
    }
    EasyMock.verify(this.stockMock);
}
```

Mockito : principe

Bibliothèque de *mocking* pour le langage Java.

Création de *mocks* :

- à partir d'interfaces ou de classes.
- création de *spy* possible sur des instances de classe.
- classe **Mockito** : méthodes statiques.

Utilisation :

- Ajout du *jar* de Mockito (mockito-core.jar) dans le *classpath*.
- Runner disponible :
`@RunWith(MockitoJUnitRunner.class)`

Remarques :

- Dépendance avec *byte-buddy* et *objenesis* (jars dans le *classpath*).

Mockito : principe

Création d'un *mock* : (alternative avec `@Mock`)

```
<interface|classe> mock =
    Mockito.mock(<interface|classe>.class)
```

Comportement d'un *mock* : (liste non exhaustive)

```
Mockito.doNothing().when(mock).<méthode>(<param.>)
Mockito.when(mock.<méthode>(<paramètres>))
    .thenReturn(<retour associé>) (0-n invocations)
    .thenThrow(<exception associée>) (0-n invocations)
    .thenCallRealMethod() (0-n invocations sur méthode d'origine)
```

Espionner une instance : `@Spy`

Vérification de l'invocation des méthodes : (liste non exhaustive)

```
Mockito.verify(mock).<méthode>(<param.>)
Mockito.verify(mock, Mockito.times(x)).<m.>(<p.>)
Mockito.verifyNoMoreInteractions(mock)
```

Mockito : exemple sur Monix

(cf. Annexe Monix, cf. Code Monix)

```
package monix.modele.vente;

import monix.modele.stock.*;

import static org.junit.Assert.*;
import org.junit.*;
import org.mockito.Mockito;

/**
 * Classe de test unitaire JUnit 4 de la classe Vente.
 *
 * <p>Utilisation d'un mock (simulacre) de stock (construit avec Mockito).</p>
 *
 * @see monix.modele.vente.Vente
 */
public class VenteTest
{
    private Stock stockMock;

    @Before
    public void setUp() throws Exception {
        this.stockMock = Mockito.mock(Stock.class);
    }

    ...
}
```

Mockito : exemple sur Monix

(cf. Annexe Monix, cf. Code Monix)

```
@Test
public void testAjouteAchatProduit_quantite() {
    Integer quantite = 2;
    Vente vente = new Vente(this.stockMock);

    Mockito.when(
        this.stockMock.donneProduit("A")
    ).thenReturn(
        new Produit("A", "l.A", new Double(1), 2)
    );

    try {
        vente.ajouteAchatProduit("A", quantite);
        assertEquals(
            (int) (quantite),
            (int) (vente.donneAchats().get("A").donneQuantite())
        );
    } catch (AchatImpossibleException e) {
        fail("Vente.AchatProduit_catch_AchatImpossibleException");
    }

    Mockito.verify(this.stockMock).donneProduit(id);
    Mockito.verifyNoMoreInteractions(this.stockMock);
}
```

Mockito : exemple sur Monix

(cf. Annexe Monix, cf. Code Monix)

```
@Test
public void testAjouteAchatProduit_increment() {
    Integer quantite = 2, increment = 3;
    Vente vente = new Vente(this.stockMock);

    Mockito.when(
        this.stockMock.donneProduit("A")
    ).thenReturn(
        new Produit("A", "l.A", new Double(1), 2)
    );

    try {
        vente.ajouteAchatProduit("A", quantite);
        vente.ajouteAchatProduit("A", increment);
        assertEquals(
            (int) (quantite + increment),
            (int) (vente.donneAchats().get("A").donneQuantite())
        );
    } catch (AchatImpossibleException e) {
        fail("Vente.AchatProduit_catch_AchatImpossibleException");
    }

    Mockito.verify(this.stockMock, Mockito.times(2)).donneProduit(id);
    Mockito.verifyNoMoreInteractions(this.stockMock);
}
```

EasyMock, Mockito : et après ?

EasyMock et Mockito ne permettent pas de substituer en particulier :

- les classes finales,
- les méthodes finales,
- les méthodes static,
- les méthodes privées,
- les enums,
- les singltons.

Perspectives :

- PowerMock
- et autres frameworks complémentaires...

Plan

3 Mise en œuvre d'une activité de test

- Cycle de développement
- Plan de test
- Cahier de test
- **Outilage**
 - Unit avec JUnit4
 - Mock avec EasyMock ou Mockito
 - **Test serveur avec JMeter**

JMeter : présentation

Apache JMeter (*The Apache Software Foundation*).

A l'origine :

- test fonctionnel nominal et
- test de robustesse (performance : montée en charge, stress)

des applications serveur.

Aujourd'hui :

- étendu à tout type d'application et
- utilisable, par méthodologie, pour d'autres types de test.

JMeter : principes

Mise en œuvre des tests :

- de nombreux protocoles supportés (TCP/IP, HTTP(S), IMAP, LDAP, etc.) ;
- simulation de plusieurs utilisateurs ;
- lancement de tests à partir de plusieurs postes.

Scénarios de tests :

- scénarios avec forge de requêtes et structures de contrôle ;
- utilisation de JT externalisés ;
- enregistrement (et modification) de scénarios de requêtes (web).

Résultats des tests :

- assertions sur les réponses serveur ;
- récolte de métriques sur les échanges client/serveur ;
- plusieurs moyens de visualisation des assertions et des métriques.

Extensibilité :

- de nombreux greffons (add-on, plugins) sont disponibles (gestion de protocoles, visualisation de résultats, etc.).

JMeter : organisation

Deux emplacements :

- **Plan de test** : pour positionner les tests à exécuter.
- **Plan de travail** : pour stocker des composants de test.

Remarques :

Enregistrement et ouverture des plans (test et travail) séparément :

- menu Fichier → Ouvrir... (idem Enregistrer sous...)
- clique droit → Ouvrir... (idem Enregistrer sous...)
- format : jmx

Ajout de composants :

- menu Éditer → Ajouter
- clique droit → Ajouter

JMeter : composants

Groupe d'unités (*thread group*)

(Plan de test : Moteurs d'utilisateurs → Groupe d'unités)

- Paramètres de l'exécution des tests :

- nombre d'utilisateurs (qui se connectent) ;
- montée en charge (intervalle entre les connexions) ;
- nombre d'itérations (pour chaque utilisateur) ;
- programmateur de démarrage (date et heure des tests).
- comportement après une erreur (arrêt du test, de l'unité, etc.).

- Ajout d'échantillons et de contrôleurs logiques.

JMeter : composants

Échantillons (*samplers*)

- Actions des unités du groupe :
 - forge et envoi de requêtes (HTTP, FTP, TCP, etc.) ;
 - action test (suspension ou arrêt du test) ;
 - appel de processus systèmes ;
 - récolte d'informations de débogage ;
 - utilisation de scripts (BeanShell, BSF, JSR223) ;
 - accès à des journaux de serveurs web ;
 - accès à des mails (POP3, IMAP) ;
 - utilisation de classes Java ;
 - lancement de tests unitaires (JUnit).

JMeter : composants

Contrôleurs logiques (*logic controllers*)

- Contrôle de l'exécution des requêtes :
 - structures de contrôle (boucles, conditions) ;
 - aléatoire (envoi de requêtes dans un ordre aléatoire) ;
 - exécution unique (requête unique - indép. nb. itérations) ;
 - transaction (regrouper des requêtes) ;
 - durée d'exécution (des requêtes) ;
 - débit (nombre d'exécutions - total ou pourcentage).
- Organisation du plan :
 - simple (regroupement logique de requêtes) ;
 - module (appel d'un *fragment d'éléments*) ;
 - inclusion (utilisation d'un jmx externe).

JMeter : composants

Configuration (*config elements*)

- Paramètres généraux des requêtes et variables :
 - connexions (HTTP (autorisation, cookies...), FTP, TCP, etc.) ;
 - lecture de données dans un fichier (CSV) ;
 - variables (aléatoires, pré-définies) ;
 - etc. (identification, compteurs, requêtes java...).

Compteurs de temps (*timers*)

- Écoulement de temps avant une requête :
 - déterministe (fixe, débit constant) ;
 - aléatoire (gaussien, uniforme) ;
 - utilisation de scripts (BeanShell, BSF, JSR223) ;
 - attente de synchronisation (entre différentes unités de test).

JMeter : composants

Pré-processeurs (*pre processors*)

- Modification des requêtes avant exécution :
 - données HTML (liens, formulaires, paramètres) ;
 - données HTTP (URL, paramètres) ;
 - paramètres utilisateur (individuels) ;
 - utilisation de scripts (BeanShell, BSF, JSR223).

Post-processeurs (*post processors*)

- Manipulation des réponses serveur :
 - extraction de données (expression régulière, XPath) ;
 - arrêt d'une unité de test (si un test ne passe pas) ;
 - accès à une base de données (JDBC) ;
 - utilisation de scripts (BeanShell, BSF, JSR223).

JMeter : composants

Assertions

- Vérification sur les réponses aux requêtes :
 - chaînes de caractères (comparaison, données, patrons) ;
 - taille, temps de réponse (limite) ;
 - HTML (syntaxe), XML (schéma, XPath) ;
 - Hash (MD5) ;
 - utilisation de scripts (BeanShell, BSF, JSR223).

Récepteurs (*listeners*)

- Exploitation des résultats des tests :
 - visualisation (arbres, tableaux, graphiques) ;
 - envoi de mail, enregistrement (données, réponses) ;
 - utilisation de scripts (BeanShell, BSF, JSR223).

JMeter : composants

Éléments hors test (*non-test elements*) (Plan de travail)

- Affichage des propriétés (système et JMeter).
- Serveur HTTP Proxy (capturer une session HTTP) :

→ Mise en place du serveur proxy :

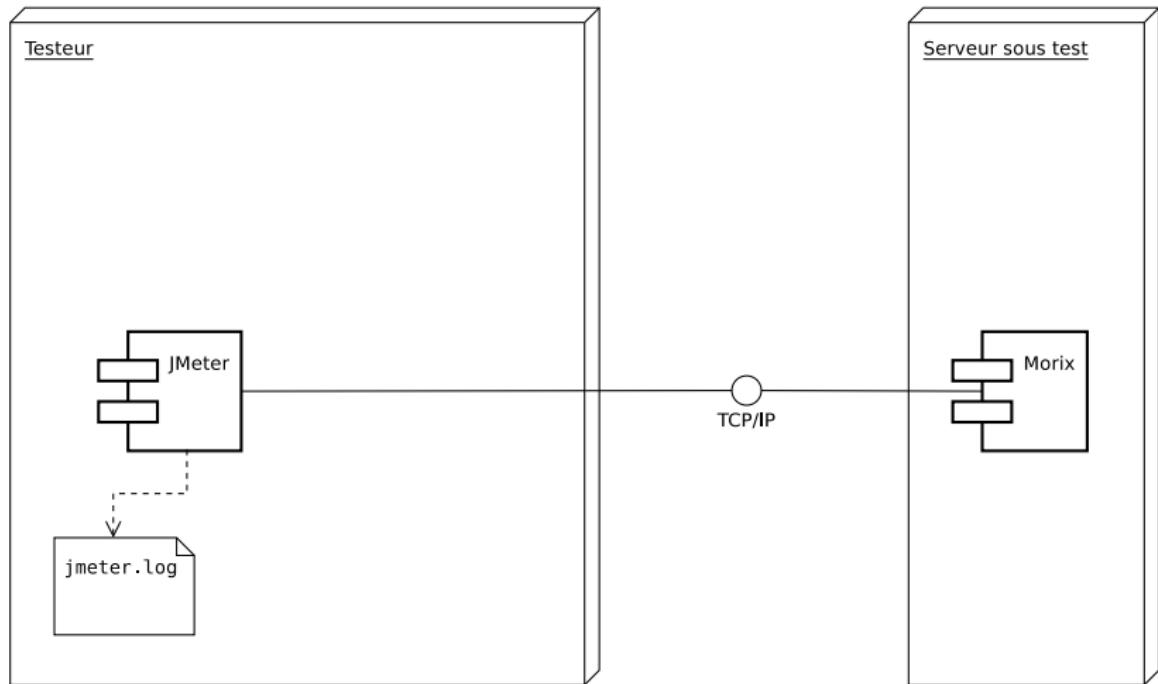
Plan de test : Ajouter → Groupe d'unités;
Groupe d'unité : Ajouter → Contrôleur enregistreur;
Plan de travail : Ajouter → Serveur HTTP Proxy;
Serveur HTTP Proxy : Lancer.

→ Utilisation du serveur proxy :

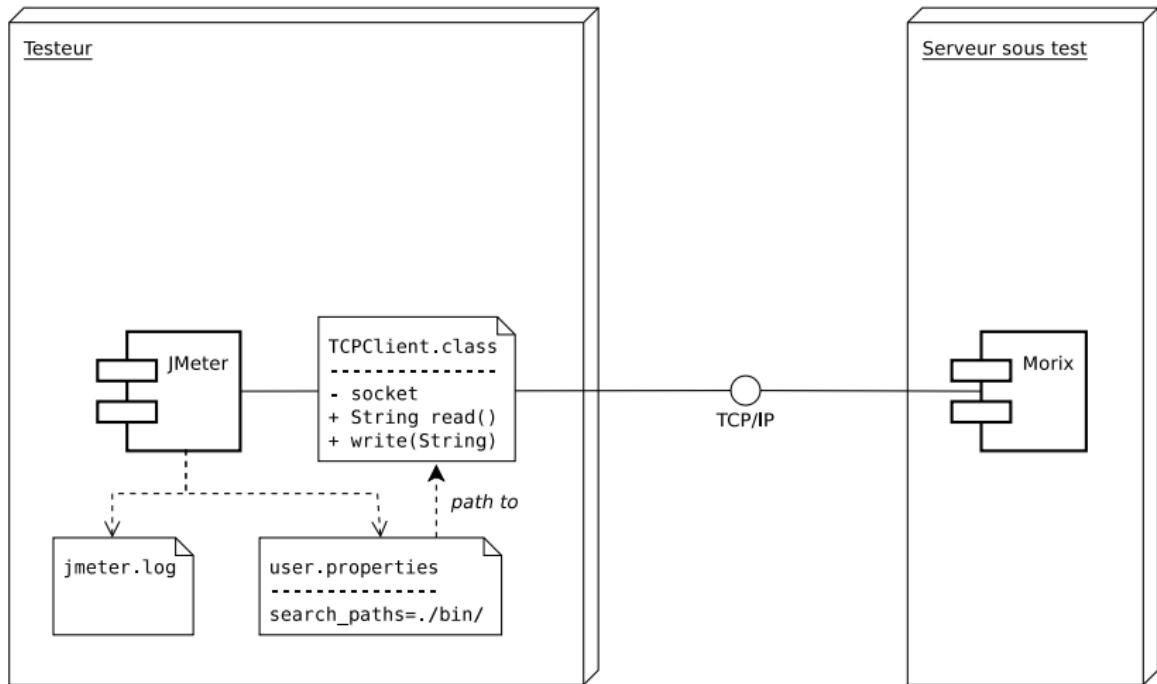
Navigateur Web : configuration du proxy (host, port);
JMeter : requêtes capturées dans le Contrôleur enregistreur.

- Serveur HTTP miroir (renvoi des données envoyées).

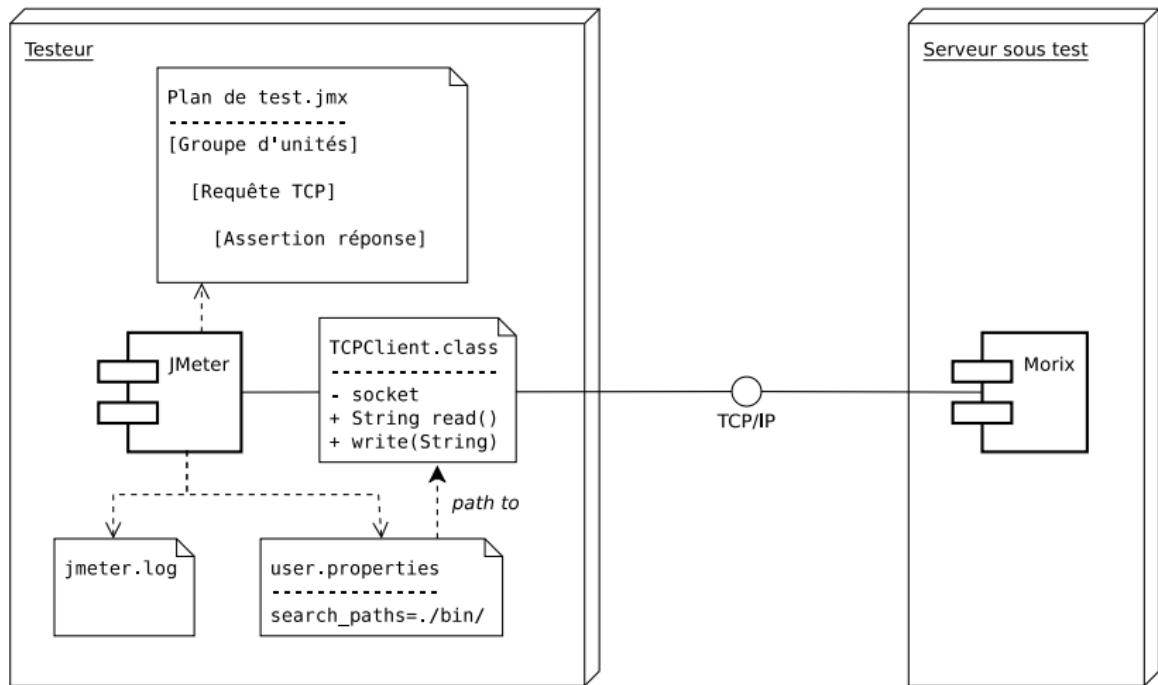
JMeter : exemple TCP/IP (sur Morix, cf. Annexe Monix)



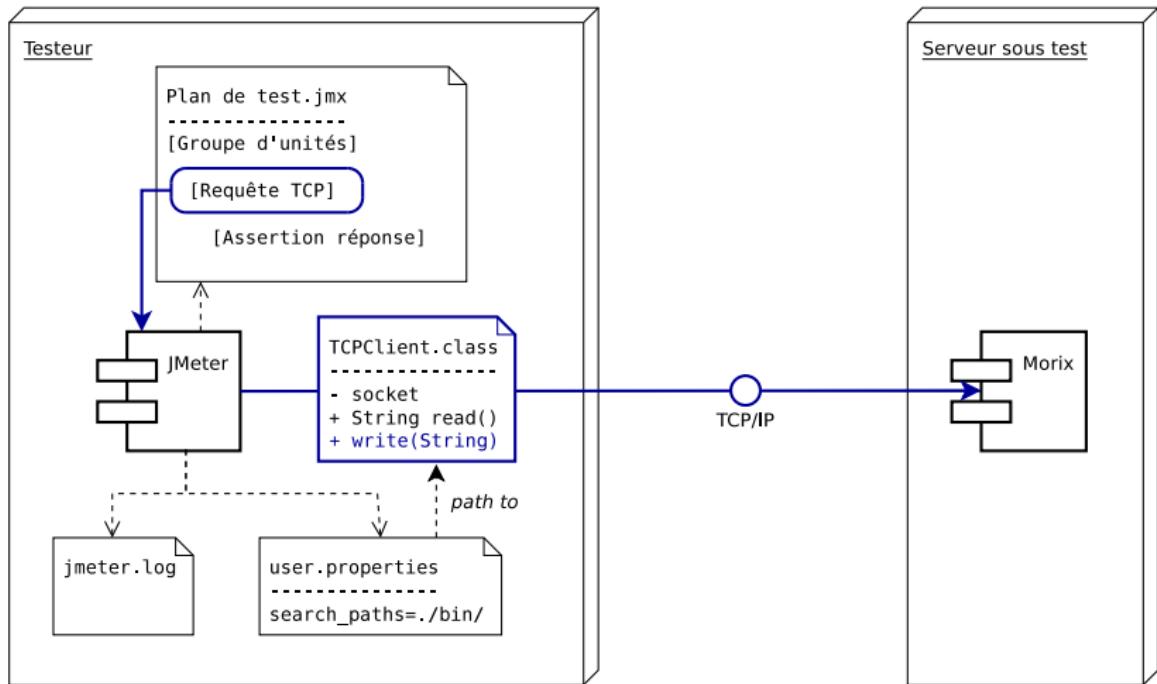
JMeter : exemple TCP/IP (sur Morix, cf. Annexe Monix)



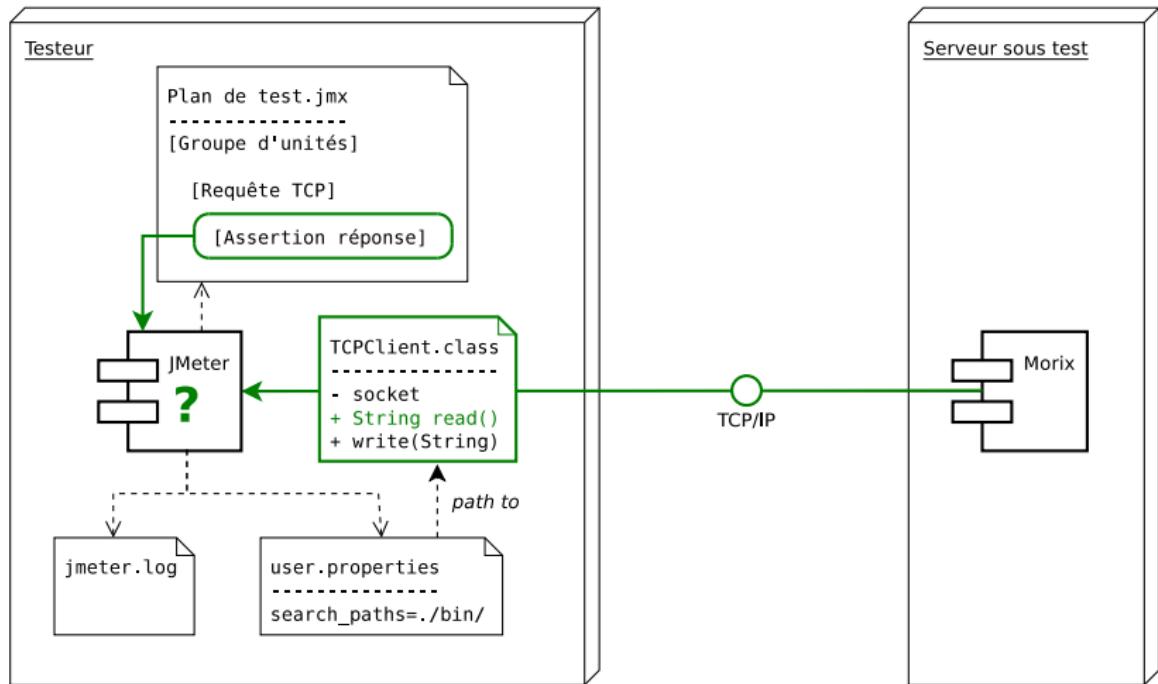
JMeter : exemple TCP/IP (sur Morix, cf. Annexe Monix)



JMeter : exemple TCP/IP (sur Morix, cf. Annexe Monix)



JMeter : exemple TCP/IP (sur Morix, cf. Annexe Monix)



JMeter : exemple scénario (sur Morix, cf. Annexe Monix)

Client 1
(JMeter)

Morix



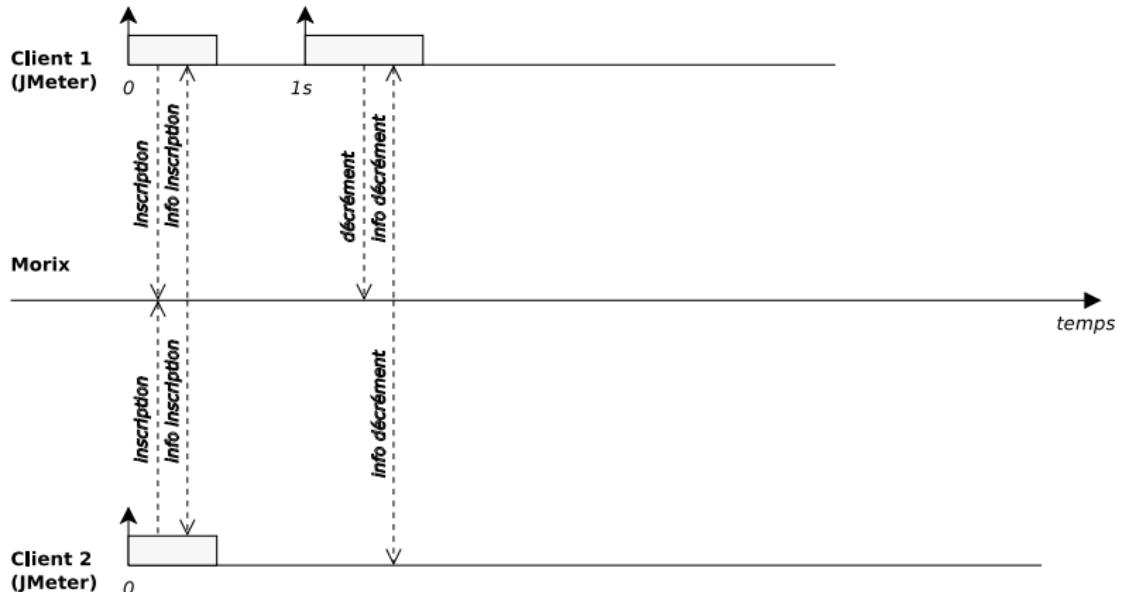
A horizontal timeline represented by a thick black arrow pointing to the right, labeled "temps" at its tip. Three vertical lines extend upwards from the timeline to represent different clients: "Client 1 (JMeter)", "Morix", and "Client 2 (JMeter)".

Client 2
(JMeter)

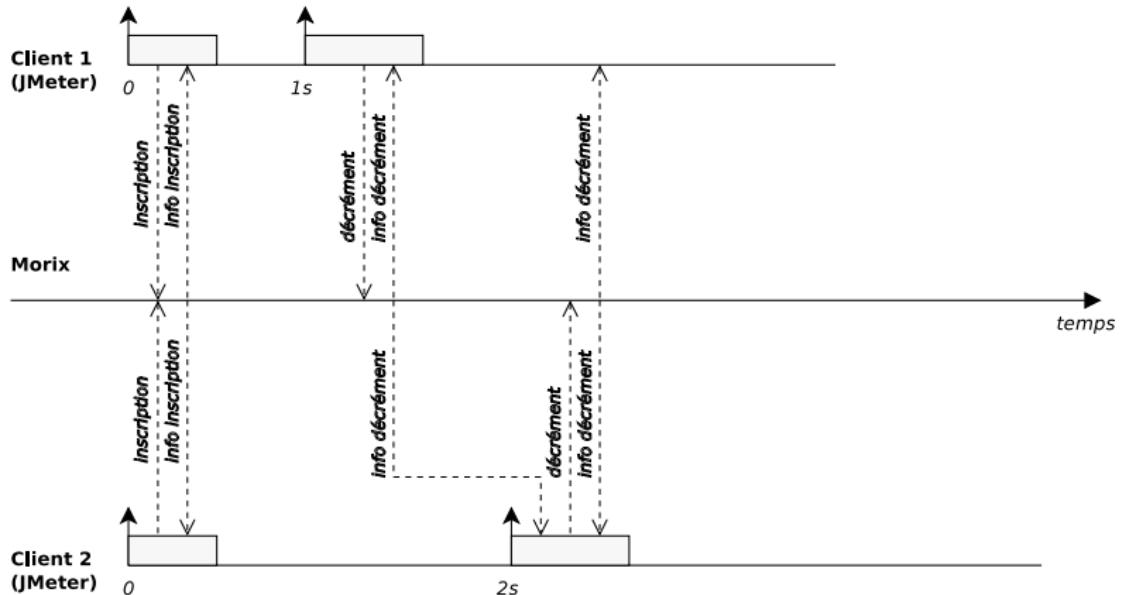
JMeter : exemple scénario (sur Morix, cf. Annexe Monix)



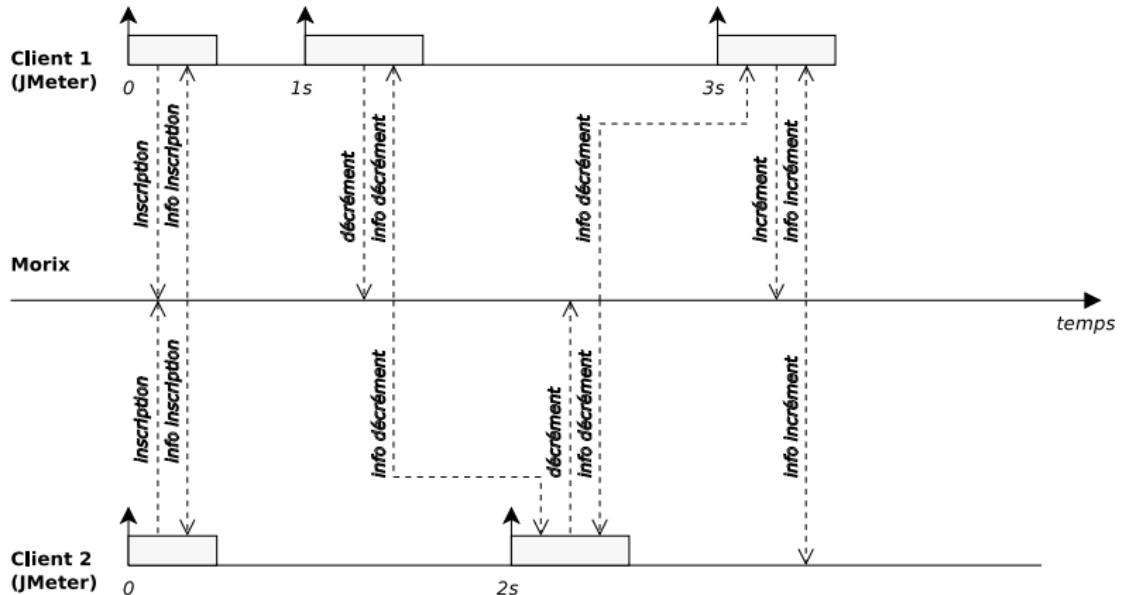
JMeter : exemple scénario (sur Morix, cf. Annexe Monix)



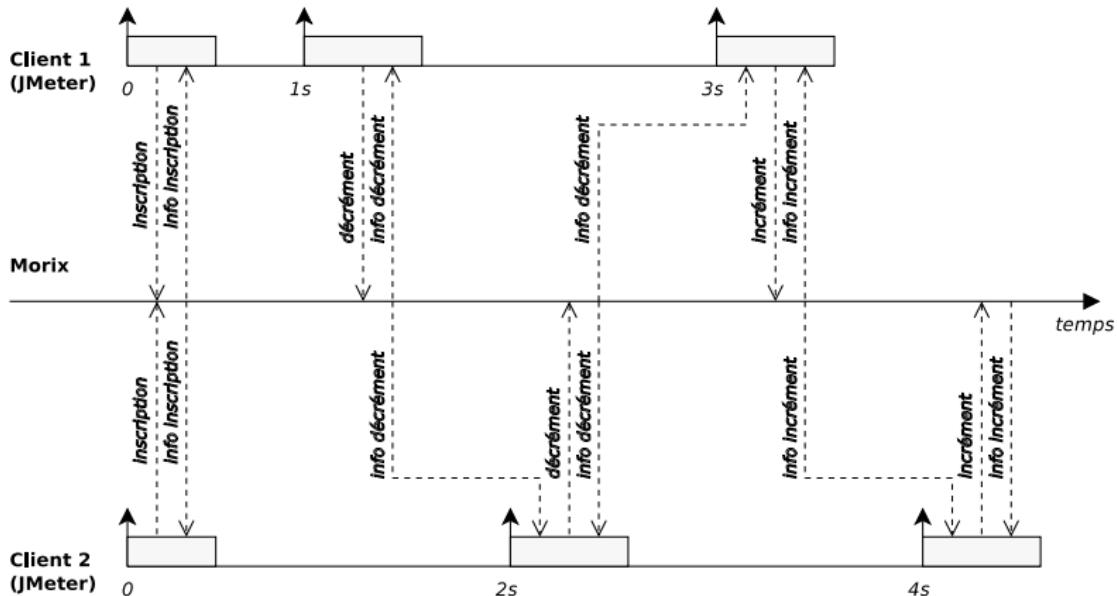
JMeter : exemple scénario (sur Morix, cf. Annexe Monix)



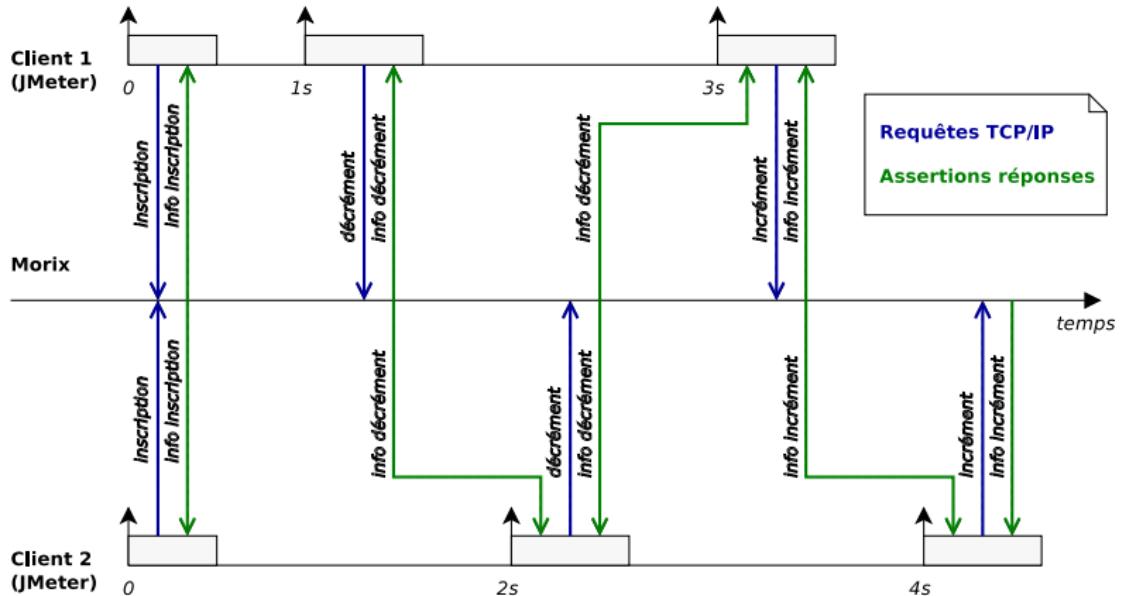
JMeter : exemple scénario (sur Morix, cf. Annexe Monix)



JMeter : exemple scénario (sur Morix, cf. Annexe Monix)



JMeter : exemple scénario (sur Morix, cf. Annexe Monix)



JMeter : exemple plan de test (sur Morix, cf. Annexe Monix)

① Création des variables pour le protocole

(Ajouter → Configurations → Variables pré-définies)

Nom : INSCRIPTION_CLIENT, Valeur : /I

Nom : MESSAGE_INSCRIPTION, Valeur : Inscription OK

② Création d'une unité de test

(Ajouter → Moteur d'utilisateurs → Groupe d'unités)

③ Création d'un contrôleur simple (dans l'unité de test)

(Ajouter → Contrôleurs → Contrôleur simple)

④ Renseignement des paramètres TCP (du contrôleur)

(Ajouter → Configurations → Paramètres TCP par défaut)

Nom ou IP du serveur : 127.0.0.1

Numéro de port : 13579

Expiration (millisecondes) : 500

⑤ Ajouter un compteur de temps fixe (au contrôleur)

(Ajouter → Compteurs de temps → Compteur de temps fixe)

Délai d'attente (en millisecondes) : 500

JMeter : exemple plan de test (sur Morix, cf. Annexe Monix)

(6) Ajouter une requête TCP (au contrôleur)

(Ajouter → Échantillons → Requête TCP)

Ré-utiliser la connexion : ✓

Texte à envoyer : \${INSCRIPTION_CLIENT} (suivi d'un retour à la ligne)

(7) Ajouter une assertion (à la requête)

(Ajouter → Assertions → Assertion réponse)

Appliquer sur : ✓ L'échantillon

Section de réponse à tester : ✓ Texte de réponse

Motif à tester : \${MESSAGE_INSCRIPTION}

(8) Ajouter un récepteur (au contrôleur)

(Ajouter → Récepteurs → Arbre de résultats)

(9) Sauver le plan de test

(Fichier → Enregistrer le plan de test)

(10) Lancer le plan de test (morix étant lancé)

(Lancer → Lancer)

(à suivre) Développement d'une classe TCPClientMorix (implements TCPClient)...

Plan

4

Techniques de vérification & validation

- Élaboration de données de test boîte noire
- Élaboration de données de test boîte blanche
- Évaluation des données de test
- Analyse structurelle statique

Accessibilité et techniques de V&V

Boîte noire :

- Test fonctionnel. :

- analyse partitionnelle,
- analyse aux limites,
- test combinatoire,
- table de décision,
- test aléatoire et statistique,
- couverture de graphe,
- test syntaxique,
- algorithmique qualitative,
- analyse transactionnelle,
- graphe cause-effet, etc.

Boîte blanche :

- Test structurel dynamique ; :

- graphe de contrôle,
- exécution abstraite,
- test évolutionniste,
- domaines finis, etc.

- Analyse structurelle statique. :

- revue de code,
- mesures de la complexité,
- analyse du flot de données,
- exécution symbolique,
- preuve formelle,
- interprétation abstraite,
- model-checking, etc.

Plan

4

Techniques de vérification & validation

- Élaboration de données de test boîte noire
- Élaboration de données de test boîte blanche
- Évaluation des données de test
- Analyse structurelle statique

Élaboration de données de test boîte noire

L'élaboration de données de test boîte noire se fait à l'aide des méthodes de test fonctionnel.

Le test fonctionnel (rappel)

Examiner le comportement fonctionnel du logiciel et sa conformité avec la spécification du logiciel.

Méthodes de test fonctionnel abordées dans ce cours :

- analyse partitionnelle ;
- analyse aux limites ;
- test combinatoire ;
- table de décision ;
- test aléatoire et test statistique ;
- couverture du graphe fonctionnel ;
- génération automatique de test.

Plan

4

Techniques de vérification & validation

- Élaboration de données de test boîte noire
 - Analyse partitionnelle
 - Analyse aux limites
 - Test combinatoire
 - Table de décision
 - Test aléatoire et test statistique
 - Couverture du graphe fonctionnel
 - Génération automatique de tests fonctionnels
- Élaboration de données de test boîte blanche
- Évaluation des données de test
- Analyse structurelle statique

Analyse partitionnelle

Classe d'équivalence

Une *classe d'équivalence* correspond à un ensemble de DT supposées tester le même comportement (*i.e.* activer le même défaut). Soit C_i une classe d'équivalence, $\cup C_i = E \wedge \forall i, j, C_i \cap C_j = \emptyset$.

Objectif

Fournir un minimum de DT pour couvrir un maximum de cas.

Analyse partitionnelle

La construction des classes d'équivalence est conditionnée par :

- la spécification des *variables d'entrée* et/ou
- la spécification des *résultats*.

Méthode de l'analyse partitionnelle

Quatre phases :

- modélisation du problème adressé par le programme (modélisation par préoccupations) ;
- pour chaque préoccupation (donnée d'entrée ou de sortie), calcul de classes d'équivalence sur les domaines de valeurs ;
- choix d'un représentant de chaque classe d'équivalence ;
- composition par produit cartésien sur l'ensemble des données d'entrée (permettant de couvrir les classes d'équivalence) pour établir les DT.

Analyse partitionnelle

Exemple

Soit P un programme calculant $f : \mathbb{R} \rightarrow \mathbb{R}$, $f(x) = \sqrt[2]{x}$.

Il y a 3 classes d'équivalence : \mathbb{R}^+ , \mathbb{R}^- et $\{0\}$.

Un jeu de test peut alors être : $JT_1 = \{-2.5, 0, 3.4\}$.

Exercice 4.1

Soit P un programme calculant la ou les racines carrées de $z \in \mathbb{C}$.

Établir les classes d'équivalence et proposer un JT.

Analyse partitionnelle

Exercice 4.2 ↗

Soit P' un programme qui, en fonction de la longueur de trois segments, détecte s'il s'agit d'un triangle, s'il est isocèle, scalène, équilatéral et si son plus grand angle est aigu, droit ou obtus. Établir les classes d'équivalence et proposer un JT.

Analyse partitionnelle

Exercice 4.3

Proposez et justifiez des données de test couvrant le comportement nominal d'une fonction réalisant le tri d'un tableau quelconque d'entiers.

Exercice 4.4

Proposez et justifiez des données de test couvrant le comportement nominal d'une fonction réalisant l'insertion d'un élément dans une liste chaînée. Les éléments de la liste se composent d'un entier et d'une chaîne de caractères. La liste est organisée par ordre croissant des entiers des éléments qu'elle contient.

Analyse partitionnelle

Points forts :

- Approche intuitive.
- Diminue le nombre de cas de test par calcul de classes d'équivalence.

Points faibles :

- Risque d'explosion du nombre de cas de test (produit cartésien, combinaison des valeurs d'entrée).
- Ne permet de tester que le comportement nominal.

Plan

4

Techniques de vérification & validation

- Élaboration de données de test boîte noire
 - Analyse partitionnelle
 - Analyse aux limites
 - Test combinatoire
 - Table de décision
 - Test aléatoire et test statistique
 - Couverture du graphe fonctionnel
 - Génération automatique de tests fonctionnels
- Élaboration de données de test boîte blanche
- Évaluation des données de test
- Analyse structurelle statique

Analyse aux limites

Analyse aux limites

Déterminer des données de test aux limites des champs de définition des différentes variables.

- On s'intéresse aux bornes des intervalles partitionnant les domaines des variables d'entrées.
- Choix des valeurs juste avant, sur et juste après les bornes de définition.
- Permet de définir d'autres types de partitions (souvent des cas particuliers - valeur extrême - des classes d'équivalence déterminées par l'analyse partitionnelle).

Analyse aux limites

Identifier :

- le domaine de définition des paramètres d'entrée,
- le type (au sens large) du paramètre
(*numeric, character...*)
(*speed, location, position, quantity, size...*),
- les caractéristiques de ces types
(*first/last, min/max, start/finish, over/under, empty/full, shortest/longest, slowest/fastest, soonest/latest, largest/smallest, highest/lowest, next-to/farthest-from...*).

Analyse aux limites

Pour un intervalle

On garde les 2 valeurs correspondant aux 2 limites, et les 4 valeurs correspondant aux valeurs des limites \pm le plus petit delta possible.

Exemple : $n \in [3..15] \Rightarrow v_1 = 3, v_2 = 15, v_3 = 2, v_4 = 4, v_5 = 14, v_6 = 16$

Pour un ensemble ordonné de valeurs

On choisit la première, la seconde, l'avant dernière et la dernière.

Exemple : $n \in \{-7, 2, 3, 157, 200\} \Rightarrow v_1 = -7, v_2 = 2, v_3 = 157, v_4 = 200$

Pour un nombre de valeurs

On définit les cas de test à partir du nombre minimum et maximum de valeurs, et des tests pour des nombres de valeurs hors limites.

Exemple : Fichier de 1 à 255 données, test pour 0, 1, 255 et 256 données.

Analyse aux limites

Exercice 4.5 ↗

Appliquer le test aux limites sur le programme des triangles (4.2).

Exercice 4.6 ↗

Appliquer le test aux limites sur le programme de tri de tableaux (4.3).

Exercice 4.7 ↗

Appliquer le test aux limites sur le programme d'insertion d'un élément dans une liste chaînée (4.4).

Analyse aux limites

Exercice 4.8 ↗

Tester un programme analysant un fichier comprenant au maximum 100 fiches d'étudiants. Chaque fiche contient le nom de l'étudiant (20 caractères maximum), son sexé (un caractère) et ses notes dans 5 matières (entiers compris entre 0 et 20).

Le but du programme est de calculer :

- la moyenne de chaque étudiant,
- la moyenne générale (par sexe ou par matière),
- le nombre d'étudiants qui ont une moyenne générale ≥ 10 .

Analyse aux limites

Points forts :

- Méthode de test fonctionnel très productive
(le comportement du programme aux valeurs limites est souvent pas ou insuffisamment examiné).
- Produit à la fois des cas de test nominaux (dans l'intervalle) et de robustesse (hors intervalle).

Points faibles :

- Caractère parfois intuitif ou subjectif de la notion de limite
→ difficulté pour caractériser la couverture de test.

Analyse aux limites et analyse partitionnelle

Les techniques d'analyse partitionnelle et aux limites :

- se complètent « naturellement »,
- couvrent l'ensemble des phases de test (unitaires, intégration, validation).

Conclusion :

Les techniques d'analyse partitionnelle et aux limites sont souvent considérées comme le minimum des tests à appliquer sur un système.

Plan

4

Techniques de vérification & validation

- Élaboration de données de test boîte noire
 - Analyse partitionnelle
 - Analyse aux limites
 - **Test combinatoire**
 - Table de décision
 - Test aléatoire et test statistique
 - Couverture du graphe fonctionnel
 - Génération automatique de tests fonctionnels
- Élaboration de données de test boîte blanche
- Évaluation des données de test
- Analyse structurelle statique

Test combinatoire

Problème :

Les combinaisons de valeurs d'entrée donnent lieu à une explosion combinatoire.

Approche *Pairwise*

Tester un fragment des combinaisons de valeurs qui garantissent que chaque combinaison de 2 variables est testée.

Idée sous-jacente :

La majorité des fautes sont détectées par des combinaisons de deux valeurs de variables.

Test combinatoire

Exemple

4 variables avec 3 valeurs possibles (81 combinaisons) :

OS : Linux, MS.XP, Mac.OSX

Réseau : RJ45, Wifi, Bluetooth

Application : OpenOffice.org, The Gimp, Blender

Imprimante : HP35, Brother2070, Canon900

Approche *Pairwise* : 9 paires.

OS	Réseau	Imprimante	Application
MS.XP	Bluetooth	Brother2070	The Gimp
Mac.OSX	RJ45	HP35	The Gimp
Mac.OSX	Wifi	Brother2070	OpenOffice.org
MS.XP	RJ45	Canon900	OpenOffice.org
MS.XP	Wifi	HP35	Blender
Linux	Bluetooth	HP35	OpenOffice.org
Linux	RJ45	Brother2070	Blender
Mac.OSX	Bluetooth	Canon900	Blender
Linux	Wifi	Canon900	The Gimp

Test combinatoire

Points forts :

- L'approche *Pairwise* se décline avec des triplets, des quadruplets, etc. (mais le nombre de tests augmente très vite).
- Une dizaine d'outils permettent de calculer les combinaisons en *Pairwise* (ou n-valeurs) (<http://www.pairwise.org>).

Points faibles :

- Le résultat attendu de chaque test doit être fourni manuellement.

Plan

4

Techniques de vérification & validation

• Élaboration de données de test boîte noire

- Analyse partitionnelle
- Analyse aux limites
- Test combinatoire
- Table de décision
- Test aléatoire et test statistique
- Couverture du graphe fonctionnel
- Génération automatique de tests fonctionnels

• Élaboration de données de test boîte blanche

- Évaluation des données de test
- Analyse structurelle statique

Table de décision

Principe

Le comportement d'un logiciel peut être décrit par les actions qu'il effectue. Selon certaines conditions sur les variables d'entrée, certaines actions seront observées ou non.

Si l'ordre des entrées n'influe pas sur les actions, une *table de décision* est un moyen simple et concis de représenter le lien entre une condition sur les variables d'entrée et une action.

Format d'une table de décision :

<i>condition 1</i>	<i>V₁</i>	<i>V_{1'}</i>	...
<i>condition 2</i>	<i>V₂</i>	<i>V_{2'}</i>	...
<i>action 1</i>	X		
<i>action 2</i>		X	
...			

→ Chaque colonne (ou « variant ») définit alors une DT.

Table de décision

Exemple

Soient C_1 , C_2 et C_3 trois conditions/variables telles que :

$$C_1 \in \{0, 1\}, C_2 \in \{0, 1\} \text{ et } C_3 \in \{0, 1, 2, 3\}$$

et soient A_1 , A_2 , A_3 et A_4 quatre actions relatives à ces conditions.

	CR	E	D																	
C_1	1	2	2	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	
C_2	2	2	4	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	
C_3	4	4	16	0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3	
A_1		x	x					x	x						x					
A_2			x						x		x	x					x			
A_3										x				x				x		
A_4					x					x					x					

CR (coefficent de répétition) : nombre de répétitions de la valeur au niveau de la table ;

E (étendue de la condition) : nombre de valeurs possibles de la condition ;

D (domaine de la condition) : nombre de colonnes qu'occupent les valeurs de la condition.

Table de décision

Exemple

Soient C_1 , C_2 et C_3 trois conditions/variables telles que :

$$C_1 \in \{0, 1\}, C_2 \in \{0, 1\} \text{ et } C_3 \in \{0, 1, 2, 3\}$$

et soient A_1 , A_2 , A_3 et A_4 quatre actions relatives à ces conditions.

	CR	E	D	<i>don't care</i> → un seul variant															
C_1	1	2	2	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	
C_2	2	2	4	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	
C_3	4	4	16	0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	
A_1				x	x				x	x					x				
A_2					x					x		x	x				x		
A_3											x				x			x	
A_4						x					x					x			

CR (coefficent de répétition) : nombre de répétitions de la valeur au niveau de la table ;

E (étendue de la condition) : nombre de valeurs possibles de la condition ;

D (domaine de la condition) : nombre de colonnes qu'occupent les valeurs de la condition.

Table de décision

Exemple

Soient C_1 , C_2 et C_3 trois conditions/variables telles que :

$$C_1 \in \{0, 1\}, C_2 \in \{0, 1\} \text{ et } C_3 \in \{0, 1, 2, 3\}$$

et soient A_1 , A_2 , A_3 et A_4 quatre actions relatives à ces conditions.

	CR	E	D	alerte → oubli d'une action ou manquement de spécification															
C_1	1	2	2	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	
C_2	2	2	4	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
C_3	4	4	16	0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3
A_1				x	x				x	x					x				
A_2					x					x		x	x				x		
A_3											x				x			x	
A_4						x					x					x			

CR (coefficent de répétition) : nombre de répétitions de la valeur au niveau de la table ;

E (étendue de la condition) : nombre de valeurs possibles de la condition ;

D (domaine de la condition) : nombre de colonnes qu'occupent les valeurs de la condition.

Table de décision

Exemple

Soient C_1 , C_2 et C_3 trois conditions/variables telles que :

$$C_1 \in \{0, 1\}, C_2 \in \{0, 1\} \text{ et } C_3 \in \{0, 1, 2, 3\}$$

et soient A_1 , A_2 , A_3 et A_4 quatre actions relatives à ces conditions.

	CR	E	D	alerte	→ erreur de consistance dans les spécifications														
C_1	1	2	2	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
C_2	2	2	4	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
C_3	4	4	16	0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3
A_1		x	x				x	x							x				
A_2			x					x			x	x	x				x		
A_3									x					x				x	
A_4				x					x						x				

CR (coefficent de répétition) : nombre de répétitions de la valeur au niveau de la table ;

E (étendue de la condition) : nombre de valeurs possibles de la condition ;

D (domaine de la condition) : nombre de colonnes qu'occupent les valeurs de la condition.

Table de décision

Exercice 4.9 ↗

Élaborer une table de décision pour le programme des triangles (4.6).

Se limiter pour cela à un programme qui détecte si le triangle est *équilatéral, isocèle ou scalène* (données en entrée : $(a, b, c) \in \mathbb{R}^{+^3}$).

Table de décision

Points forts :

- Modélisation simple basée sur la logique ;
- Réduction de la table automatisable ;
- Détection éventuelle d'oublis ou d'erreurs dans la spécification.

Points faibles :

- Lisibilité : taille de la table ;
- Conditions et actions déterminées de manière artisanale (par ex. analyse partitionnelle).

Plan

4

Techniques de vérification & validation

• Élaboration de données de test boîte noire

- Analyse partitionnelle
- Analyse aux limites
- Test combinatoire
- Table de décision

• Test aléatoire et test statistique

- Couverture du graphe fonctionnel
- Génération automatique de tests fonctionnels

• Élaboration de données de test boîte blanche

• Évaluation des données de test

• Analyse structurelle statique

Test aléatoire et test statistique

Principe

Le jeu de tests est sélectionné à l'aide d'une fonction de calcul sur le domaine d'entrée, à partir de la spécification.

- **Test aléatoire** : utilisation d'une fonction aléatoire.
- **Test statistique** : utilisation d'une loi de distribution.

Test aléatoire et test statistique

Exemple

Un programme d'inversion de matrice carrée de taille entre 2 et 10.

→ $a \in [2..10]$ aléatoire, donne la taille de la matrice.

Puis, remplissage de la matrice avec des données aléatoires.

Autres exemples

- Échantillonnage de 5 en 5 pour une donnée d'entrée représentant une distance.
- Utilisation d'une loi de Gauss pour une donnée représentant la taille des individus.

Test aléatoire et test statistique

Points forts :

- Objectivité des DT.
- Automatisable pour la sélection des DT
(plus difficile pour le résultat attendu).
- Des résultats satisfaisants
(50 à 75% de couverture rapidement).
- → intéressant en première approximation.

Points faibles :

- Fonctionnement en aveugle.
- Pas de prise en compte de comportements spécifiques.

Plan

4

Techniques de vérification & validation

• Élaboration de données de test boîte noire

- Analyse partitionnelle
- Analyse aux limites
- Test combinatoire
- Table de décision
- Test aléatoire et test statistique
- **Couverture du graphe fonctionnel**
- Génération automatique de tests fonctionnels

• Élaboration de données de test boîte blanche

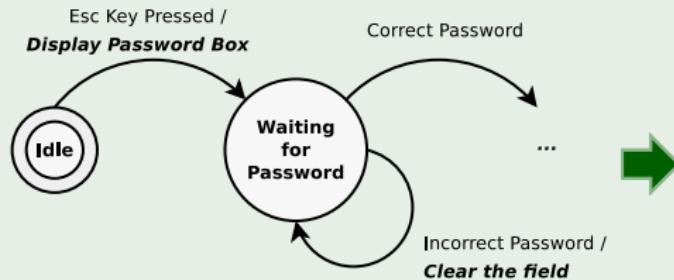
- Évaluation des données de test
- Analyse structurelle statique

Couverture du graphe fonctionnel

Principe

Présence d'une modélisation des spécifications sous forme de machine à états (FSM). L'objectif est alors de couvrir tous les arcs ou tous les sommets du graphe fonctionnel.

Exemple



Couverture de tous les sommets :

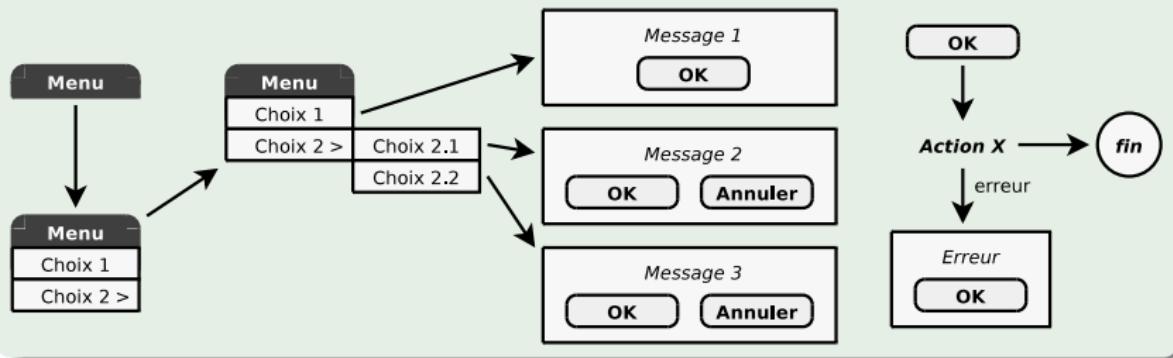
- (1) Idle,
- (2) Waiting for Password.

Couverture de tous les arcs :

- (1) Esc Key Pressed,
- (2) Incorrect Password,
- (3) Correct Password.

Couverture du graphe fonctionnel

Exercice 4.10



Exercice 4.11

Application sur Morix

Élaborer le JT permettant de couvrir la machine à états du protocole de communication avec le composant morix.

Couverture du graphe fonctionnel

Points forts :

- Adaptée à des logiciels interagissant fortement avec l'environnement (communication, IHM, temps réel...).
- Très efficace.
- Par expérience : couvre 40 à 60% des instructions exécutables.

Points faibles :

- Disponibilité d'une description adaptée.
- Complexité grandissante de cette modélisation.

Plan

4

Techniques de vérification & validation

• Élaboration de données de test boîte noire

- Analyse partitionnelle
- Analyse aux limites
- Test combinatoire
- Table de décision
- Test aléatoire et test statistique
- Couverture du graphe fonctionnel
- Génération automatique de tests fonctionnels

• Élaboration de données de test boîte blanche

- Évaluation des données de test
- Analyse structurelle statique

Génération automatique de tests à partir d'un modèle de la spécification : Model-based testing I

- Test boîte noire
- Peut s'effectuer sur l'ensemble des phases de test (unitaires, d'intégration, de validation, de non-régression).
- Se base sur les modèles, celui-ci sera donc adapté au niveau testé (module, système) et aux objectifs de test.

Génération automatique de tests à partir d'un modèle de la spécification : Model-based testing II

- Se fait par rapport à des directives (scénarios de test)
 - critères de couverture du modèle,
 - critères de sélection sur le modèle.
- Se décompose en 3 phases :
 - modélisation de la spécification,
 - génération des cas de test à partir de celle-ci,
 - exécution de ces cas de test sur le système réel.

Model-based testing : test basé sur les modèles

De nombreux travaux de recherche sur ce domaine :

- Comment simuler l'exécution des spécifications ?
- Quels critères de couverture du modèle ?
- Comment maîtriser l'explosion combinatoire ?
- Comment assurer la liaison entre les tests générés à partir du modèle et l'environnement d'exécution des tests ?
- Quelle modélisation des spécifications fonctionnelles dans le contexte de la génération de tests ?
- etc.

Et outils : Conformiq, MaTeLo, Smartesting, Time Partition Testing...

Modélisation : besoins I

Modèle abstrait

Modèle fonctionnel (et non modèle d'architecture et/ou d'implémentation).

Modèle détaillé et précis

Le modèle doit permettre de calculer les cas de test et les résultats attendus.

⇒ établir automatiquement le verdict de test

Modèle validé et vérifié

Si le verdict de test est “Fail” : l'erreur est-elle dans l'implémentation ou dans le modèle ?

Modélisation : besoins II

Modèle adapté aux objectifs de test

Plus le modèle intègre d'informations inutiles par rapport aux objectifs de tests, plus cela génère de "bruit" (informations inutiles) en génération de tests.

Modèle adapté au système sous test

Modèle tenant compte des points de contrôle et d'observation du système sous test.

Modélisation : modèles

- Systèmes de transitions (États / Transitions / Évènements)
- Diagrammes objet et association (Entité-Relation, héritage...)
- Représentation pré-post conditions
- Etc.

Techniques de calcul pour la génération I

Model-checking

- Techniques de parcours du modèle éprouvées.
- La négation de l'objectif de test est vue comme une propriété à démontrer (contre-exemple : cas de test).
- Problème d'explosion combinatoire si méthode valuée
⇒ Model-Checking symbolique.

Interprétation abstraite

- Propagation des équations sur le modèle à partir d'entrées symboliques.
- Nécessite de pouvoir instancier des cas de test concrets.
⇒ Résolution de contraintes.

Techniques de calcul pour la génération II

Programmation en logique avec contraintes

- Calcul des conditions de chemin par résolution de contraintes.
- Le système de contraintes courant représente un ensemble d'états possibles du système.

Apports de la génération automatique : couverture du modèle

- Couverture systématique des spécifications suivant tous les comportements
- Traçabilité entre les exigences et les tests
⇒ Maîtrise de la couverture de tests.

Génération du nombre minimal de données de tests en fonction des critères de couverture sélectionnés.

⇒ Maîtrise de l'explosion combinatoire du nombre de cas de test.

Apports de la génération automatique : cohérence des spécifications

En cas d'évolutions fonctionnelles :

- génération des tests correspondant aux nouveaux comportements,
- tests des comportements inchangés
⇒ Automatise le test des changements fonctionnels et des effets de bord (test de non-régression).

Apports de la génération automatique : maturité du processus de validation

L'automatisation de la génération de tests permet à l'ingénieur validation de se consacrer à l'essentiel :

- le pilotage de la génération (objectifs de test)
- l'analyse des anomalies détectées

⇒ Meilleure maturité du processus de validation.

Une fois la chaîne automatisée

- reproductibilité de la génération
- suppression de l'écriture des scripts

⇒ Réduction de l'effort de test.

Quand mettre en oeuvre la génération automatique à partir de spécifications ?

Critères décisifs

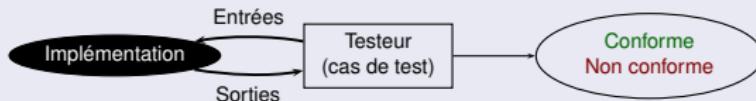
- Fort besoin en validation.
- Croissance forte des fonctionnalités.
- Besoin de réduire l'effort et le temps de test à terme.

Exemple : Test de conformité (de systèmes réactifs)

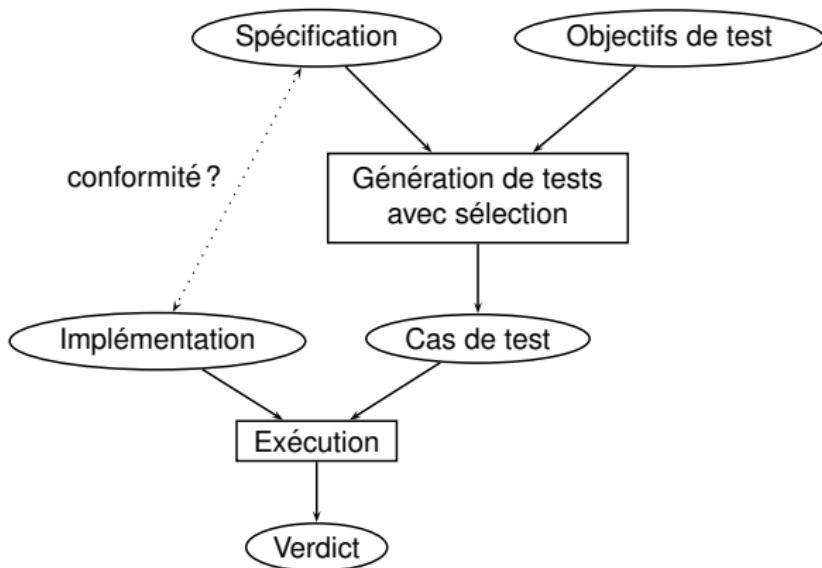
- Test fonctionnel
- Implémentation sous test (IUT) boîte noire
- Spécification (référence, oracle)
- Détection de non-conformité, pas de preuve de conformité de l'IUT

Principe

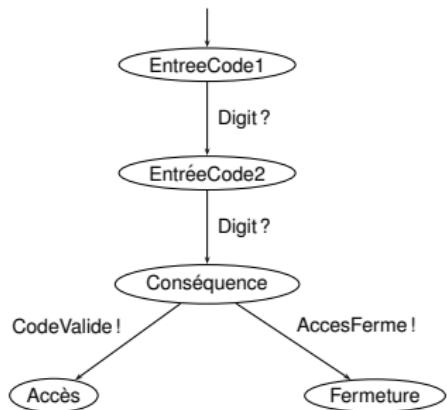
- Exécution de l'IUT
- Contrôle des entrées
- Observation et comparaison des sorties avec celles attendues



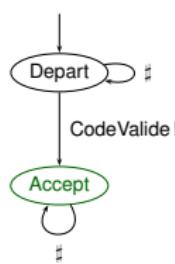
Génération de tests de conformité



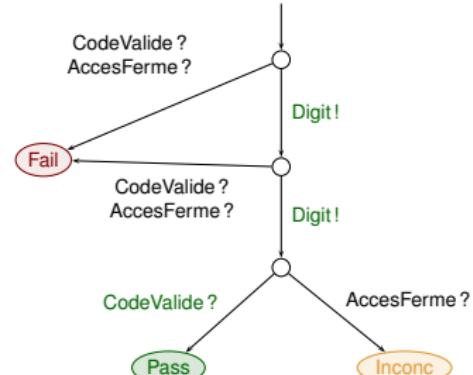
Cas de test



Spécification



Objectif de test



Cas de test

Verdicts

- 1 **Fail** : détection d'une erreur de conformité ;
- 2 **Pass** : satisfaction de l'objectif de test ;
- 3 **Inconc** : le comportement visé par l'objectif ne peut plus être réalisé.

Génération automatique de tests fonctionnels

Avantages :

- Formalisation ;
- Automatisation ;
- Possibilité de génération de tests sur l'ensemble des phases de test, test de non-régression inclus ;
- Maturité du processus : prise de recul, ré-utilisabilité, maintenance.

Inconvénients :

- Effort de modélisation (en particulier des spécifications) ;
- Maturité du processus : compétences nécessaires ;
- Coût important lors de la création du processus ;
- Passage à l'échelle ?

Conclusion sur le test fonctionnel

Le test fonctionnel :

- Permet de définir un ensemble de tests dès les phases de spécification et conception.
- Souvent fait de manière artisanale.
- En constante évolution : beaucoup de travaux de recherche sur la formalisation et l'automatisation de la génération de tests fonctionnels.

Plan

4

Techniques de vérification & validation

- Élaboration de données de test boîte noire
- **Élaboration de données de test boîte blanche**
- Évaluation des données de test
- Analyse structurelle statique

Élaboration de données de test boîte blanche

L'élaboration de données de test boîte blanche se fait à l'aide des méthodes de test structurel dynamique.

Le test structurel dynamique (rappel)

Les méthodes de tests dynamique consistent en l'analyse du code source du logiciel (ou d'un modèle de celui-ci) afin de produire des données de test. Le programme à tester est ensuite exécuté avec les données de test.

Méthodes de test structurel dynamique abordées dans ce cours :

- couverture du graphe de contrôle :
 - flot de contrôle ;
 - flot de données ;
- exécution abstraite ;
- test évolutionniste.

Plan

4

Techniques de vérification & validation

- Élaboration de données de test boîte noire
- **Élaboration de données de test boîte blanche**
 - Couverture du graphe de contrôle : introduction
 - Couverture du graphe de contrôle : flot de contrôle
 - Couverture du graphe de contrôle : flot de données
 - Exécution abstraite
 - Test évolutionniste
- Évaluation des données de test
- Analyse structurelle statique

Couverture du graphe de contrôle

Objectif

Produire des DT qui exécuteront un certain ensemble de comportements du programme. Ces comportements sont symbolisés par des chemins sur le graphe de flot de contrôle.

Approches de couverture :

- tous les nœuds
- tous les arcs
- tous les chemins indépendants
- PLCS (Portion Linéaire de Code Suivie d'un Saut)
(LCSA, Linear Code Subpath And Jump)
- CDCM (Couverture des Décisions et Conditions Modifiées)
(MCDC, Modified Condition Decision Coverage)
- ...

Couverture du graphe de contrôle

Adopter une méthode de test structurel basée sur la couverture consiste à proposer un ensemble de chemins sur le graphe de contrôle.

Satisfaire une méthode de test structurel de couverture

consiste à trouver les jeux de tests qui sensibilisent les chemins de contrôle (ou chemins d'exécution) et qui couvrent les chemins prévus par la méthode adoptée.

Graphe de flot de contrôle

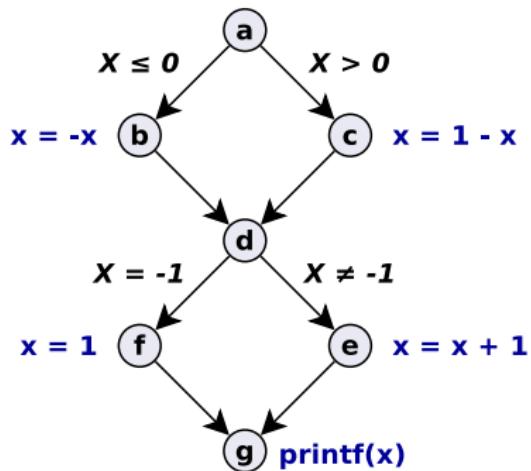
Le graphe de flot de contrôle est un graphe représentant tous les chemins d'exécution (ou chemins de contrôle) d'un programme.

- **Nœud** : ensemble d'instructions séquentielles.
- **Sommet** : nœud (unique) d'entrée du programme.
- **Sortie** : nœud de sortie du programme.
- **Chemin de contrôle** : ensemble de nœuds séquentiels partant du sommet et allant à une sortie
(~ une exécution possible mais pas forcément réalisable).
- **Branche** : sous-chemin de contrôle lié à une décision unique.
- **Saut** : nœud d'entrée, de sortie ou d'arrivée d'un branchement.
- **Arc de saut** : un arc menant à un nœud de type saut.

Graphe de flot de contrôle

Exemple

```
void foo( int x ) {
    if ( x <= 0 )    x = -x;
    else x = 1 - x;
    if ( x == -1 )   x = 1;
    else x = x + 1;
    printf(" x = %d ", x );
}
```



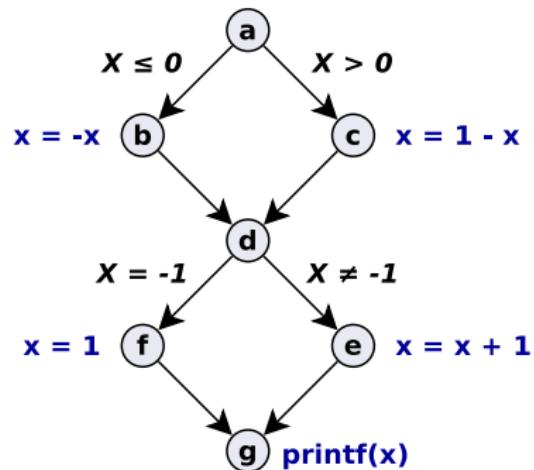
Graphe de flot de contrôle

Exemple

```
void foo( int x ) {
    if ( x <= 0 )    x = -x;
    else x = 1 - x;
    if ( x == -1 )   x = 1;
    else x = x + 1;
    printf(" x = %d ", x );
}
```

Chemins de contrôle :

- [a, c, d, f, g] est un chemin de contrôle.
- [c, d, f, g] n'est pas un chemin de contrôle.



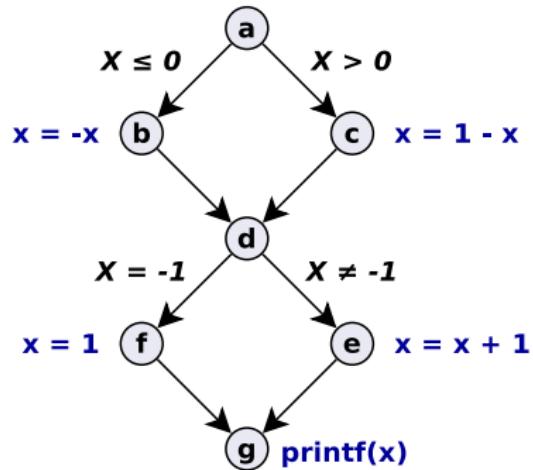
Graphe de flot de contrôle

Exemple

```
void foo( int x ) {
    if ( x <= 0 )    x = -x;
    else x = 1 - x;
    if ( x == -1 )   x = 1;
    else x = x + 1;
    printf(" x = %d ", x );
}
```

4 chemins de contrôle :

- $\mu_1 = [a, c, d, f, g]$
- $\mu_2 = [a, b, d, e, g]$
- $\mu_3 = [a, c, d, e, g]$
- $\mu_4 = [a, b, d, f, g]$



Graphe de flot de contrôle

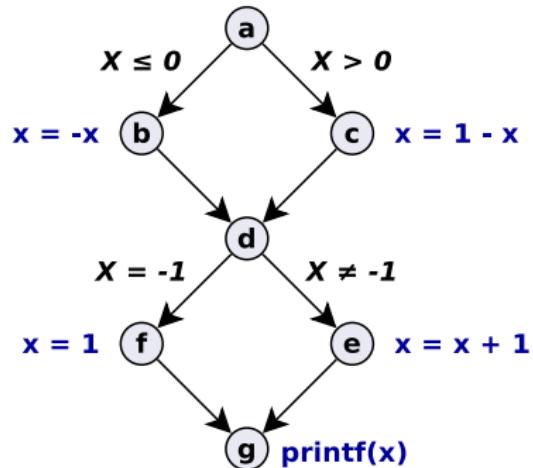
Exemple

```
void foo( int x ) {
    if ( x <= 0 )    x = -x;
    else x = 1 - x;
    if ( x == -1 )   x = 1;
    else x = x + 1;
    printf(" x = %d ", x );
}
```

4 chemins de contrôle :

- $x = 2$ sensibilise $[a, c, d, f, g]$
- $x = 0$ sensibilise $[a, b, d, e, g]$
- $x = 1$ sensibilise $[a, c, d, e, g]$
- \emptyset pour $[a, b, d, f, g]$

(non exécutable)

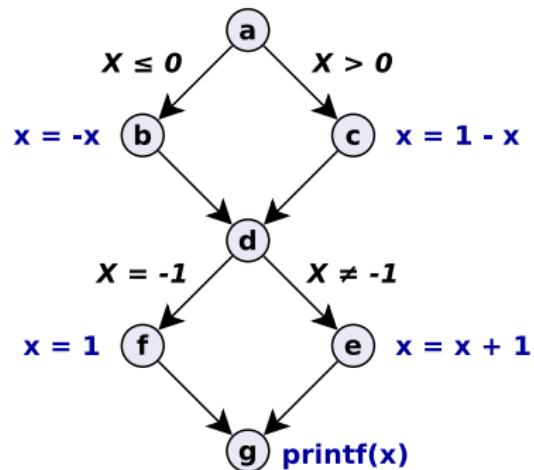


Graphe de flot de contrôle

Exemple

```
void foo( int x ) {
    if ( x <= 0 )    x = -x;
    else x = 1 - x;
    if ( x == -1 )   x = 1;
    else x = x + 1;
    printf(" x = %d ", x );
}
```

Le problème de la détection de chemins non exécutables est un problème difficile, indécidable dans le cas général.



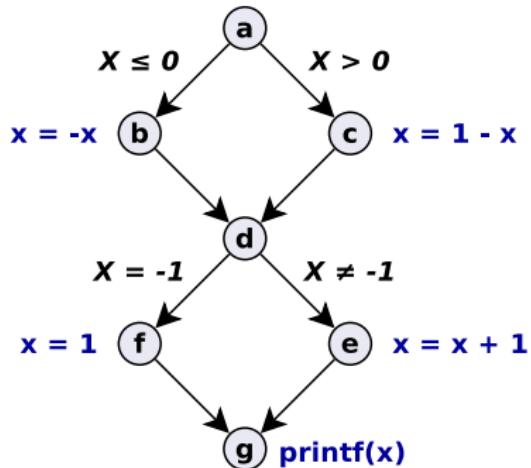
Graphe de flot de contrôle

Exemple

```
void foo( int x ) {
    if ( x <= 0 )    x = -x;
    else x = 1 - x;
    if ( x == -1 )   x = 1;
    else x = x + 1;
    printf(" x = %d ", x );
}
```

Le graphe peut être exprimé sous une forme algébrique :

$$\begin{aligned} G &= abdfg + abdeg + acdfg + acdeg \\ G &= a(bdf + bde + cdf + cde)g \\ G &= a(b + c)d(e + f)g \end{aligned}$$



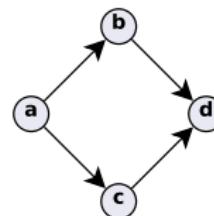
Graphe de flot de contrôle

Une opération d'addition ou de multiplication est associée à toutes les structures primitives apparaissant dans les graphes de contrôle :

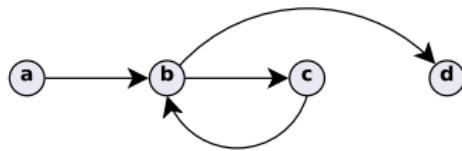
- forme séquentielle : ab



- forme alternative : $a(b + c)d$



- forme itérative : $ab(cb)^*d$

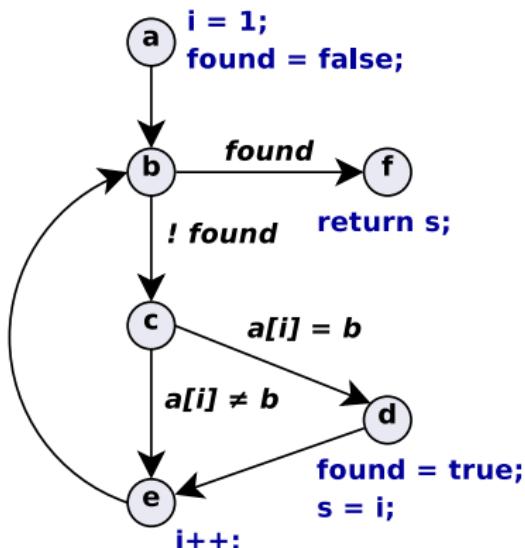


Graphe de flot de contrôle

Exemple avec boucle

```
int foo( int * a, int b ) {
    int s, i = 1;
    bool found = false;
    while( ! found) {
        if ( a[i] == b ) {
            found = true;
            s = i;
        }
        i++;
    }
    return s;
}
```

$$G = ab[c(1 + d)eb]^*f$$



Graphe de flot de contrôle - Exercices

Exercice 4.12 ↗

```
void foo( int n ) {  
    if ( n <= 0 ) n = 1 - n;  
    if ( n & 1 ) n = 3 * n + 1;  
    else n = n / 2;  
    printf("%d", n);  
}
```

Graphe de contrôle ?

Expression des chemins ?

Graphe de flot de contrôle - Exercices

Exercice 4.13 ↗

```
Triangle TestTriangle( int a, int b, int c ) {  
    Triangle triangle;  
  
    if ( ( a < b + c ) && ( b < a + c ) && ( c < a + b ) )  
        triangle = true;  
    else triangle = false;  
  
    if ( triangle ) {  
        if ( ( ( a == b ) || ( a == c ) || ( b == c ) )  
            && ! ( ( a == b ) && ( a == c ) ) )  
            triangle = isocele;  
        if ( ( a == b ) && ( a == c ) )  
            triangle = equilateral;  
        if ( ( a != b ) && ( a != c ) && ( b != c ) )  
            triangle = scalene;  
    }  
    return triangle;  
}
```

Établir le graphe de flot de contrôle de cette fonction.

Graphe de flot de contrôle - Exercices

Exercice 4.14

```
int foo (int* a) {  
    int i = 0;  
    int s = 0;  
    while(i <= 3){  
        if(a[i] > 0) s = s + a[i];  
        i++;  
    }  
    return s;  
}
```

- Établir le graphe de flot de contrôle de ce programme.
- Fournir l'expression des chemins.

Graphe de flot de contrôle - Exercices

Exercice 4.15

```
void init (int[][] matrice) {  
    int i, j;  
    for (i = 0 ; i < IMAX ; i++) {  
        for (j = 0 ; j < JMAX ; j++) {  
            matrice[i][j] = 0;  
        }  
    }  
}
```

- Établir le graphe de flot de contrôle de ce programme.
- Fournir l'expression des chemins.

Graphe de flot de contrôle - Exercices

Exercice 4.16

```
void init (int[][] matrice) {  
    int i, j;  
    for (i = 0 ; i < IMAX ; i++) {  
        matrice[i][0] = 0;  
        for (j = 0 ; j < JMAX ; j++) {  
            if (i == j) {  
                matrice[i][j] = 1;  
            }  
        }  
    }  
}
```

- Établir le graphe de flot de contrôle de ce programme et fournir l'expression des chemins.

Graphe de flot de contrôle - utilisations

- Tests selon les critères de couverture du graphe ([slide 226](#))
- Mesures de complexité, notamment le nombre cyclomatique ([slide 242](#))

Plan

4

Techniques de vérification & validation

- Élaboration de données de test boîte noire
- **Élaboration de données de test boîte blanche**
 - Couverture du graphe de contrôle : introduction
 - **Couverture du graphe de contrôle : flot de contrôle**
 - Couverture du graphe de contrôle : flot de données
 - Exécution abstraite
 - Test évolutionniste
- Évaluation des données de test
- Analyse structurelle statique

Flot de contrôle : tous les nœuds

Critère de couverture tous les nœuds

Objectif :

Couvrir toutes les instructions.

Méthode :

Sensibiliser les chemins de contrôle qui permettent de visiter tous les nœuds du graphe de contrôle.

Taux de couverture :

$$\text{TER1 (Test Effectiveness Ratio 1)} = \frac{\text{nb nœuds couverts}}{\text{nb total de nœuds}}$$

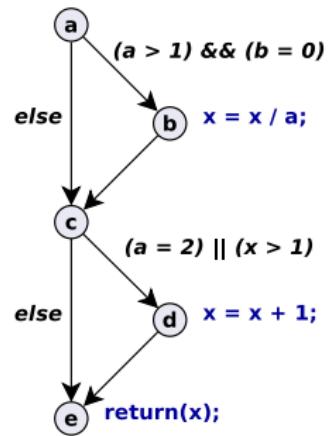
Flot de contrôle : tous les nœuds

Exemple

```
int foo( int a, int b, int x ) {
    if ( ( a > 1 ) && ( b == 0 ) )
        x = x / a;

    if ( ( a = 2 ) || ( x > 1 ) )
        x = x + 1;

    return x;
}
```



$DT : (a = 2, b = 0, x = 1)$ sensibilise $[a, b, c, d, e]$

Flot de contrôle : tous les nœuds

Autre exemple

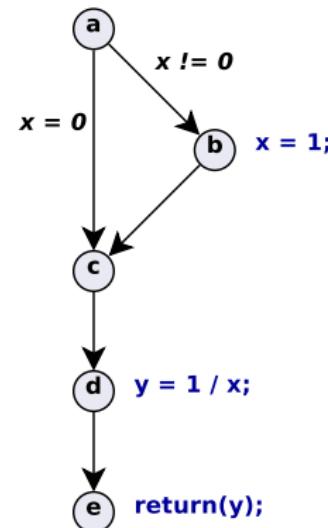
```
int foo( int x ) {
    int y;

    if ( x != 0 ) x = 1;
    y = 1 / x;

    return y;
}
```

$DT : (x = 2)$ sensibilise [a, b, c, d, e]

→ mais faute non détectée.



Flot de contrôle : tous les nœuds

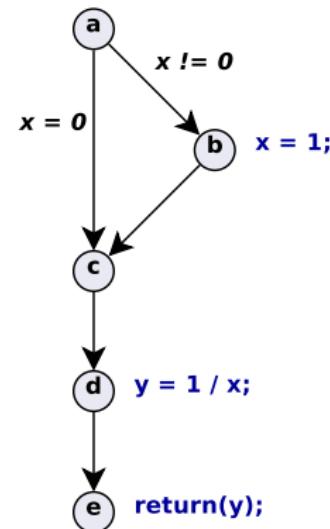
Autre exemple

```
int foo( int x ) {
    int y;

    if ( x != 0 ) x = 1;
    y = 1 / x;

    return y;
}
```

Le critère tous les nœuds est insuffisant pour détecter un grand nombre de fautes.



Flot de contrôle : tous les arcs |

Critère de couverture tous les arcs

Objectif :

Couvrir toutes les valeurs de vérité pour chaque noeud de décision.

Méthode :

Sensibiliser les chemins de contrôle qui permettent de visiter tous les arcs du graphe de contrôle.

Taux de couverture :

$$\text{TER2 (Test Effectiveness Ratio 2)} = \frac{\text{nb arcs couverts}}{\text{nb total d'arcs}}$$

Hiérarchie :

Tous les arcs \Rightarrow Tous les nœuds

Flot de contrôle : tous les arcs

Exemple

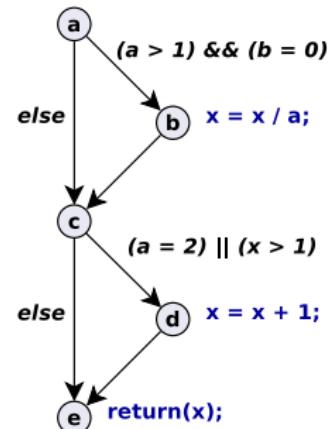
```
int foo( int a, int b, int x ) {
    if ( ( a > 1 ) && ( b == 0 ) )
        x = x / a;

    if ( ( a = 2 ) || ( x > 1 ) )
        x = x + 1;

    return x;
}
```

$DT_1 : (a = 2, b = 0, x = 1)$ sensibilise $[a, b, c, d, e]$

$DT_2 : (a = 1, b = 0, x = 1)$ sensibilise $[a, c, e]$



Flot de contrôle : tous les arcs

Autre exemple

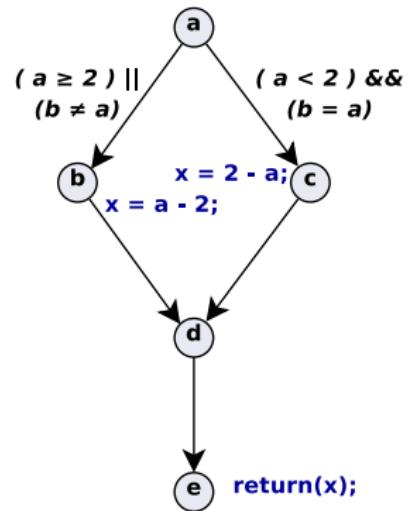
```
int foo( int a, int b ) {
    int x;
    if ( ( a < 2 ) && ( b == a ) )
        x = 2 - a;
    else x = a - 2;
    return x;
}
```

$DT_1 : (a = 1, b = 1)$ sensibilise $[a, c, d, e]$

$DT_2 : (a = 3, b = 2)$ sensibilise $[a, b, d, e]$

→ Le $JT\{DT_1, DT_2\}$ satisfait le critère tous les arcs sans couvrir toutes les décisions possibles.

Exemple : $DT_3 : (a = 3, b = 3)$.

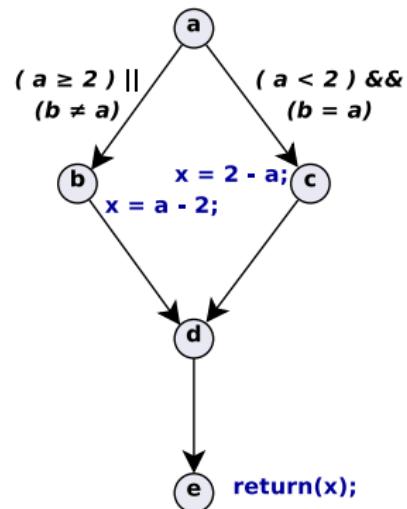


Flot de contrôle : tous les arcs

Autre exemple

```
int foo( int a, int b ) {
    int x;
    if ( ( a < 2 ) && ( b == a ) )
        x = 2 - a;
    else x = a - 2;
    return x;
}
```

Couvrir toutes les décisions possibles : cas des conditionnelles composées.



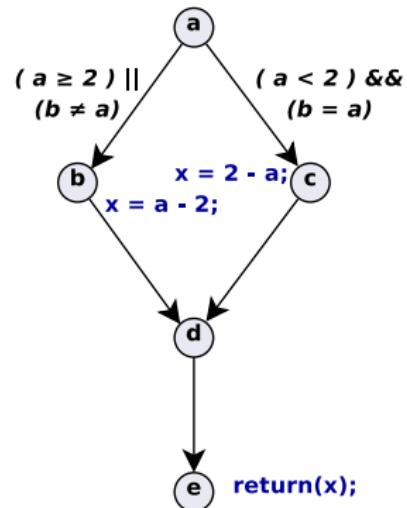
Flot de contrôle : tous les arcs

Autre exemple

```
int foo( int a, int b ) {
    int x;
    if ( ( a < 2 ) && ( b == a ) )
        x = 2 - a;
    else x = a - 2;
    return x;
}
```

Couvrir toutes les décisions possibles : cas des conditionnelles composées.

Couverture des Décisions et Conditions Modifiées
(*MCDC, Modified Condition Decision Coverage*).



Flot de contrôle : tous les arcs - CDCM

CDCM

Le critère de *décisions-conditions modifiées* est satisfait si :

- le critère tous les arcs est satisfait,
- chaque sous-expression dans les conditions prend toutes les combinaisons de valeurs possibles.

Les conditions A et B et A ou B nécessitent les DT :

- A = B = vrai
- A = B = faux
- A = vrai, B = faux
- A = faux, B = vrai

Remarque : difficulté combinatoire lors d'expressions logiques complexes.

Flot de contrôle : tous les arcs

Exemple décomposition des conditions

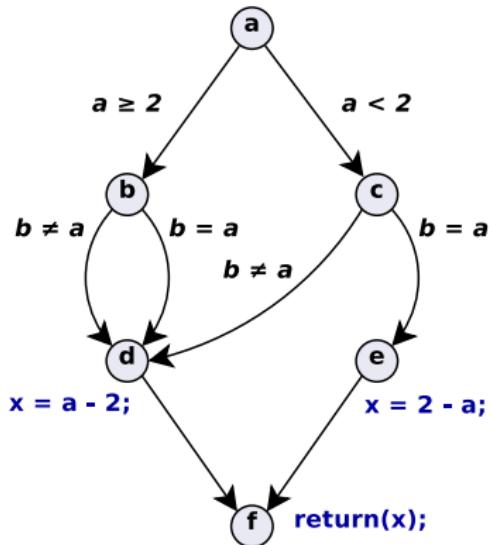
```
int foo( int a, int b ) {
    int x;
    if ( ( a < 2 ) && ( b == a ) )
        x = 2 - a;
    else x = a - 2;
    return x;
}
```

$DT_1 : (a = 1, b = 1)$ sensibilise [a, c, e, f]

$DT_2 : (a = 3, b = 2)$ sensibilise [a, b, d, f]

$DT_3 : (a = 1, b = 0)$ sensibilise [a, c, d, f]

$DT_4 : (a = 3, b = 3)$ sensibilise [a, b, d, f]

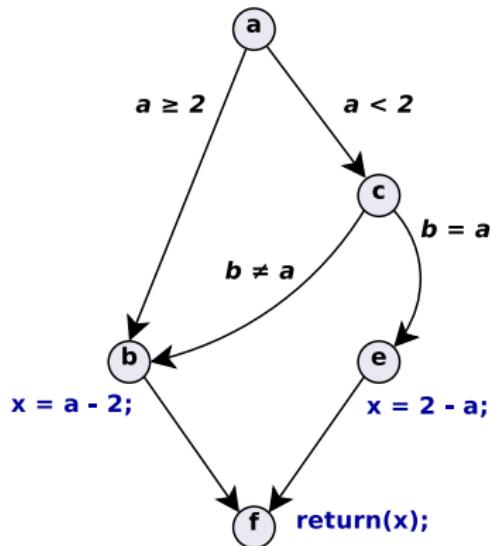


Flot de contrôle : tous les arcs

Exemple décomposition des conditions

```
int foo( int a, int b ) {
    int x;
    if ( ( a < 2 ) && ( b == a ) )
        x = 2 - a;
    else x = a - 2;
    return x;
}
```

Sensibilité au traitement des conditionnelles par le compilateur : le nombre d'arcs dépend de la manière dont l'algorithme est codé **et compilé**.



Flot de contrôle : tous les arcs

Exercice 4.17 ↗

```
float foo( int * a ) {  
    int inf, sup, i;  
    float sum = 0;  
  
    scanf("%d %d", &inf, &sup);  
    i = inf;  
  
    while( i <= sup ) {  
        sum += a[i];  
        i++;  
    }  
    return 1/sum;  
}
```

Flot de contrôle : tous les chemins indépendants I

Critère de couverture tous les chemins indépendants

Objectif :

Parcourir tous les arcs dans chaque configuration possible.

Méthode :

Sensibiliser tous les chemins indép. du graphe de contrôle.

Taux de couverture :

$$T = \frac{\text{nb chemins indépendants couverts}}{\text{nb total de chemins indépendants}}$$

Hiérarchie :

Tous les chemins indép. \Rightarrow Tous les arcs \Rightarrow Tous les nœuds

Flot de contrôle : tous les chemins indépendants II

Chemins indépendants

Deux chemins sont dits indépendants s'ils sont **vectoriellement** indépendants.

En d'autres termes, les chemins doivent différer d'au moins un arc (on ne tient pas compte du nombre d'itérations d'un arc ni de l'ordre dans lequel les arcs sont parcourus).

Flot de contrôle : tous les chemins indépendants III

Méthode :

- ➊ évaluer le nombre cyclomatique du graphe de flot de contrôle G , noté $\nu(G)$, correspondant au nombre de chemins indépendants du graphe ;
- ➋ produire une DT initiale permettant de couvrir le maximum de nœuds de décisions du graphe en une longueur de chemin minimale ;
- ➌ produire une DT modifiant la valeur de vérité de la 1ère instruction de décision du chemin issu de l'étape 2. Itérer ce processus jusqu'à ce que toutes les instructions de décision aient été modifiées. Vous devez arrêter ce processus lorsque le nombre de chemins indépendants couverts est égal à $\nu(G)$.

Calcul du nombre cyclomatique

3 méthodes :

- si graphe planaire, $\nu(G) = f = a - n + 2^{\ddagger}$ avec :
 - f : nombre de faces/régions, a : nbre d'arcs, n : nbre de noeuds ;
- $\nu(G) = d + 1$ avec d : nombre de noeuds de décision

Attention aux structures conditionnelles à choix multiples qui peuvent considérablement augmenter $\nu(G)$.

Note : ce calcul est également utilisé pour la mesure de complexité de Mc Cabe ([slide 290](#)).

‡. Théorème d'Euler

Flot de contrôle : tous les chemins indépendants

Exemple

```
int foo( int a, int b, int x ) {
    if ( ( a > 1 ) && ( b == 0 ) )
        x = x / a;

    if ( ( a = 2 ) || ( x > 1 ) )
        x = x + 1;

    return x;
}
```

$$\nu(G) = 3$$

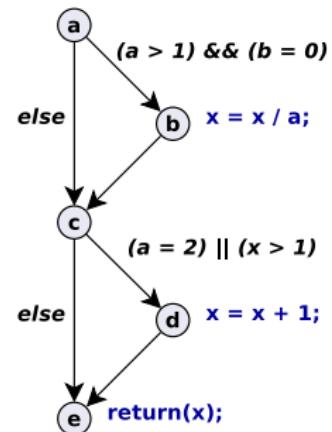
$DT_1 : (a = 1, b = 0, x = 1)$ sensibilise [a, c, e]

$DT_2 : (a = 3, b = 0, x = 3)$ sensibilise [a, b, c, e]

$DT_3 : (a = 1, b = 0, x = 2)$ sensibilise [a, c, d, e]

$DT_4 : (a = 2, b = 0, x = 1)$ sensibilise [a, b, c, d, e]

$DT_4 : (a = 2, b = 0, x = 1)$ sensibilise [a, b, c, d, e]



Flot de contrôle : tous les chemins indépendants

Exercice 4.18 ↗

```
void foo( int n ) {  
    if ( n <= 0 ) n = 1 - n;  
    if ( n & 1 ) n = 3 * n + 1;  
    else n = n / 2;  
    printf("%d", n);  
}
```

JT suivant les critères : tous les noeuds,
tous les arcs et tous les chemins
indépendants ?

Flot de contrôle : tous les chemins indépendants

Exercice 4.19 ↗

```
int foo( int i, int x ) {  
    int found = 0;  
    int a[2];  
  
    scanf("%d %d", a[0], a[1]);  
  
    while ( i < 2 ) {  
        if ( a[i] == x ) found = 1;  
        else found = 0;  
        i++;  
    }  
    return( found );  
}
```

JT suivant le critère tous les chemins indépendants ?

Méthodologie en cas de boucle I

Dans la limite des ressources disponibles :

- appliquer le critère *tous-les-chemins-indépendants* ;
- effectuer des tests sur les boucles pour les cas suivants :
 - pas d'itération de boucle ;
 - 1 itération ;
 - m ($2 \leq m \leq n$) itérations ;
 - $n - 1$ itérations ;
 - n itérations (n étant le nombre maximum d'itérations).

Méthodologie en cas de boucle II

- dans le cas de boucles imbriquées :
 - commencer les tests par rapport à la boucle la plus en profondeur en prenant des valeurs minimales pour les boucles supérieures ;
 - passer ensuite à la boucle directement supérieure en n'effectuant qu'un tour de boucle pour la boucle précédente ;
 - renouveler le processus jusqu'à avoir testé toutes les boucles.

Flot de contrôle

Point fort :

- Les approches basées sur les graphes de contrôle sont partiellement automatisables
(taux de couverture, génération automatique de DT).

Points faibles :

- On ne teste que ce qui est fait mais pas ce qui aurait dû être fait.
- Les boucles engendrent des explosions combinatoires.

Flot de contrôle : PLCS I

Critère de couverture Portion Linéaire de Code suivie d'un Saut
(PLCS ou *LCSAJ, Linear Code Subpath And Jump*)

Objectif :

Augmenter le nombre de chemins de manière à avoir de plus grandes possibilités de détection des erreurs.

Principe (Glossaire CFTL, version 1.0)

PLCS : une Portion Linéaire de Code et Saut, consistant en les trois items suivants -conventionnellement identifiés par des numéros de ligne dans un listing de code source- : le début de la séquence linéaire, la fin de la séquence linéaire et la ligne cible à laquelle le contrôle est passé en fin de séquence linéaire.

Flot de contrôle : PLCS II

PLCS

Un PLCS est un chemin partant d'un nœud de type saut et aboutissant à nouveau à un nœud de même type. L'arc entre l'avant-dernier et le dernier nœud doit être le seul arc de saut du chemin.

Méthode :

Sensibiliser les chemins de contrôle qui permettent de couvrir la totalité des PLCS du graphe de contrôle.

Taux de couverture :

$$\text{TER3} \ (\textit{Test Effectiveness Ratio 3}) = \frac{\text{nb PLCS couverts}}{\text{nb total PLCS}}$$

Hiérarchie :

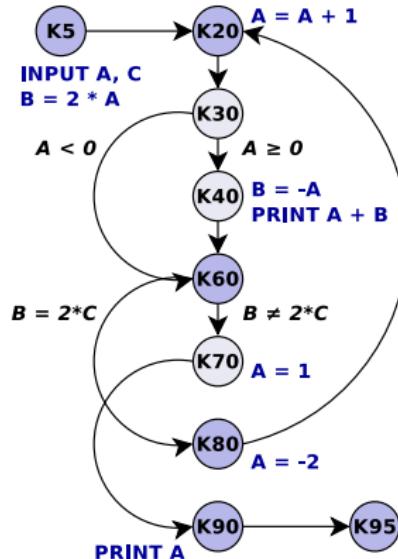
PLCS \Rightarrow Tous les arcs \Rightarrow Tous les nœuds

Flot de contrôle : PLCS

Exemple

```

005 INPUT A, C
010 B = 2 * A
020 A = A + 1
030 IF A < 0 THEN GOTO 60
040 B = -A
050 PRINT A + B
060 IF B = 2 * C THEN GOTO 80
070 A = 1 : GOTO 90
080 A = -2 : GOTO 20
090 PRINT A
095 END
    
```

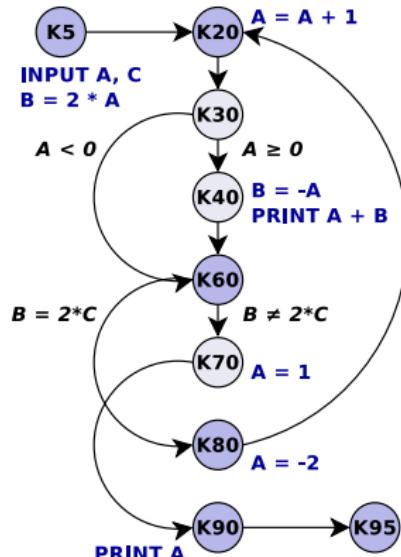


Flot de contrôle : PLCS

Exemple

10 PLCS :

- 1 : [K5, K20, K30, K60]
- 2 : [K5, K20, K30, K40, K60, K80]
- 3 : [K5, K20, K30, K40, K60, K70, K90]
- 4 : [K20, K30, K60]
- 5 : [K20, K30, K40, K60, K80]
- 6 : [K20, K30, K40, K60, K70, K90]
- 7 : [K60, K80]
- 8 : [K60, K70, K90]
- 9 : [K80, K20]
- 10 : [K90, K95]



Flot de contrôle : PLCS

Autres critères de type PLCS :

- TER4 = $\frac{\text{nb chemins composés de 2 PLCS couverts}}{\text{nb total de chemins composés de 2 PLCS}}$
- TER5 = $\frac{\text{nb chemins composés de 3 PLCS couverts}}{\text{nb total de chemins composés de 3 PLCS}}$

Hiérarchie :

TERn ⇒ ... ⇒ TER5 ⇒ TER4 ⇒ TER3

Inconvénient majeur :

La définition de la PLCS est purement syntaxique.
 La classification des nœuds se fonde donc sur des critères essentiellement textuels (existence des GOTO, séquence des instructions) et non sur la structure réelle du graphe de contrôle.

Plan

4

Techniques de vérification & validation

- Élaboration de données de test boîte noire
- **Élaboration de données de test boîte blanche**
 - Couverture du graphe de contrôle : introduction
 - Couverture du graphe de contrôle : flot de contrôle
 - **Couverture du graphe de contrôle : flot de données**
 - Exécution abstraite
 - Test évolutionniste
- Évaluation des données de test
- Analyse structurelle statique

Flot de données

Principe

Les critères de couverture basés sur le flot de données sélectionnent les données de test en fonction des définitions et des utilisations des variables du programme.

Définitions sur les occurrences de variables :

- Une variable est **définie** lors d'une instruction si la valeur de la variable est modifiée (affectation),
- Une variable est **référencée** si la valeur de la variable est utilisée.

Définitions sur les utilisations de variables :

- **p-utilisation** : la variable référencée est utilisée dans le prédicat d'une instruction de décision (if, while, ...).
- **c-utilisation** : autres cas (par exemple, un calcul).

Flot de données

Instruction utilisatrice :

Une instruction I_2 est utilisatrice d'une variable x par rapport à une instruction I_1 , si la variable x est définie en I_1 et directement référencée dans I_2 , c'est-à-dire sans redéfinition de x entre I_1 et I_2 .

Chemin d'utilisation :

Un chemin d'utilisation dr-strict relie l'instruction de définition d'une variable à une instruction utilisatrice.

Exemple

- | | |
|-------------------|---|
| (1) $x := 7;$ | ● (5) est <u>c-utilisatrice</u> de x par rapport à (1). |
| (2) $a := x + 2;$ | ● (5) est <u>c-utilisatrice</u> de a par rapport à (4). |
| (3) $b := a * a;$ | ● Le chemin [1, 2, 3, 4, 5] est <u>dr-strict</u> pour la variable x . |
| (4) $a := a + 1;$ | ● Le chemin [1, 2, 3, 4, 5] n'est pas <u>dr-strict</u> pour la variable a . |
| (5) $y := x + a;$ | |

Flot de données : *toutes-les-définitions, tous-les-utilisateurs*

Critère toutes les définitions :

Pour chaque définition, il y a au moins un chemin dr-strict dans un test.

Critère tous les utilisateurs :

Pour chaque définition et pour chaque référence accessible (utilisation) à partir de cette définition, il y a un chemin dr-strict dans un test
(couverture de tous les utilisateurs : noeuds c-utilisateurs ou arcs p-utilisateurs).

Hiérarchie :

tous les utilisateurs ⇒ toutes les définitions

Flot de données

Exemple

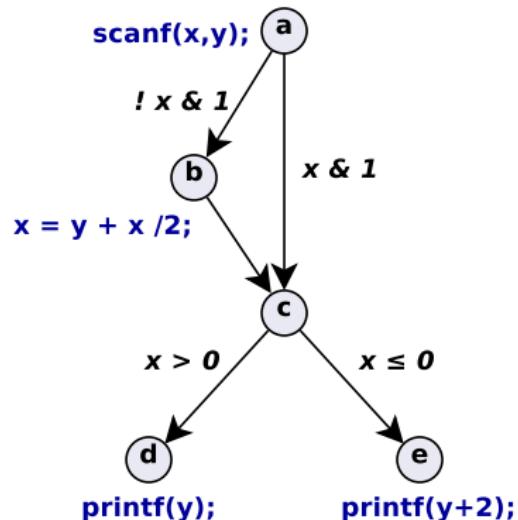
```
scanf("%d %d", &x, &y);
if ( ! x & 1 ) x = y + x /2;
if ( x > 0 ) printf("%d", y);
else printf("%d", y + 2);
```

Toutes les définitions :

- [a, c, e]
- [a, b, c, e]

Tous les utilisateurs :

- [a, c, e]
- [a, b, c, e]
- [a, c, d]
- [a, b, c, d]



Flot de données

Exercice 4.20 ↗

```
scanf("%d %d", &x, &y);
if ( x <= 0 ) y = y + x;
if ( y & 1 ) n = x / 2;
else n = 3 * y + 1;
printf("%d", n);
```

JT suivant les critères :

- toutes les définitions ?
- tous les utilisateurs ?

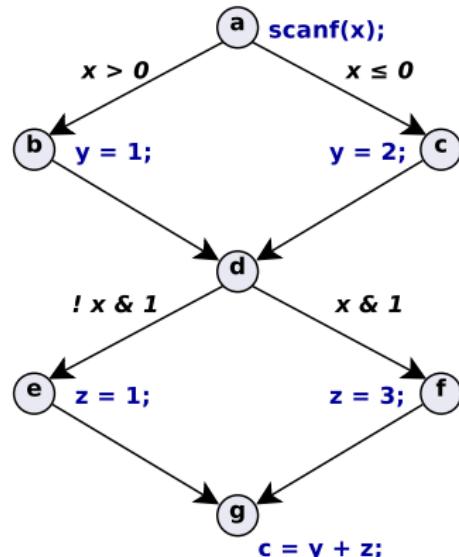
Flot de données

Exemple (limite tous les utilisateurs)

```
scanf ("%d", &x);
if ( x > 0 ) y = 1;
else y = 2;
if ( x & 1 ) z = 3;
else z = 1;
c = y + z;
```

Tous les utilisateurs :

- [a, b, d, e, g]
- [a, c, d, f, g]



→ ne couvre pas tous les chemins d'utilisation (ex : ④ → ⑥ → ⑦ → ⑧).
 ⇒ critère tous les du chemins.

Flot de données

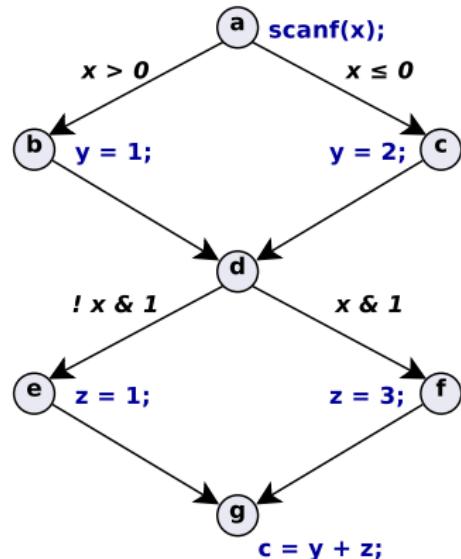
Exemple (limite tous les utilisateurs)

```
scanf ("%d", &x);
if ( x > 0 ) y = 1;
else y = 2;
if ( x & 1 ) z = 3;
else z = 1;
c = y + z;
```

Tous les chemins :

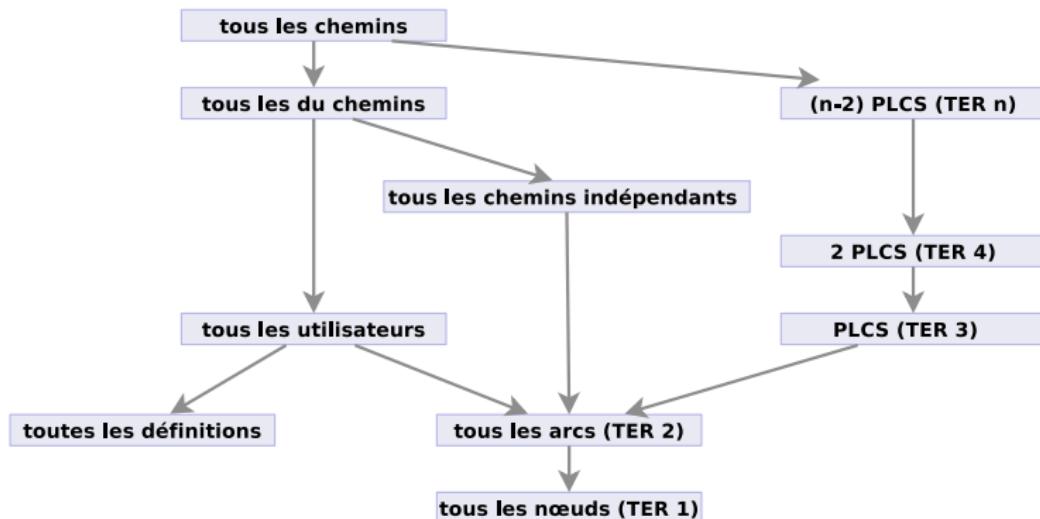
ajoute au critère tous les utilisateurs la couverture de tous les chemins possibles entre la définition et la référence, en se limitant aux chemins sans cycle.

- $[a, b, d, e, g]$
- $[a, c, d, f, g]$
- $[a, b, d, f, g]$
- $[a, c, d, e, g]$



Hiérarchie des critères

Flot de contrôle et flot de données :



Plan

4

Techniques de vérification & validation

- Élaboration de données de test boîte noire
- **Élaboration de données de test boîte blanche**
 - Couverture du graphe de contrôle : introduction
 - Couverture du graphe de contrôle : flot de contrôle
 - Couverture du graphe de contrôle : flot de données
 - **Exécution abstraite**
 - Test évolutionniste
- Évaluation des données de test
- Analyse structurelle statique

Exécution abstraite

Objectif :

Obtenir des DT (test dynamique) ou des analyses (test statique ou vérification) plus pertinentes.

Méthode :

Enrichir les données critiques d'un logiciel en leur adjoignant un modèle abstrait dont les calculs algébriques s'opèrent en même temps que les évaluations concrètes de l'exécution.

Exécution abstraite - Exemples I

- Modèle de test aux limites : force la valeur de certaines conditions afin d'étudier le comportement du logiciel dans des configurations qu'on espère non-nominales.

Points forts :

- Formalisation partielle (et partielle) de la notion de limite ;
- Bon cadre pour vérification dynamique.

Points faibles :

- Applicable seulement sur langages supportant la surcharge d'opérateurs ;
- Calculs algébriques en fond ralentissent temps d'exécution ;
⇒ technique difficilement applicable pour systèmes temps-réel.

Exécution abstraite - Exemples II

- Analyse des domaines finis : géométrisation des domaines formés par les DT ce qui permet la détection d'erreurs lors de décalages de droites dans le plan.

Points forts :

- Simplicité apparente de la détection d'erreur des DT.

Points faibles :

- Espace de faible dimension (conditions à au plus 2 variables) ;
- Conditions portant souvent sur des variables intermédiaires et non sur des entrées
 - ⇒ exécution symbolique inévitable
 - ⇒ méthode difficilement automatisable s'adressant à des logiciels à forte composante arithmétique.

Plan

4

Techniques de vérification & validation

- Élaboration de données de test boîte noire
- **Élaboration de données de test boîte blanche**
 - Couverture du graphe de contrôle : introduction
 - Couverture du graphe de contrôle : flot de contrôle
 - Couverture du graphe de contrôle : flot de données
 - Exécution abstraite
 - **Test évolutionniste**
- Évaluation des données de test
- Analyse structurelle statique

Test évolutionniste I

Objectif :

Obtenir des DT intéressantes plus rapidement que par les approches aléatoires.

Méthode :

Modéliser l'environnement d'un logiciel et faire évoluer cet environnement de manière à reproduire des schémas de comportements (structurels ou fonctionnels) que l'utilisateur a ciblés.

Cette évolution se base sur les interactions avec le logiciel, il adapte les données (entrées du logiciel) selon le comportement (les sorties) qu'il observe.

Test évolutionniste II

Points forts :

Vraie alternative aux approches aléatoires :

- bien meilleures performances en temps d'exécution ;
- bien meilleures performances en objectifs de test satisfaisants.

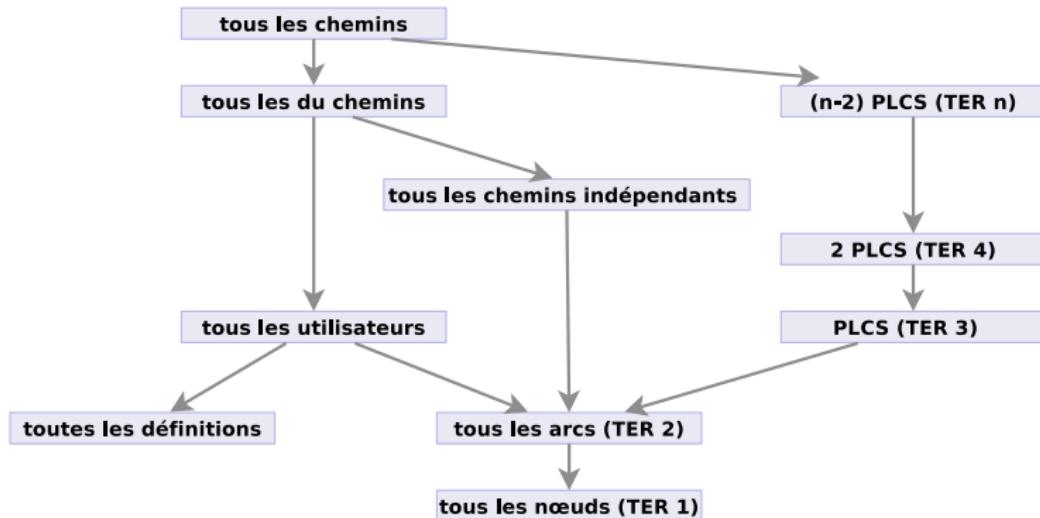
⇒ Le test évolutif réalise plus rapidement un plus grand pourcentage d'objectifs de couverture structurelle.

Points faibles :

- Éventuelle difficulté à modéliser l'ensemble des stimuli captés par un logiciel et ses réponses associées ;
- Réponses doivent être assez riches : test évolutif appliqué à logiciel à 1 réponse (oui ou non) serait analogue à un test aléatoire.

Hiérarchie des critères

Flot de contrôle et flot de données :



Conclusion sur le test structurel dynamique

Le test structurel dynamique :

- Permet de définir un ensemble de tests après ou lors de l'implémentation.
- Techniques de couverture très populaires.
- Certaines phases sont automatisables (calcul du taux de couverture de test ou calcul des DT par test évolutionniste par ex.).
- Travaux de recherche sur les techniques de couverture de graphe dans l'objectif d'automatiser la sélection des DT.

Plan

4

Techniques de vérification & validation

- Élaboration de données de test boîte noire
- Élaboration de données de test boîte blanche
- Évaluation des données de test**
- Analyse structurelle statique

Plan

4

Techniques de vérification & validation

- Élaboration de données de test boîte noire
- Élaboration de données de test boîte blanche
- **Évaluation des données de test**
 - Mutant
- Analyse structurelle statique

Mutant

Objectif :

Évaluer les DT vis-à-vis de la liste des fautes les plus probables qui ont été envisagées (modèles de fautes).

Principe

Considérer des variantes du programme en introduisant des fautes (ou mutations, ou opérateurs mutationnels) dans celui-ci.

Mise en œuvre :

- Un seul défaut donne naissance à un mutant.
- Les mutations correspondent à des transformations syntaxiques élémentaires du programme (remplacement d'un opérateur, remplacement d'un nom de variable ou de constante, etc.).

Mutant

Règles de mutations d'opérateurs (instructions) :

- *expression deletion*
- *boolean expression negation*
- *term associativity shift*
- *arithmetic operator by arithmetic operator*
- *relational operator by relational operator*
- *logical operator by logical operator*
- *logical negation*
- *variable by variable replacement*
- *variable by constant replacement*
- *constant by required constant replacement*

Mutant

Utilisation :

Exécution du mutant et comparaison avec le résultat du programme d'origine.

- Si le résultat diffère : « *le mutant est tué* ».
- Si le résultat concorde : « *le mutant survit* ».

Comparaison des critères :

Le meilleur critère est celui qui produit des JT tuant le plus de mutants.

Comparaison des JT :

Le meilleur JT est celui qui tue le plus de mutants.

Mutant

Points forts :

- Permet d'évaluer la qualité des critères et des JT.

Points faibles :

- Coûteuse à mettre en œuvre.
- Fondée sur l'hypothèse forte que seules des fautes mineures sont présentes dans le programme.

Plan

4

Techniques de vérification & validation

- Élaboration de données de test boîte noire
- Élaboration de données de test boîte blanche
- Évaluation des données de test
- Analyse structurelle statique

Analyse structurelle statique

L'analyse structurelle statique se fait à l'aide des méthodes de test structurel statique.

Le test structurel statique

Les méthodes de test structurel statique consistent en l'analyse du code source du logiciel (ou d'un modèle de celui-ci) afin d'y détecter des fautes, sans exécution du programme.

Méthodes de test structurel statique abordées dans ce cours :

- revue ;
- mesures de complexité ;
- analyse du flot de données ;
- exécution symbolique ;
- preuve formelle ;
- interprétation abstraite.

Plan

4

Techniques de vérification & validation

- Élaboration de données de test boîte noire
- Élaboration de données de test boîte blanche
- Évaluation des données de test
- Analyse structurelle statique
 - Activités de revue
 - Mesures de complexité
 - Analyse du flux de données
 - Exécution symbolique
 - Preuve formelle
 - Interprétation abstraite

Revue

Revue ou inspection (*review or inspection*), IEEE 729

Examen détaillé d'une spécification, d'une conception ou d'une implémentation par une personne ou un groupe de personnes, afin de déceler des fautes, des violations de normes de développement ou d'autres problèmes.

- esprit critique
- diplomatie
- coûteux
- efficace (60 à 95% des fautes décelées)

Règle 2/3-1/3 : 2/3 des fautes dans 1/3 du code (ou du document).

(~ règle des 80-20 : 80% des fautes dans 20% du code).

Revue

La revue de **code** appartient à la famille du test structurel statique (boîte blanche).

Le test structurel statique

Les méthodes de tests statiques consistent en l'analyse du code source du logiciel (ou d'un modèle de celui-ci) afin d'y détecter des erreurs, sans exécution du programme.

Principe de la revue

Détection de fautes

- dans les documents
(spécification, conception, code source, test) ;
- dans les choix
(architecture, algorithme, synchronisation, protection de données...).

Évaluation de la capacité d'un logiciel

- à être maintenu,
- à évoluer.

Principe de la revue

Une technique très efficace si bien menée :

- vitesse 100 à 200 lignes de code/heure ;
- pas plus d'une heure 1/2 (150 à 300 lignes par session) ;
- une liste de points à vérifier.

Quelques règles de fonctionnement :

- lecture croisée ;
- jamais sur sa production ;
- après une compilation sans *warning* ni erreur ;
- en réunion avec les membres de l'équipe.

Organisation d'une revue : liste de points à vérifier

Exemple de points à vérifier (code) :

- *Data Reference Errors*
- *Data Declaration Errors*
- *Computation Errors*
- *Comparison Errors*
- *Control Flow Errors*
- *Procedure (Subroutine) Parameter Errors*
- *Input/Output Errors*
- *Interface Consistency and Completeness*
- *Data Flow*
- *Missing Functions*
- *Input Checking*
- *Documentation Consistent*

Organisation d'une revue : composition d'une équipe

Composition typique d'une équipe d'inspection :

- Le modérateur :
 - coordonne l'inspection
 - (produit, équipe, procédure d'inspection, respect des échéances, critères de validation) ;
- Le secrétaire :
 - prend les notes durant les réunions d'inspection ;
- Les inspecteurs :
 - évaluent le produit et
 - participent à la rédaction du rapport ;
- Le lecteur :
 - guide les inspecteurs dans l'évaluation ;
- Le vérificateur :
 - coordonne la rédaction du rapport et
 - assure le suivi des modifications demandées ;
- L'auteur :
 - apporte sa connaissance sur la réalisation du produit.

Organisation d'une revue : processus

Exemple de processus d'inspection :

- 1 Élaborer l'agenda et formaliser la motivation de l'inspection.
- 2 Constituer l'équipe d'inspection.
- 3 Présenter le produit à l'équipe d'inspection et positionner les membres.
- 4 Mener l'inspection en suivant les points à vérifier.
- 5 Collecter les données résultantes de l'inspection.
- 6 Rédiger le rapport d'inspection
(modifications, échéances, documents de références).
- 7 Vérifier les modifications apportées au produit suite à l'inspection.

Remarque : Un processus d'inspection peut être appliqué aux tests eux-mêmes.

Plan

4

Techniques de vérification & validation

- Élaboration de données de test boîte noire
- Élaboration de données de test boîte blanche
- Évaluation des données de test
- **Analyse structurelle statique**
 - Activités de revue
 - **Mesures de complexité**
 - Analyse du flot de données
 - Exécution symbolique
 - Preuve formelle
 - Interprétation abstraite

Mesures de la complexité

Principe

Des études statistiques ont montré que plus le code d'un logiciel est complexe, plus il y a de risques d'erreurs.

Idée : Il suffit donc de mesurer la complexité du code et de la réduire afin de réduire le risque d'erreur.

Métriques :

Mesure du code (ou de document de spécification ou de conception) évaluant certains critères (ne mesure pas seulement la complexité).

- Nombre de lignes de code (KLOC, *Kilo Lines Of Code*)
- Mc Cabe, Halstead, Henry & Kafura, Chidamer & Kemerer, Li & Henry, ...

Mesures de la complexité : mesure de Mc Cabe

Principe de la mesure de Mc Cabe

La complexité d'un programme est directement liée aux différentes exécutions possibles (chemins indépendants) qui peuvent en résulter.

⇒ basée sur le calcul du **nombre cyclomatique** ([slide 242](#)) du **graphe de flot de contrôle** ([slide 210](#)).

Exemples de limites par fonction :

- *Hewlett-Packard* : $\nu < 15$,
- *Euro Star* : $\nu < 20$.

Pour un fichier, la limite standard est de 100.

Mesures de la complexité : mesures de Halstead

Principe des mesures de Halstead

La complexité d'un programme est directement liée à la distribution des différentes instructions et variables qu'il met en jeu.

→ dépendance au « volume » d'information.

Mesures de la complexité : mesures de Halstead

Mesures de Halstead pour la complexité :

- n_1 : nombre d'opérateurs distincts
- n_2 : nombre d'opérandes distinctes
- N_1 : nombre total d'opérateurs utilisés
- N_2 : nombre total d'opérandes utilisées
- longueur d'un programme : $N = N_1 + N_2$
- vocabulaire d'un programme : $n = n_1 + n_2$
- volume d'un programme : $V = N \times \log_2(n)$
- difficulté d'un programme : $D = n_1/2 \times N_2/n_2$
- effort : $E = D \times V$
- temps pour implémenter (s) : $T = E/18$
- nombre de bugs potentiels (nb de bugs fournis) : $B = E^{2/3}/3000$

Mesures de Halstead

Exercice 4.21

```
int foo(int x) {  
    if ( x > 2 ) x = x * 4;  
    else {  
        for (int i = 5; i > x; i--)  
            x = x * i;  
    }  
    return x;  
}
```

Mesures de la complexité : mesures de Halstead

Mesures de Halstead pour la maintenance :

$$MI_{code} = 171 - 5.2 * \ln(aveV) - 0.23aveCCe - 16.2 * \ln(aveLOC)$$

$$MI_{comment} = 50 * \sin(\sqrt{2.4 * perCM}) \text{ avec :}$$

- $aveV$: volume moyen ;
- $aveCCe$: complexité cyclomatique moyenne ;
- $aveLOC$: nombre moyen de lignes de code ;
- $perCM$: pourcentage moyen de lignes de commentaires.

→ Plus les valeurs sont élevées, plus le logiciel est facile à maintenir
 (85 : facile, 65..85 : correct, < 65 : difficile)

Mesures de la complexité : Henry & Kafura

Principe des métriques d'Henry & Kafura

La complexité d'un programme est liée à son contexte d'exécution.

- calcul du nombre d'informations « entrantes » et « sortantes ».
- adaptées au contexte objet.

Mesures de la complexité : Chidamber & Kemerer

Métriques de Chidamber & Kemerer (contexte objet) :

- WMC (*Weighted Method per Class*) :

nombre de méthodes d'une classe

pondéré par le nombre cyclomatique et le nombre d'attributs

(valeur élevée → effort de développement et de maintenance, faible réutilisation) ;

- DIT (*Depth of Inheritance Tree*) :

nombre de classes ancêtres pour atteindre la racine

(valeur élevée → compréhension difficile, faible réutilisation)

(valeur faible → risque d'une mauvaise utilisation du concept objet) ;

- NOC (*Number Of Children*) :

nombre de descendants d'une classe

(valeur élevée → bonne réutilisation, importance de la classe donc du test)

(valeur faible → risque d'une mauvaise utilisation du concept objet) ;

Mesures de la complexité : Chidamber & Kemerer

Métriques de Chidamber & Kemerer (contexte objet) :

- CBO (*Coupling Between Object Classes*) :
nombre de couplage avec le reste du système
(valeur élevée → difficulté de maintenance, test, réutilisation) ;
- RFC (*Response For a Class*) :
nombre de méthodes qui peuvent être appelées
(valeur élevée → difficultés de compréhension, effort de test) ;
- LCOM (*Lack of Cohesion in Methods*) :
nombre de méthodes qui ne partagent pas un attribut commun
moins le nombre de méthodes qui le font
(valeur élevée → faible cohésion).

Mesures de la complexité : Li & Henry

Métriques de Li & Henry (complètent Chidamber & Kemerer) :

- MPC (*Message Passing Coupling*) : nombre d'appels à des méthodes publiques d'autres classes ;
- DAC (*Data Abstraction Coupling*) : nombre d'instances d'autres classes (attributs, variables locales) ;
- NOM (*Number Of local Methods*) ;
- SIZE1 (nombre de point-virgules dans une classe) ;
- SIZE2 (nombre d'attributs + NOM).

Plan

4

Techniques de vérification & validation

- Élaboration de données de test boîte noire
- Élaboration de données de test boîte blanche
- Évaluation des données de test
- **Analyse structurelle statique**
 - Activités de revue
 - Mesures de complexité
 - **Analyse du flux de données**
 - Exécution symbolique
 - Preuve formelle
 - Interprétation abstraite

Analyse du flux de données

Objectif :

Analyser le code source afin de détecter des anomalies universelles du flux de données.

Méthode :

Analyser le flux des variables d'un programme tel qu'il est défini par le séquencement des différentes définitions et utilisations.

Peut être effectué de manière statique mais également dynamique (exécution abstraite).

Analyse du flux de données

Principe :

Calculer les dr-chaînes de toutes les variables apparaissant dans le programme.

dr-chaîne

La dr-chaîne d'une variable reflète son flux développé lors de l'exécution du programme.

Elle est composée des symboles **d** (définition) et **r** (référence/utilisation).

Analyse du flux de données : exemple simple

Exemple

```
void foo() {
    int x, y, z, a, b, c;

    scanf("%d", &c);
    x = 7;
    y = x + a;
    b = x + y + a;
    c = y + 2*x + z;
    printf("%d\n", c);
}
```

Anomalies détectées

Anomalies pouvant survenir dans un programme principal :

- ① *r..* : valeur indéfinie lors de la 1ère utilisation ;
- ② *..dd..* : définitions consécutives, seule la dernière est utile ;
- ③ *..d* : dernière définition inutile.

Si fonction non principale, on compare (automatiquement) les dr-chaînes obtenues avec les dr-chaînes théoriques calculées selon les intentions du programmeur (en-têtes des fonctions) :

- si (1) : la variable peut être une entrée de la fonction ;
- si (3) : la variable peut être une sortie de la fonction.

Analyse du flux de données : conditionnelles

Exemple

```
void foo() {
    int x, a, b, c;

    a = b;
    if(a == 1) a = 6
    else a = b;

    if(x == 1) b = a
    else a = b + c + 2;
}
```

$$\begin{aligned}\pi(a) &= dr(d+d)(r+d) = drdd + drdr \\ \pi(b) &= r(1^a+r)(d+r) = rd + rr + rrd + rrr \\ \pi(x) &= r \\ \pi(c) &= r + 1\end{aligned}$$

a. 1 signifie ni définie ni référencée.

Analyse du flux de données : boucles

Exemple

```
void foo(){
    int i, j, n, sum;

    scanf("%d", &n);
    sum = 0;
    i = 1;
    while(i<=n) {
        sum += i;
        i = i + 1;
    }
    printf("%d\n", sum);
}
```

$$\pi(\text{sum}) = (dr)^+ = dr + drdr + drdrdr + \dots$$

$$\pi(i) = dr + drrrdr + drrrdrrrdr + \dots$$

$$\pi(n) = dr^+$$

Analyse du flux de données : boucles

Exemple

```
void foo() {
    int i, j, n, sum;

    scanf("%d", &n);
    sum = 0;
    i = 1;
    while(i<=n) {
        sum += i;
        j = i+1;
    }
    printf("%d\n", sum);
}
```

$$\pi(\text{sum}) = dr + (dr)^2$$

$$\pi(i) = dr + drrrr + dr(rrr)^2$$

$$\pi(n) = dr + dr^2$$

Analyse du flux de données : boucles

Exemple

```
void foo() {
    int i, j, n, sum;

    scanf ("%d", &n);
    sum = 0;
    i = 1;
    while(i<=n) {
        sum += i;
        i = i-1;
    }
    printf ("%d\n", sum);
}
```

$$\pi(\text{sum}) = dr + (dr)^2$$

$$\pi(i) = dr + drrrdr + dr(rrdr)^2$$

$$\pi(n) = dr + dr^2$$

Analyse du flux de données : limites

Exemple

```
void foo() {  
    int a,b;  
  
    scanf("%d", &b);  
    a = 8;  
    if(b > 10) a = 7  
    else b = a;  
    printf("%d\n%d\n", a, b);  
}
```

$$\pi(a) = ddr + drr$$

$$\pi(b) = drr + drdr$$

Analyse du flux de données : limites

Technique d'analyse statique \Rightarrow les pointeurs et les tableaux ne peuvent être traités (impossible de savoir quel est la variable ou l'élément du tableau défini ou utilisé).

Exemple

```
a[i] = 0;  
a[i+1] = 7;
```

$$\pi(a) = dd$$

\Rightarrow approche dynamique du flux de données : exécution abstraite d'un automate du flux de données (instrumentation d'une fonction calculant dynamiquement les dr-chaines d'un tableau).

Analyse du flux de données : exercice

Exercice 4.22 ↗

```
int fact(int n){  
    int f,i;  
  
    i = n;  
    f = 1;  
    while(i > 1){  
        f = f * i;  
        i--;  
    }  
    return f;  
}
```

Analyse du flux de données : exercice

Exercice 4.23 ↗

```
int fact2(int n){  
    int f,i;  
  
    i = 1;  
    while(i <= n){  
        if(i == 1)  
            f = 1;  
        else  
            f = f * i;  
        i++;  
    }  
    return f;  
}
```

Analyse du flux des données

Points forts :

- Facilement automatisable ;
- Technique statique et dynamique.

Points faibles :

- Spectre de vérification réduit.

Bilan :

- Dédié à tout type d'algorithme car facilement automatisable mais ne vérifie qu'un minimum d'erreurs universelles.

Plan

4

Techniques de vérification & validation

- Élaboration de données de test boîte noire
- Élaboration de données de test boîte blanche
- Évaluation des données de test
- **Analyse structurelle statique**
 - Activités de revue
 - Mesures de complexité
 - Analyse du flux de données
 - **Exécution symbolique**
 - Preuve formelle
 - Interprétation abstraite

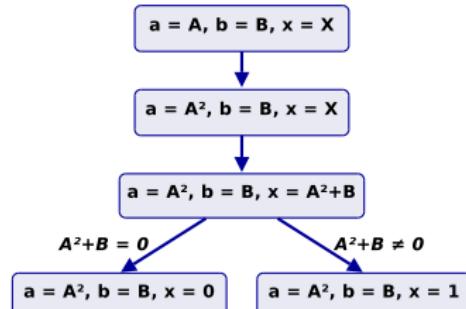
Exécution symbolique

Principe de l'exécution symbolique

Assigner des valeurs symboliques aux variables (dans un algorithme) et non pas des valeurs numériques. L'algorithme est alors exécuté selon la sémantique de chacune de ses instructions.

Exemple

```
int foo(int a, int b) {
    int x ;
    a = a * a;
    x = a + b;
    if ( x == 0 ) x = 0;
    else x = 1;
    return x;
}
```



Exécution symbolique

Exercice 4.24 ↗

```
int foo(int x) {  
    int a, b;  
  
    if ( x > 0 ) a = -x;  
    else a = x;  
  
    b = -a;  
  
    if ( -2 * b <= 0 )  
        printf(" Cas 1 ");  
    else printf(" Cas 2 ");  
}
```

Que peut on en déduire grâce au calcul de l'arbre symbolique ?

Exécution symbolique

Points forts :

- Une description plus abstraite du code.
- Permet de trouver du code mort.

Points faibles :

- Explosion combinatoire.
- Problème de la sensibilisation indécidable.
- Nécessite une interaction pour résoudre certaines incertitudes.

Bilan :

- Dédié à des algorithmes de calcul numérique de petite taille.
- Sera certainement automatisé dans les années futures.

Plan

4

Techniques de vérification & validation

- Élaboration de données de test boîte noire
- Élaboration de données de test boîte blanche
- Évaluation des données de test
- **Analyse structurelle statique**
 - Activités de revue
 - Mesures de complexité
 - Analyse du flot de données
 - Exécution symbolique
- **Preuve formelle**
- Interprétation abstraite

Preuve formelle

Objectif

Prouver qu'un programme est correct : s'assurer que, étant donné un ensemble initial d'entrées, le programme termine et donne des résultats prévus dans la spécification (on suppose celle-ci correcte).

Méthodes

- Méthode inductive : établit les propriétés de l'algorithme d'une manière récurrente ;
- Méthode logique : se base sur les propriétés mêmes des instructions pour prouver, pas à pas, que l'algorithme garde les "bonnes" propriétés (définies par le spécifieur).

Preuve formelle I

Principe

Vérification d'un programme par rapport à sa spécification.

Spécification formelle

Formellement, une spécification est une relation S entre les entrées et les sorties du système.

Elle sera exprimée sous la forme d'un modèle formel (un système de transitions (décrivant la sémantique opérationnelle) par exemple).

Pour simplifier ici nous la décrirons sous la forme d'un ensemble de couples (entrée, sortie acceptée) :

$$S = \{(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (2, 4), (3, 3), (3, 4), (3, 5), (4, 4), (4, 5), (4, 6)\}$$

Preuve formelle II

Le comportement réel d'un programme $P1$ sera symbolisé par une relation $[P1]$:

$$[P1] = \{(1, 3), (2, 3), (3, 3), (4, 6), (5, 6), (6, 6)\}$$

- Toutes les entrées prévues par S sont acceptées par $[P1]$.
- Les sorties obtenues après ces entrées sont toutes prévues par la spécification.
- $[P1]$ accepte l'entrée 6 non considérée par la spécification (peu importe donc la sortie correspondante).

$[P1]$ est totalement exact par rapport à la spécification.

Preuve formelle III

$$[P2] = \{(1, 1), (2, 2), (3, 3), (6, 6)\}$$

- $[P2]$ ne donne pas de résultat pour l'entrée 4 (algo ne termine peut-être pas).
- Les sorties obtenues après les entrées spécifiées sont toutes prévues par la spécification.
- $[P2]$ accepte l'entrée 6 non considérée par la spécification (peu importe donc la sortie correspondante).

$[P2]$ est partiellement exact par rapport à la spécification.

Preuve formelle IV

$$[P3] = \{(1, 1), (2, 1), (3, 3), (4, 3), (5, 5), (6, 5)\}$$

- Toutes les entrées prévues par S sont acceptées par $[P3]$.
- La sortie obtenue après l'entrée 2, 1, n'est pas prévue par la spécification.
- $[P3]$ accepte l'entrée 6 non considérée par la spécification (peu importe donc la sortie correspondante).

$[P3]$ n'est **pas** partiellement exact par rapport à la spécification.

Preuve formelle : difficultés

- Spécifications peuvent être si peu précises que n'importe quel algorithme pourrait les satisfaire partiellement.
- Choix du niveau auquel la vérification doit être réalisée : conception détaillée ou code source ?
Si conception tellement détaillée que code généré automatiquement, vérification faite sur conception détaillée.

Méthode inductive

Principe

- 1 Assertions locales (relations que les variables doivent satisfaire à un endroit précis) placées aux points critiques par le concepteur : soient ai les assertions et pi des blocs d'instructions,
 $a0; p1; a1; p2; a2;$
- 2 Vérifier, pour tout i , que si ai est vraie, alors $p(i + 1)$ (en s'exécutant) doit terminer et modifier les variables de sorte que $a(i + 1)$ soit vraie.

Exemple

```
{a1 : x > 0}  
{p2} x = x + 1;  
{a2 : x > 0}
```

Méthode inductive : Exemple

Exemple : calcul de la somme des entiers entre 1 et n

```

int somme (int n) {
    int i, s;

    {a0 : n >=0}
    s = 0;
    i = 0;
    {a1 : i <=n et s = 0}
    while(i < n){
        {a2 : i < n et s = i(i+1)/2}
        i = i+1;
        s = s + i;
    }
    {a3 : i = n et s = n(n+1)/2}
    return s;
}

```

$$\begin{array}{lll}
 a0 \text{ vraie} & \Rightarrow & n \geq 0 \\
 s = 0 & \Rightarrow & s = 0 \\
 i = 0 \text{ et } n \geq 0 & \Rightarrow & i \leq n \\
 s = 0 \text{ et } i \leq n & \Rightarrow & a1 \text{ vraie.}
 \end{array}$$

$a0 \Rightarrow a1$

Méthode inductive : Exemple

Exemple : calcul de la somme des entiers entre 1 et n

```

int somme (int n) {
    int i, s;

    {a0 : n >=0}
    s = 0;
    i = 0;
    {a1 : i <=n et s = 0}
    while(i < n){
        {a2 : i < n et s = i(i+1)/2}
        i = i+1;
        s = s + i;
    }
    {a3 : i = n et s = n(n+1)/2}
    return s;
}

```

<i>a1 vraie</i>	\Rightarrow	$i = n$ ou $i < n$
$i = n$ et $i = 0$	\Rightarrow	boucle non exécutée
$n = 0$ et $s = 0$	\Rightarrow	$n = 0$
$s = n(n + 1)/2$	\Rightarrow	$s = n(n + 1)/2$
et $i = n$	\Rightarrow	<i>a3 vraie.</i>
$i < n$ et $s = 0$ et $i = 0$	\Rightarrow	<i>a2 vraie.</i>

a0 \Rightarrow a1 \Rightarrow a2

Méthode inductive : Exemple

Exemple : calcul de la somme des entiers entre 1 et n

```

int somme (int n) {
    int i, s;

    {a0 : n >=0}
    s = 0;
    i = 0;
    {a1 : i <=n et s = 0}
    while(i < n) {
        {a2 : i < n et s = i(i+1)/2}
        i = i+1;
        s = s + i;
    }
    {a3 : i = n et s = n(n+1)/2}
    return s;
}

```

Après les 2 affectations, $i < n$ réexaminé.

- Si $i < n$, a2 est vérifiée (invariant de boucle) :

soit i_k la valeur de i à la k -ième itération,

$$s = \frac{i_k(i_k+1)}{2} + i = \frac{i_k(i_k+1)}{2} + i_k + 1 = \frac{(i_k+1)(i_k+2)}{2}$$

- Si maintenant $i = n$:

soit $i_s = n - 1$ la valeur de i en a2;

alors $s = i_s(i_s + 1)/2 + i$.

Donc :

$$s = \frac{(n-1)(n-1+1)}{2} + n = \frac{n(n+1)}{2}$$

Méthode inductive : Exemple

Exemple : calcul de la somme des entiers entre 1 et n

```
int somme (int n) {
    int i, s;

    {a0 : n >=0}
    s = 0;
    i = 0;
    {a1 : i <=n et s = 0}
    while(i < n){
        {a2 : i < n et s = i(i+1)/2}
        i = i+1;
        s = s + i;
    }
    {a3 : i = n et s = n(n+1)/2}
    return s;
}
```

$a0 \Rightarrow a1 \Rightarrow a2 \Rightarrow a3$

Algorithme totalement exact.

Preuve formelle - Méthode inductive

Points faibles :

- Démonstration de la terminaison de chaque bloc d'instructions nécessaire pour prouver que le programme est totalement exact (problème indécidable dans le cas général).
- Sinon, programme partiellement exact.
- Mais pas d'algorithme général pour prouver l'exactitude d'un algorithme par rapport à une quelconque spécification.
- ⇒ intervention manuelle pour réduire la non-décidabilité de la preuve.

Méthode inductive - Exercices

Exercice 4.25

```
int fact(int n){  
    int f,i;  
  
    i = n;  
    f = 1;  
  
    while(i > 1){  
  
        f = f * i;  
        i--;  
    }  
  
    return f;  
}
```

Méthode inductive - Exercices

Exercice 4.26

```
int fact2(int n){  
    int f,i;  
  
    i = 1;  
  
    while(i <= n) {  
  
        if(i == 1)  
            f = 1;  
        else  
            f = f * i;  
        i++;  
    }  
  
    return f;  
}
```

Méthode logique I

Principe

Vérifier un algorithme A (ensemble d'instructions) par rapport à la spécification (ens. de conditions/propriétés que les variables de A doivent satisfaire **avant** l'exécution du programme (**pré-conditions P**) couplées à celles qu'elles devront satisfaire **après** la terminaison du prgm, s'il termine (**post-conditions R**).

$$A \models S : \{P\} A \{R\}$$

Objectif

Prouver formellement que si les pré-conditions sont satisfaites, alors l'exécution de A assure la satisfaction de R .

Méthode logique II

Méthode

- ➊ Établir, à partir de A et de R , une pré-condition intermédiaire $P1$;
- ➋ Montrer, à l'aide de la logique mathématique, que la pré-condition donnée par le spécifieur (P) implique la précondition $P1$.

Méthode logique III

Exemple

{P0 : $(x > 1)$ et $(y < 0)$ } $z = x - y$ {R : $z > 0$ }

1 P1 à partir de A et R? P1 : $(x-y) > 0$

{P1 : $x > y$ } $z = x - y$ {R : $z > 0$ }

2 $P0 \Rightarrow P1$?

$(x > 1)$ et $(y < 0) \Rightarrow x > 1 > 0 > y \Rightarrow x > y$

Preuve formelle

Points forts :

- Test idéal car fournit la preuve formelle de l'exactitude d'un algorithme.
- Partiellement automatisable.

Points faibles :

- Preuve de l'algorithme, pas du programme.
- Utilisation manuelle lourde ⇒ utilisation d'outils formels (coûteux en espace mémoire et temps d'exécution).
- Déterminer les “bonnes” propriétés à prouver difficile.
- Si hypothèses fausses, preuve de n'importe quoi.

Bilan :

- Dédié aux tests unitaires de logiciels critiques.

Plan

4

Techniques de vérification & validation

- Élaboration de données de test boîte noire
- Élaboration de données de test boîte blanche
- Évaluation des données de test
- Analyse structurelle statique
 - Activités de revue
 - Mesures de complexité
 - Analyse du flot de données
 - Exécution symbolique
 - Preuve formelle
 - Interprétation abstraite

Interprétation abstraite I

Objectif

Analyser de manière statique un programme :

- afin de prouver certaines propriétés sur le comportement exécutif du programme ;
- de manière automatique ;
- sans réellement exécuter le programme.

Interprétation abstraite II

Applications

- Optimisations de code sophistiquées ;
- Détections d'inconsistances (accès tableaux, pointeurs) ;
- *Runtime errors* ;
- Aide à la preuve (extraction d'invariants) ;
- Génération automatique de cas de test.

Interprétation abstraite III

Interprétation abstraite

Théorie unifiant un grand nombre de techniques d'analyse statique de programmes.

- [Cousot&Cousot 76, 77, 79...]
- Formalise l'analyse approchée de programmes.
- Permet de comparer la précision de différentes analyses.
- Favorise la conception d'analyses sophistiquées.

Interprétation abstraite IV

Analyse

Dans ce contexte, l'analyse consiste en l'exécution abstraite du programme, pour laquelle des valeurs abstraites sont calculées à partir des valeurs (plus complexes) du programme.

⇒ analyse des valeurs possibles d'un prgm au cours d'une exécution quelconque ;

⇒ l'analyse statique calcule des approximations des valeurs des variables (ensembles d'états).

Interprétation abstraite V

Remarques

- Analyses de flot de données portent sur des propriétés de chemin.

Ex : variable est-elle référencée après avoir été définie ?

- Analyses par interprétation abstraite se focalisent sur des propriétés d'état.

Ex : la variable x est-elle toujours positive à tel point de contrôle ?

Interprétation abstraite : analyse des signes

Analyse des signes

On associe à chaque point de contrôle k et chaque variable x une propriété p où p peut être :

- $x < 0,$
- $x \leq 0,$
- $x > 0,$
- $x \geq 0,$
- $x = 0,$
- $x \neq 0,$
- \top (pas d'information),
- \perp (k non accessible).

Interprétation abstraite : analyse des signes

Exemple

```
x = 1 ; y = 10
{x > 0, y > 0}
while(x < y) {
    x = 2*x ; y = y - 1;
    {x > 0, y ≥ 0}
}
{x > 0, y ≥ 0}
```

Interprétation abstraite : analyse des intervalles

Analyse des intervalles

On synthétise à chaque point de contrôle k et chaque variable numérique x un intervalle de valeurs dans lequel on est sûr que x prend sa valeur.

Interprétation abstraite : analyse des intervalles

Exemple : tri par insertion

Vérification des accès au tableau de taille 100 :

```
for(int i = 0 ; i < 99 ; i++){
    i ∈ [0, 98]
    p = T[i] ; j = i+1 ;
    i ∈ [0, 98], j ∈ [1, 99]
    while(j<=100 && T[j] < p){
        i ∈ [0, 98], j ∈ [1, 99]
        T[j-1] = T[j] ; j = j+1;
        i ∈ [0, 98], j ∈ [2, 100]
    }
    i ∈ [0, 98], j ∈ [1, 100]
    T[j-1] = p ;
}
```

Automates interprétés - introduction I

- Générés automatiquement à partir de la syntaxe du code source (génération non formalisée dans ce cours).
- Plus pratiques que les graphes de flot de contrôle ici : on associe les informations aux points de contrôle ; les instructions se font sur les transitions ce qui permet de savoir si les informations se situent avant ou après une ou des instructions.

Un automate interprété est un automate fini étendu par des variables. Aux transitions sont associées des instructions qui spécifient l'évolution des valeurs des variables.

Un peu de vocabulaire...

Automates interprétés - introduction II

Espace d'états

Un **état** est un couple $(k, \vec{v} \in (\mathcal{K} \times \mathcal{V} = \mathcal{S}))$ où

- \mathcal{K} est l'ensemble des points de contrôle k ;
- \mathcal{V} est l'ensemble des variables dont la valeur est notée \vec{v} ;
- \mathcal{S} est l'ensemble des états.

Conditions

Une **condition** $c \in (\mathcal{C} = \mathcal{V} \rightarrow \mathbb{B})$ est un prédicat sur la valeur des variables.

Affectations

Une **affectation** $a \in (\mathcal{V} \rightarrow \mathcal{V} = \mathcal{A})$ modifie la valeur des variables (plusieurs variables peuvent être modifiées en parallèle).

Automates interprétés - introduction III

Automate interprété

Un **automate interprété** est un triplet $(\mathcal{K}, \mathcal{K}^{init}, \rightarrow)$ tel que :

- \mathcal{K} : ensemble des points de contrôle ;
- \mathcal{K}^{init} : ensemble des points de contrôle initiaux ;
- $\rightarrow \subseteq \mathcal{K} \times \mathcal{C} \times \mathcal{A} \times \mathcal{K}$: une relation décrivant le comportement de l'automate.

On note $k_1 \xrightarrow{c?a} k_2$ pour $(k_1, c, a, k_2) \in \rightarrow$.

Intuitivement, $k_1 \xrightarrow{c?a} k_2$ spécifie qu'en un point k_1 :

- si la valeur \vec{v}_1 satisfait la condition c
- alors le contrôle passe de k_1 à k_2 et les variables prennent la valeur : $\vec{v}_2 = a(\vec{v}_1)$.

Automates interprétés - sémantique opérationnelle

La sémantique opérationnelle d'un automate interprété est un système de transitions $(\mathcal{S}, \mathcal{S}^{init}, \rightarrow)$ où :

- $\mathcal{S} = \mathcal{K} \times \mathcal{V}$ est l'espace d'états, en général infini (un état étant un couple $(k, \vec{v} \in \mathcal{K} \times \mathcal{V})$;
- $\mathcal{S}^{init} = (\{(k, \vec{u})|k \in \mathcal{K}^{init}\} \subseteq \mathcal{S})$: est l'ensemble des états initiaux ;
- $\rightarrow \subseteq \mathcal{S} \times \mathcal{S}$: est la relation de transition entre états.

Une transition $(k_1, \vec{v}_1) \xrightarrow{c?a} (k_2, \vec{v}_2)$ signifie :

- que le contrôle passe du point k_1 au point k_2 ;
- que les variables changent de valeur, \vec{v}_1 devient \vec{v}_2 .

$$\frac{k_1 \xrightarrow{c?a} k_2, c(\vec{v}_1) = true, \vec{v}_2 = a(\vec{v}_1)}{(k_1, \vec{v}_1) \rightarrow (k_2, \vec{v}_2)}$$

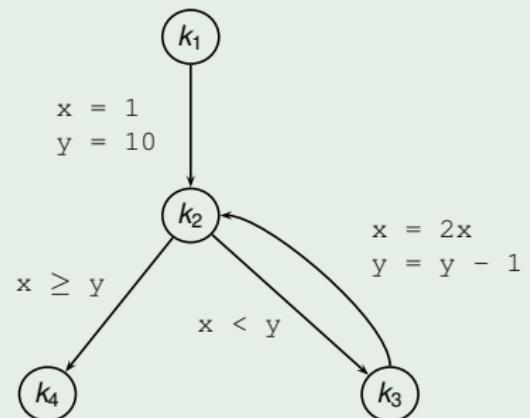
Automates interprétés - Exemple

Exemple

Programme :

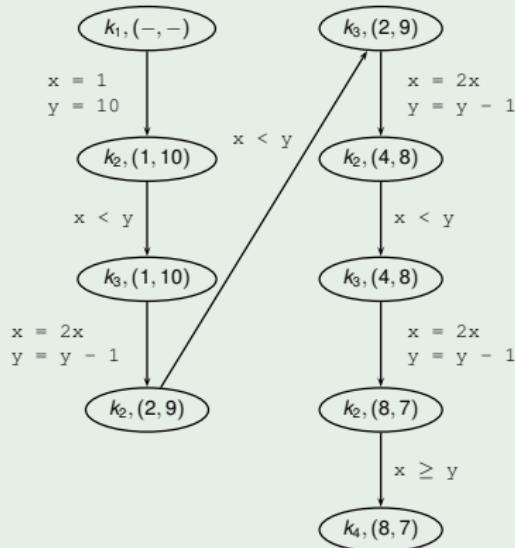
```
x = 1 ; y = 10;  
while(x < y){  
    x = 2x ;  
    y--;  
}
```

Automate interprété :



Automates interprétés - Exemple

Sémantique opérationnelle :



Analyse d'accessibilité

Objectif de l'analyse par interprétation abstraite : effectuer l'analyse d'accessibilité.

Analyse d'accessibilité

Calcule l'ensemble des états accessibles Acc à partir de l'état initial lors d'une exécution.

Formellement :

$$Acc = \{s \in \mathcal{S} \mid \exists s^{init} \in \mathcal{S}^{init} : s^{init} \xrightarrow{*} s\}$$

où $\xrightarrow{*} = \bigcup_{n \geq 0} \xrightarrow{n}$ est la fermeture transitive de $\xrightarrow{\cdot}$.

On associe à chaque point de contrôle $k \in \mathcal{K}$, l'ensemble $X_k \in \wp(\mathcal{V})$ des valeurs accessibles pour les variables au point k :

$$X_k = \{\vec{v} \in \mathcal{V} \mid (k, \vec{v}) \in X\},$$

avec $X \in \wp(\mathcal{S})$ ($\wp(\mathcal{S}) = \wp(\mathcal{K} \times \mathcal{V}) \simeq \mathcal{K} \rightarrow \wp(\mathcal{V})$).

Sémantique collectrice des automates interprétés I

La sémantique opérationnelle définit un système de transitions entre états. Ici nous manipulons des ensembles d'états.

⇒ Utilisation d'une **sémantique collectrice** qui définit les relations entre ensembles de valeurs accessibles X_k à chaque point de contrôle k.

Sémantique collectrice des automates interprétés II

Cas d'une transition :

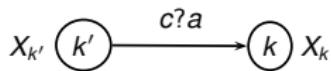
$$X_k = X_k^{init} \cup a(c?(X_{k'}))$$

avec :

$$a(X) = \{a(\vec{v}) \mid \vec{v} \in X\}$$

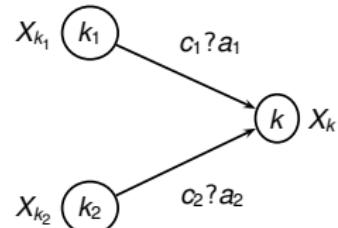
$$c?(X) = \{\vec{v} \in X \mid c?(\vec{v}) = \text{true}\}$$

$$X_k^{init} = \begin{cases} \mathcal{V} & \text{si } k \in K^{init} \\ \emptyset & \text{sinon} \end{cases}$$



Cas général :

$$X_k = X_k^{init} \cup \bigcup_{k' \xrightarrow{c?a} k} a(c?(X_{k'}))$$



avec

$$X_k = a_1(c_1?(X_{k_1})) \cup a_2(c_2?(X_{k_2}))$$

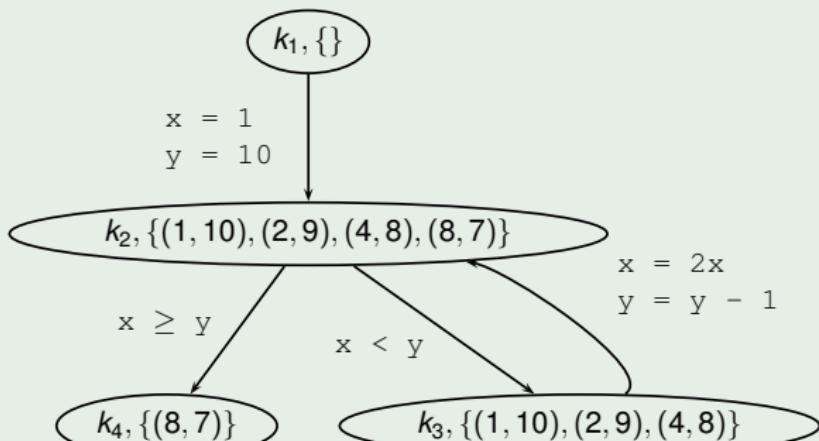
Sémantique collectrice - Exemple

Exemple

Programme :

```
x = 1 ; y = 10;
while(x < y) {
    x = 2x ;
    y--;
}
```

Sémantique collectrice :



Analyse d'accessibilité I

Les X_k sont donc reliés par un système d'équations de point fixe :

$$X_k = F_k(X_1, X_2, \dots, X_{|\mathcal{K}|}), k \in \mathcal{K}$$

ou encore $X = F(X)$.

Analyse exacte

Plus petite solution de ce système ou limite de :

$$X_k^0 = \emptyset, X_k^{n+1} = F_k(X_1^n, X_2^n, \dots, X_{|\mathcal{K}|}^n)$$

(théorème de Kleene, les F_k sont continues)

Analyse d'accessibilité II

Problème indécidable

Intuitivement :

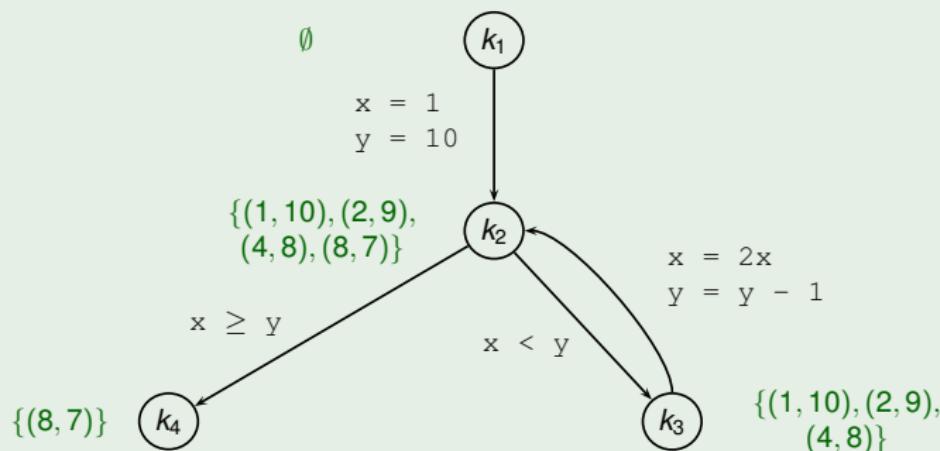
- Représentation des X_k très difficile (peuvent être infinis) ;
- La limite peut ne pas être accessible en un nombre fini d'étapes.

⇒ **analyse exacte** ne peut être effectuée que dans certains cas : types énumérés, variables entières bornées.

On parle parfois d'**exécution symbolique**.

Analyse exacte - Exemple

Simulation sur automate interprétré :



Stabilisation en 5 itérations (point fixe atteint)

Analyse d'accessibilité approchée

Objectif

Repousser les limites de la non-décidabilité du problème d'accessibilité.

Principe

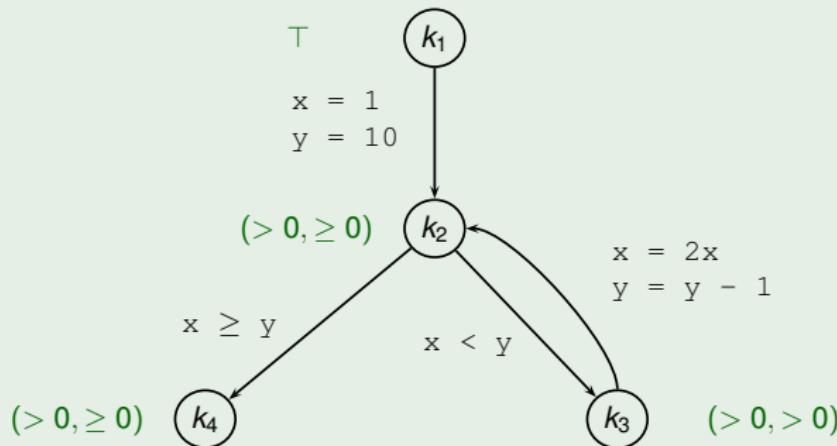
Analyse exacte basée sur treillis, analyse abstraite basée sur treillis abstrait ;
Liens entre les treillis : connexions de Galois.

Exemples d'abstraction

- Signes ;
- Intervalles ;
- Polyèdres ;
- ...

Analyse approchée - Exemple par les signes

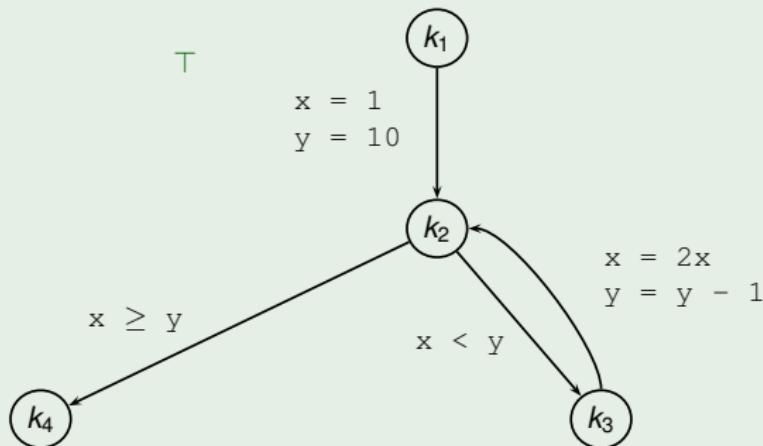
Simulation sur automate interprété :



Stabilisation en 3 itérations (point fixe atteint)

Analyse approchée - Exemple par les intervalles

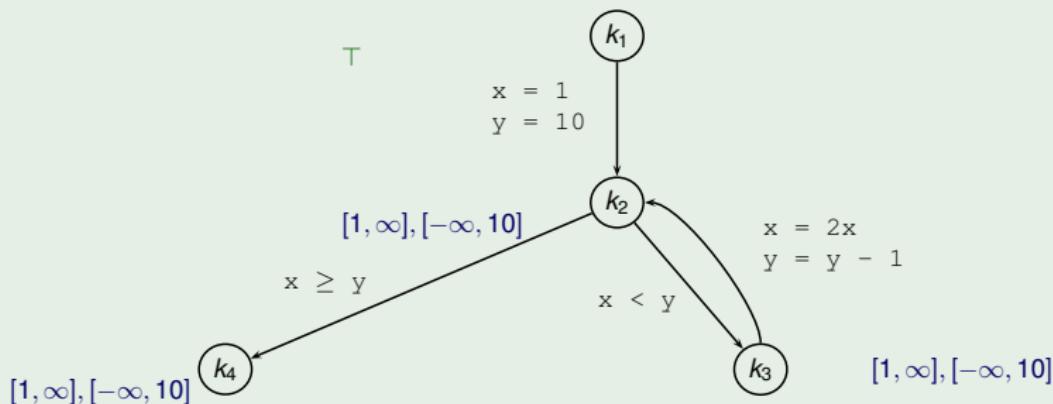
Simulation sur automate interprété :



Stabilisation en 10 itérations (point fixe atteint)

Analyse approchée - Exemple par les intervalles avec élargissement

Simulation sur automate interprété :



Élargissement

Stabilisation en 2 itérations (point fixe atteint)

Interprétation abstraite - Exercice

Exercice 4.27 ↗

Programme :

```
s = 0 ; i = 1;
while(i <= n) {
    s = s + i ;
    i++;
    if(s > c) n = i;
}
```

- ➊ Donner l'automate interprété de ce programme (et sa sémantique collectrice ?).
- ➋ Effectuer l'analyse par intervalles de ce programme pour :
 - ➌ $I(n) = [0, 20]$ et
 $I(c) = [300, 1000]$;
 - ➍ $I(n) = [0, 20]$ et
 $I(c) = [200, 1000]$.

Interprétation abstraite

Points forts :

- Puissance de calcul.
- Automatisable.

Points faibles :

- Analyse exacte : applicable que dans certains cas.
- Analyse approchée : savoir définir une approximation pertinente en précision et temps de calcul (problème d'indécidabilité).

Bilan :

- Utilisé pour les tests unitaires et tests de validation pour des systèmes critiques.
- À approfondir dans un futur proche.

Conclusion sur l'analyse structurelle statique

L'analyse structurelle statique :

- Automatisation possible par des outils (mesure de complexité, analyse du flot de données).
- Investissement en ingénieurs (lecture de code).
- Investissement en puissance de calcul (exécution symbolique, preuve formelle, interprétation abstraite).
- Principalement utilisée en test unitaire ou de validation pour du logiciel critique.

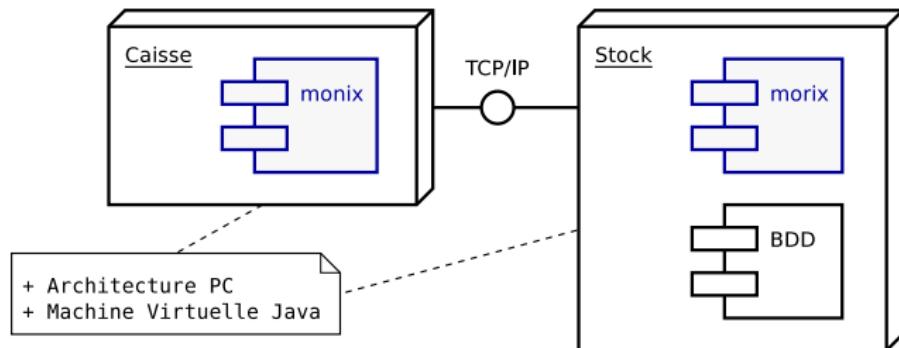
- Monix
- Références

- Monix
- Références

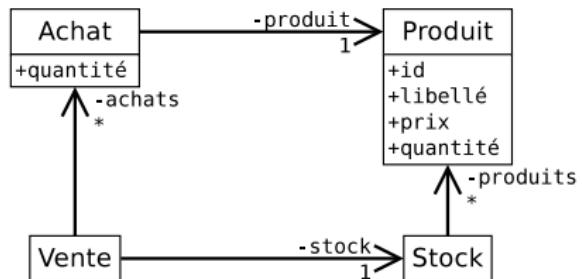
Monix : domaine et déploiement

Extrait du document de spécification (Monix_specification_v3.0.pdf)

Déploiement :



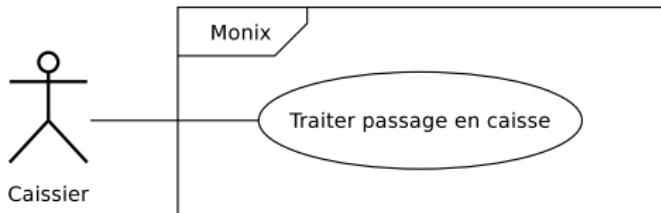
Domaine :



Monix : cas d'utilisation

Extrait du document de spécification (Monix_specification_v3.0.pdf)

Cas d'utilisation : Traiter passage en caisse.



Monix : cas d'utilisation

Extrait du document de spécification ([Monix_specification_v3.0.pdf](#))

Cas d'utilisation : Traiter passage en caisse.

Résumé : Un caissier passe des produits en caisse pour un vente. Les produits sont ajoutés ou annulés de la vente. La vente est clôturée. Le stock est mis à jour.

Portée : Le logiciel de la caisse enregistreuse (monix), le logiciel serveur de gestion du stock (morix), la base de données.

Niveau : Utilisateur.

Acteur principal : Caissier.

Pré-conditions : Caisse en état de fonctionnement. Pas de passage en cours.

Garanties en cas de succès : Tous les produits sont enregistrés dans la vente. Le stock est mis à jour.

Monix : cas d'utilisation

Extrait du document de spécification (Monix_specification_v3.0.pdf)

Cas d'utilisation : Traiter passage en caisse.

Scénario nominal :

1. Le caissier saisit l'identifiant d'un produit.
2. La caisse affiche le libellé et le prix du produit.
3. Le caissier saisit la quantité de produits.
4. Le caissier signale l'ajout du produit.
5. La caisse ajoute la quantité de produits à la vente.
6. La caisse affiche l'ajout du produit.
7. La caisse affiche le montant total de la vente.
8. Le caissier signale la clôture de la vente.
9. La caisse déclenche la mise à jour du stock.
10. La caisse affiche le stock modifié.

Monix : cas d'utilisation

Extrait du document de spécification (Monix_specification_v3.0.pdf)

Cas d'utilisation : Traiter passage en caisse.

Exception [Article non identifié] :

2.a La caisse affiche que le produit est inconnu.

Exception [Article non identifié pour un ajout] :

5.a.1 La caisse affiche que l'achat est impossible.

5.a.2 Va en 8.

Variante [Annulation d'un produit] :

4.a.1 Le caissier signale l'annulation du produit.

4.a.2 La caisse soustrait la quantité de produits.

4.a.3 La caisse affiche l'annulation du produit.

4.a.4 Va en 7.

Exception [Article non identifié pour une annulation] :

4.a.2.a La caisse affiche que l'achat est impossible.

4.a.2.b Va en 8.

Variante [Saisir plusieurs produits] :

8.a Va en 1.

Monix : monix : IHM

Extrait du document de spécification (Monix_specification_v3.0.pdf)

Interface Homme-Machine :

<identifiant produit>	<quantité>
<libellé produit>	<prix>
<input type="button" value="Ajouter"/> <input type="button" value="Enlever"/>	
<informations vente>	
<prix total>	
<input type="button" value="Fin de la vente"/>	

Vue caisse

<id>	<libellé>	<prix>	<quantité>

Vue stock

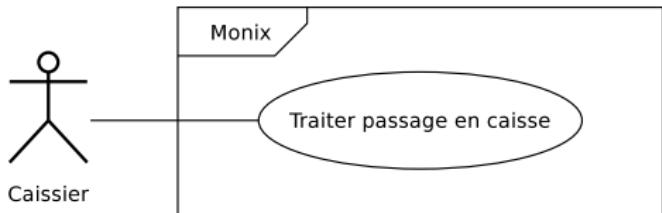
<ticket client>	
<prix total>	

Vue client

Monix : test de validation

Extrait du document de spécification (Monix_specification_v3.0.pdf)

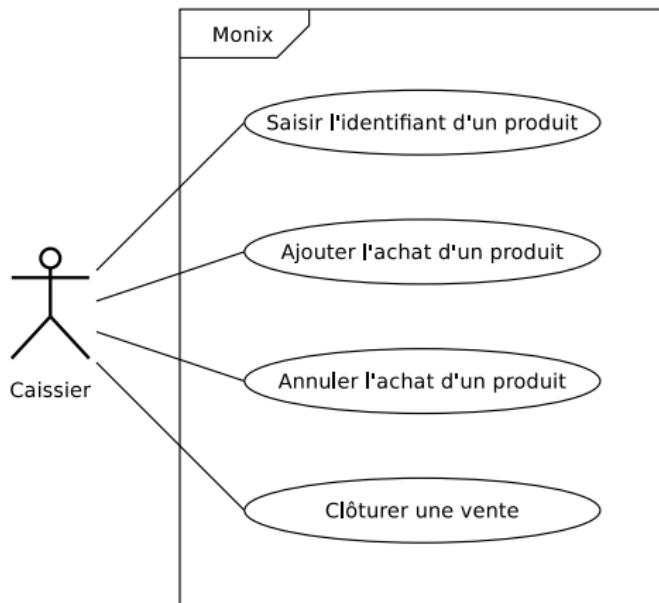
Cas d'utilisation : Traiter passage en caisse.



Monix : test de validation

Extrait du cahier de test (TV, ref : spécification) (Monix_TV_v3.0.pdf)

Scénarios Validation : C.U. Traiter passage en caisse.



Monix : test de validation

Extrait du cahier de test (TV, ref : spécification) (Monix_TV_v3.0.pdf)

Scénarios Validation : C.U. Traiter passage en caisse.

Scénario : Saisir l'identifiant d'un produit		
Action(s)	Résultat(s) interne(s)	Résultat(s) observable(s)
1. Vue caisse : Saisie \$id → <identifiant produit> Perte de focus → <identifiant produit>		2. Vue caisse : <libellé produit> ← Produit.libellé Produit.id = \$id <prix> ← Produit.prix Produit.id = \$id

Scénario : Saisir l'identifiant d'un produit [Article non identifié]		
Action(s)	Résultat(s) interne(s)	Résultat(s) observable(s)
1. Vue caisse : Saisie \$id → <identifiant produit> Perte de focus → <identifiant produit>		2.a Vue caisse : <libellé produit> ← "Produit inconnu"

Monix : test de validation

Extrait du cahier de test (TV, ref : spécification) (Monix_TV_v3.0.pdf)

Scénarios Validation : C.U. Traiter passage en caisse.

Scénario : Ajouter l'achat d'un produit

Pré-condition : Saisir l'identifiant d'un produit

Action(s)	Résultat(s) interne(s)	Résultat(s) observable(s)
3. Vue caisse : Saisie \$qtt → <quantité>		
4. Vue caisse : Clique [Ajouter]	5. Composant monix : <pre>Vente ← (+) Achat { quantite ← + \$qtt produit ← Produit Produit.id = \$id }</pre>	6. Vue caisse : <pre><informations vente> ← "+ produit \$id × \$qtt × Produit.prix" Produit.id = \$id</pre> Vue client : <pre><ticket client> ← + "produit \$id × \$qtt × Produit.prix" Produit.id = \$id</pre> 7. Vue caisse : <pre><prix total> ← + \$qtt * Produit.prix Produit.id = \$id</pre> Vue client : <pre><prix total> ← + \$qtt * Produit.prix Produit.id = \$id</pre>

Monix : test de validation

Extrait du cahier de test (TV, ref : spécification) (Monix_TV_v3.0.pdf)

Scénarios Validation : C.U. Traiter passage en caisse.

Scénario : Ajouter l'achat d'un produit [Article non identifié pour un ajout]

Pré-condition : Saisir l'identifiant d'un produit [Article non identifié]

Action(s)	Résultat(s) interne(s)	Résultat(s) observable(s)
3. Vue caisse : Saisie \$qtt → <quantité>		
4. Vue caisse : Clique [Ajouter]		5.a.1 Vue caisse : <informations vente> ← "Achat impossible"

Monix : test de validation

Extrait du cahier de test (TV, ref : spécification) (Monix_TV_v3.0.pdf)

Scénarios Validation : C.U. Traiter passage en caisse.

Scénario : Annuler l'achat d'un produit

Pré-condition : Saisir l'identifiant d'un produit

Action(s)	Résultat(s) interne(s)	Résultat(s) observable(s)
3. Vue caisse : Saisie \$qtt → <quantité>		
4.a.1 Vue caisse : Clique [Enlever]	4.a.2 Composant monix : <pre>Vente ← (+) Achat { quantite ←- \$qtt produit ← Produit Produit.id = \$id }</pre>	4.a.3 Vue caisse : <pre><informations vente> ← "- produit \$id × \$qtt × Produit.prix" Produit.id = \$id</pre> Vue client : <pre><ticket client> ←+ "produit \$id × -\$qtt × Produit.prix" Produit.id = \$id</pre> 4.a.4 Vue caisse : <pre><prix total> ←- \$qtt * Produit.prix Produit.id = \$id</pre> Vue client : <pre><prix total> ←- \$qtt * Produit.prix Produit.id = \$id</pre>

Monix : test de validation

Extrait du cahier de test (TV, ref : spécification) (Monix_TV_v3.0.pdf)

Scénarios Validation : C.U. Traiter passage en caisse.

Scénario : Annuler l'achat d'un produit [Article non identifié pour une annulation]		
Pré-condition : Saisir l'identifiant d'un produit [Article non identifié]	Action(s)	Résultat(s) interne(s)
Résultat(s) observable(s)		
3. Vue caisse : Saisie \$qtt → <quantité>		
4.a.1 Vue caisse : Clique [Enlever]		4.a.2.a Vue caisse : <informations vente> ← "Achat impossible"

Monix : test de validation

Extrait du cahier de test (TV, ref : spécification) (Monix_TV_v3.0.pdf)

Scénarios Validation : C.U. Traiter passage en caisse.

Scénario : Clôturer une vente

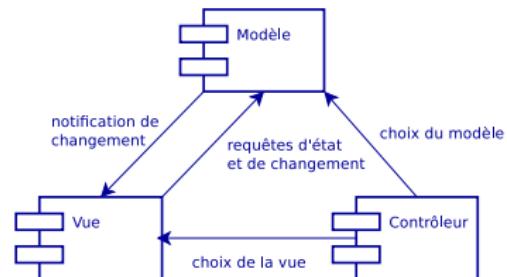
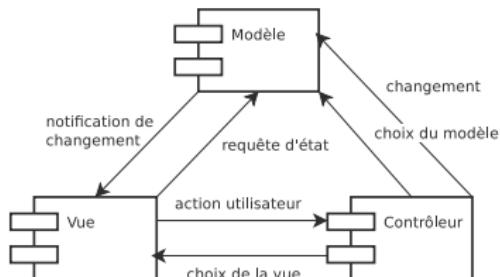
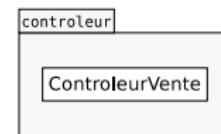
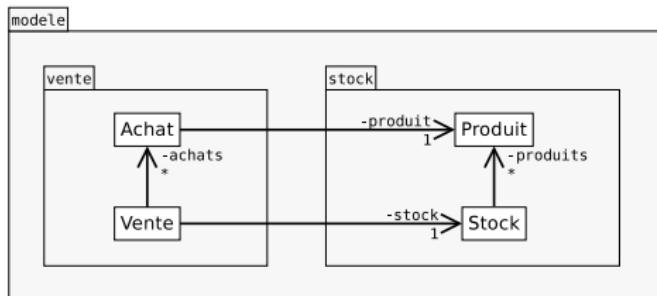
Pré-condition : Ajouter l'achat d'un produit | Annuler l'achat d'un produit

Action(s)	Résultat(s) interne(s)	Résultat(s) observable(s)
8. Vue caisse : Clique [Fin de la vente]	9. Composant monix : requête à morix ← $\forall (\$id, \$qtt)$ modifier $\pm \$qtt$ Produit Produit.id = \$id Composant morix : requête à BDD ← $\forall (\$id, \$qtt)$ UPDATE Produit SET <i>quantité</i> $\pm= \$qtt$ WHERE Produit.id = \$id	10. Vue stock : $\forall (\$id, \$qtt)$ <i><quantité></i> $\pm= \$qtt$ <i><id></i> = \$id

Monix : monix : conception de l'architecture

Extrait du document de conception (Monix_conception_v3.0.pdf)

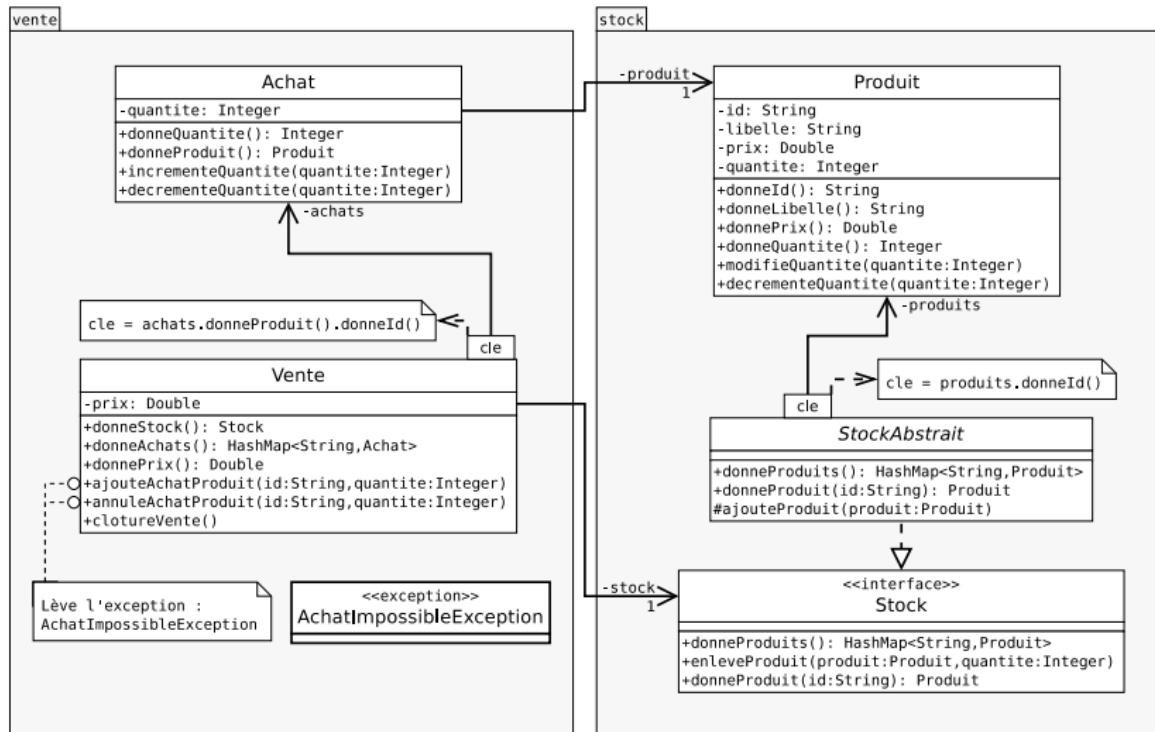
Achitecture :



Monix : monix : conception détaillée du modèle

Extrait du document de conception (Monix_conception_v3.0.pdf)

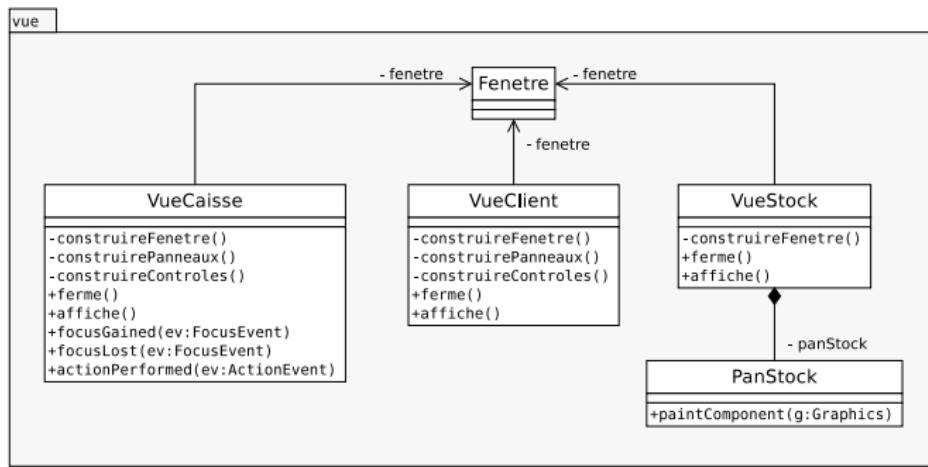
Modèle (avec stock abstrait) :



Monix : monix : conception détaillée des vues

Extrait du document de conception (Monix_conception_v3.0.pdf)

Vue (éléments graphiques) :

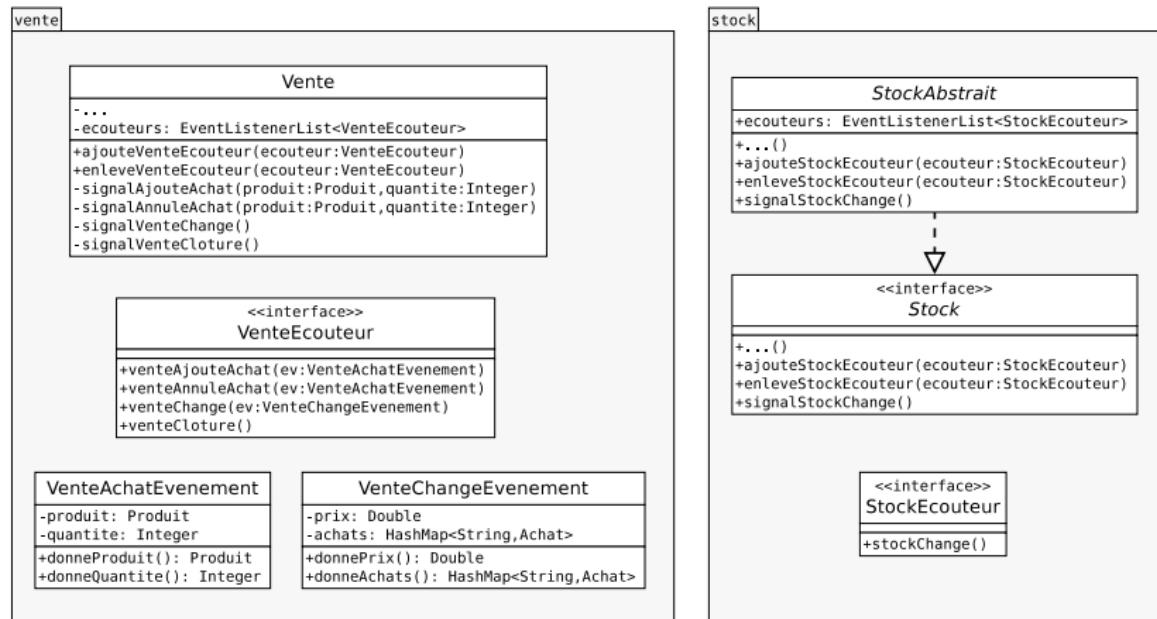


→ Java AWT et Java Swing.

Monix : monix : conception détaillée du modèle (MVC)

Extrait du document de conception (Monix_conception_v3.0.pdf)

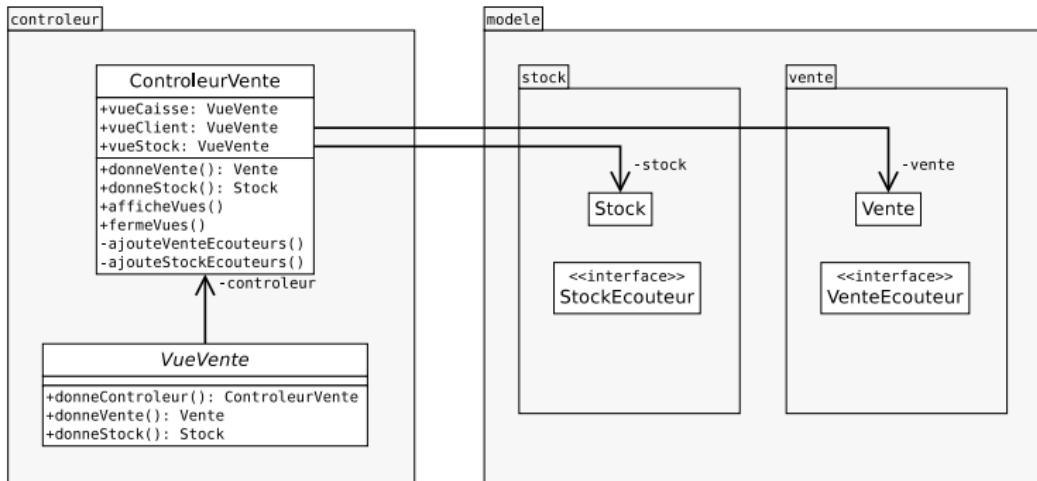
Modèle (architecture MVC) :



Monix : monix : conception détaillée du contrôleur

Extrait du document de conception (Monix_conception_v3.0.pdf)

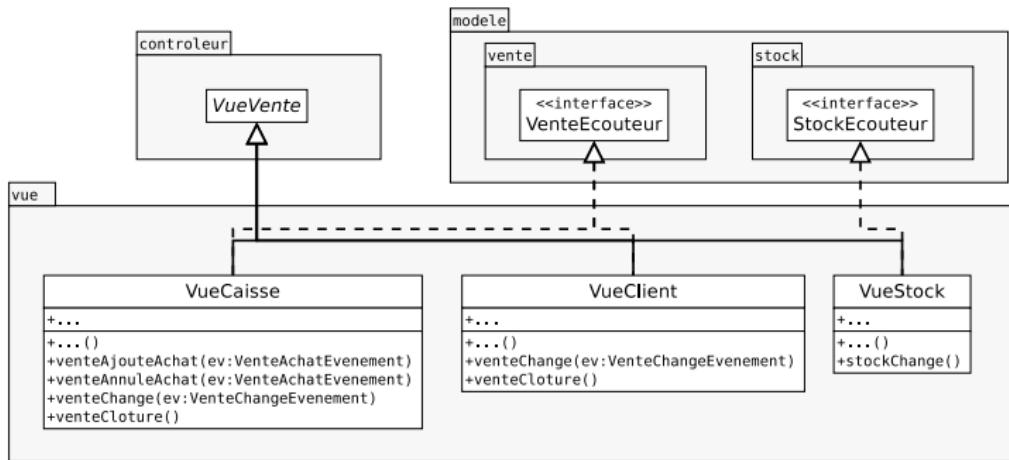
Contrôleur (architecture MVC) :



Monix : monix : conception détaillée des vues (MVC)

Extrait du document de conception (Monix_conception_v3.0.pdf)

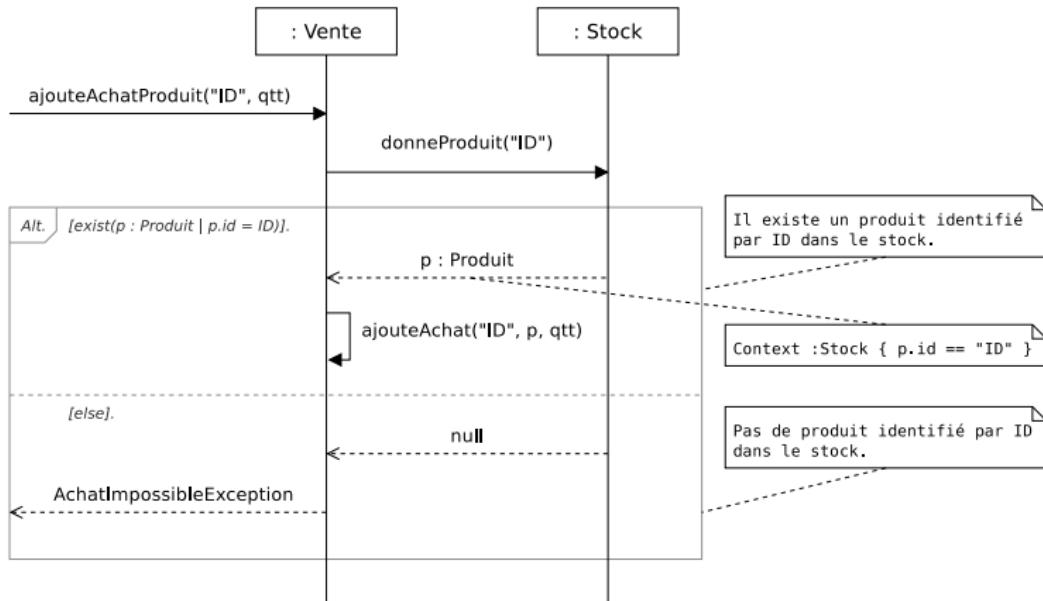
Vue (architecture MVC) :



Monix : monix : conception détaillée du modèle

Extrait du document de conception (Monix_conception_v3.0.pdf)

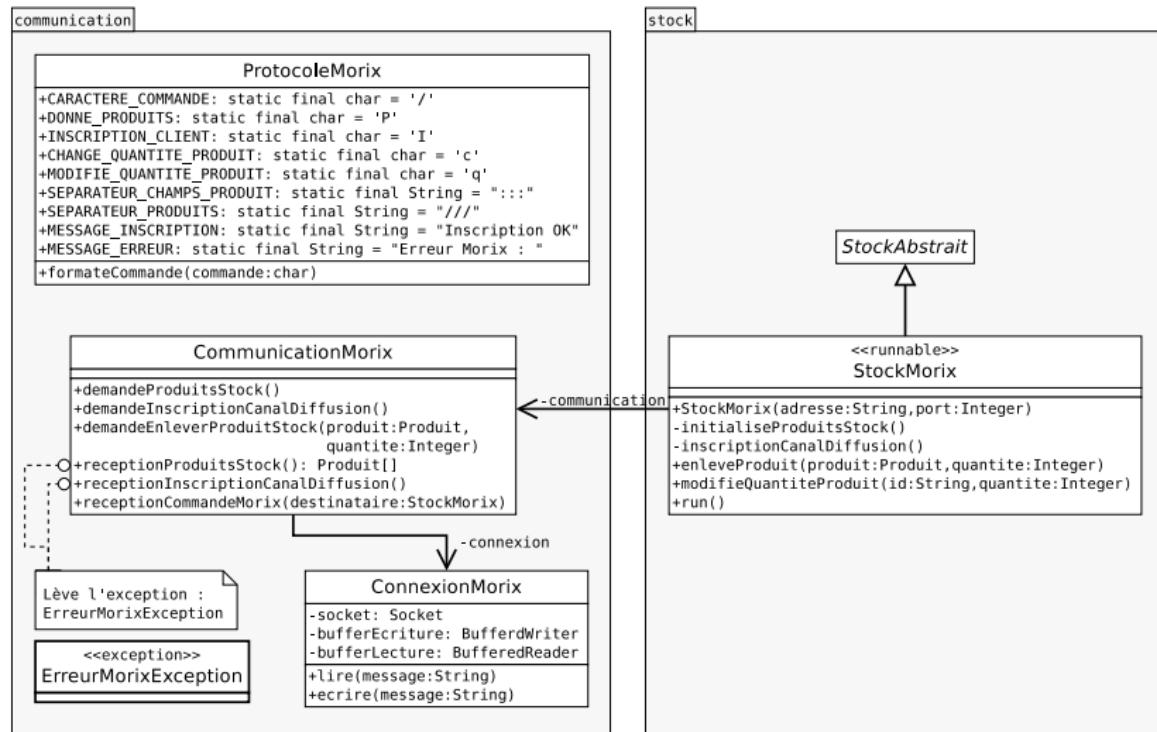
Modèle (séquence) : Vente : ajouteAchatProduit (String, Integer)



Monix : monix : conception détaillée avec morix

Extrait du document de conception (Monix_conception_v3.0.pdf)

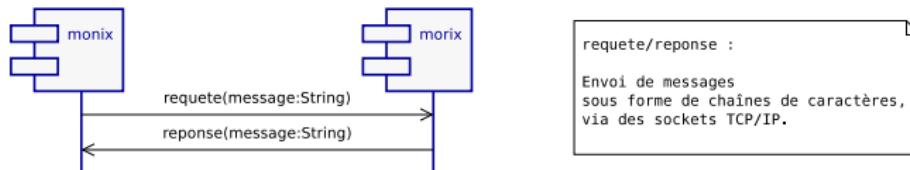
Communication (avec morix, gestion d'un stock Morix) :



Monix : monix : conception détaillée avec morix

Extrait du document de conception (Monix_conception_v3.0.pdf)

Communication (avec morix, gestion d'un stock Morix) :



Protocole de construction des messages :

Nom	Valeur	Signification	Utilisation
DONNE_PRODUITS	P	Demande de tous les produits.	/P
INSCRIPTION_CLIENT	I	Inscription au canal de diffusion.	/I
CHANGE_QUANTITE_PRODUIT	c	Change (+/-) la quantité d'un produit.	/c id :: qtt (*)
MODIFIE_QUANTITE_PRODUIT	q	Modifie la quantité d'un produit.	/q id :: qtt (*)

(*) *id* est l'identifiant du produit, *qtt* est la quantité.

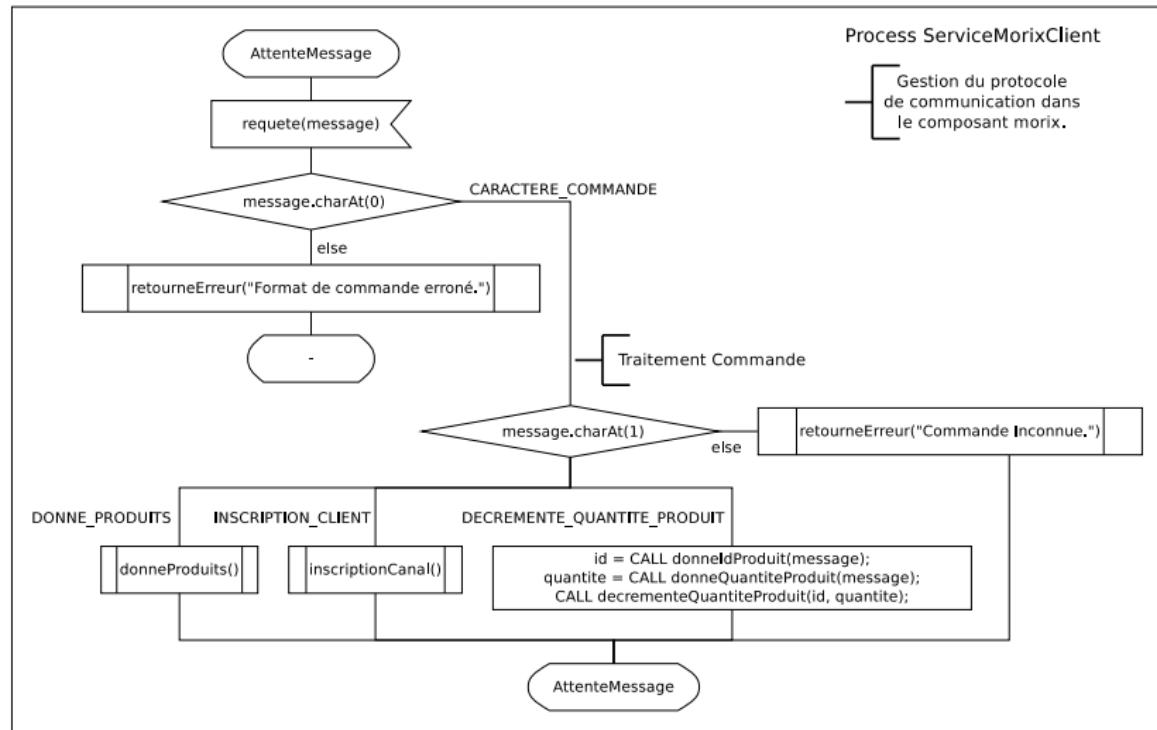
CARACTERE_COMMANDE	/	Débute une commande.
SEPARATEUR_CHAMPS_PRODUIT	:::	Séparateur entre les champs d'un produit.
SEPARATEUR_PRODUIT	///	Séparateur entre les produits.

MESSAGE_INSCRIPTION	Inscription OK	Message de retour d'une inscription au canal de diffusion.
MESSAGE_ERREUR	Erreur Morix :	Prefixe de tout message d'erreur retourné par Morix.

Monix : monix : conception détaillée avec morix

Extrait du document de conception (Monix_conception_v3.0.pdf)

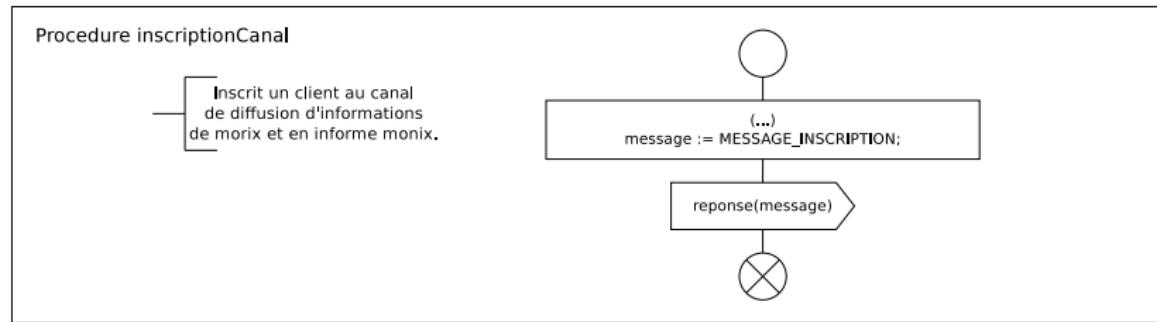
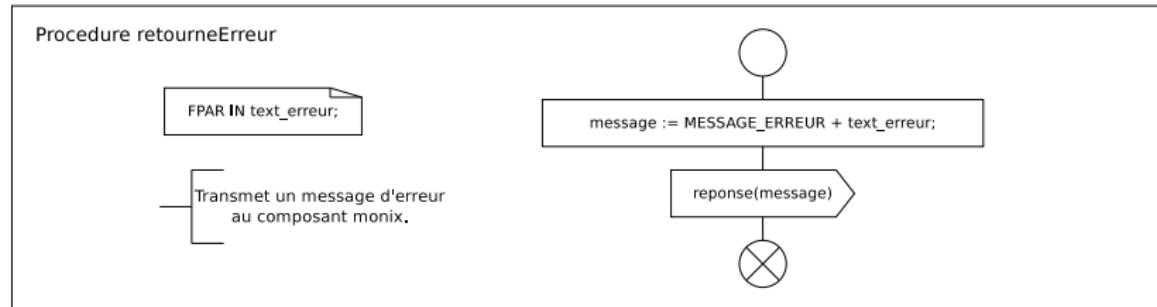
Communication (avec morix, gestion d'un stock Morix) :



Monix : monix : conception détaillée avec morix

Extrait du document de conception (Monix_conception_v3.0.pdf)

Communication (avec morix, gestion d'un stock Morix) :



- Monix
- Références

Références : livres (fr)

-  [Le test des logiciels,](#)
Éditions Hermès, ISBN 2-7462-0083-X,
S. XANTHAKIS, P. RÉGNIER et C. KARAPOULIOS
-  [Pratique des tests logiciels,](#)
Dunod, 2009, ISBN 978-2-10-051862-3,
J.F. PRADAT-PYRE et J. PRINTZ
-  [Industrialiser le test fonctionnel,](#)
Dunod, 2009, ISBN 978-2-10-051533-2,
B. LEGEARD, F. BOUQUET et N. PICKAERT
-  [Test logiciel en pratique,](#)
Vuibert Informatique, 2002, ISBN 978-2711748068,
J. WATKINS
-  [Gestion des test logiciels,](#)
Editions ENI, 2011, ISBN 978-2746070035,
E. ITIÉ
-  [Les tests logiciels fondamentaux,](#)
Hermes Science Publications, 2011, ISBN 978-2746231559,
B. HOMÈS

Références : livres (en)

-  [The Art of Software Testing](#),
3rd Ed., John Wiley & Sons, 2012, ISBN 978-1-118-03196-4,
G.J. MYERS
-  [Software Testing : a Craftsman's Approach](#),
3nd Ed., CRC Press, 2008, ISBN 978-0849374753,
P. JORGENSEN
-  [Systematic Software Testing](#),
Artech House, 2002, ISBN 978-1580535083,
R.D. CRAIG and S.P. JASKIEL
-  [A Practitioner's Guide to Software Test Design](#),
Artech House, 2004, ISBN 978-1580537919,
L. COPELAND
-  [Software Engineering](#),
6th Ed., Addison-Wesley, 2001,
I. SOMMERVILLE

Références : articles

-  [Abstract Interpretation Based Program Testing](#)
In Proceedings of the SSGRR 2000 Computer & eBusiness International Conference,
P. COUSOT and R. COUSOT
-  [Static Analysis by Abstract Interpretation of Embedded Critical Software](#)
In ACM SIGSOFT Software Engineering Notes, 2011,
J. BERTRANE et all.
-  [Model-Based Testing of Reactive Systems](#)
Lecture Notes in Computer Science. Springer Verlag, 2005, vol. 3472,
M. BROY et all.

Références : internet

-  [Comité Français des Tests Logiciels,](http://www.cftl.net)
<http://www.cftl.net>
-  [TestISSIMO,](http://www.testissimo.com)
<http://www.testissimo.com>
-  [Open source software testing tools, news and discussion,](http://www.opensourcetesting.org)
<http://www.opensourcetesting.org>
-  [JUnit,](http://www.junit.org)
<http://www.junit.org>
<http://junit.org/junit4/javadoc/latest/>
<https://github.com/junit-team/junit>
-  [EasyMock,](http://easymock.org)
<http://easymock.org>
-  [Mockito,](http://site.mockito.org)
<http://site.mockito.org>
-  [JMeter,](http://jmeter.apache.org/)
<http://jmeter.apache.org/>

Références : standards

- BS 7925-2 (1998) *Software Component Testing.*
- IEEE 610.12 (1990) *Standard Glossary of Software Engineering Terminology.*
- IEEE 829 (2008) *Standard for Software Test Documentation.*
- IEEE 1008 (1993) *Standard for Software Unit Testing.*
- IEEE 1012 (1986) *Standard for Verification and Validation Plans.*
- IEEE 1028 (1997) *Standard for Software Reviews and Audits.*
- IEEE 1044 (1993) *Standard Classification for Software Anomalies.*
- IEEE 1219 (1998) *Software Maintenance.*
- ISO/IEC 2382-1 (1993) *Data processing, Voc. Part 1 : Fundamental terms.*
- ISO 9000 (2000) *Quality Management Systems - Fundamentals and Voc.*
- ISO/IEC 9126-1 (2001) *Software Engineering - Software Product Quality - Part 1 : Quality characteristics and sub-characteristics.*
- ISO/IEC 12207 (1995) *I.T. - Software Life Cycle Processes.*
- ISO/IEC 14598-1 (1996) *I.T. - Software Product Evaluation.*

Références

A quality assurance engineer walks into a bar.
He runs into a bar.

- Orders a beer.
- Orders ten beers.
- Orders 2,147,483,648 beers.
- Orders -1 beers.
- Orders a nothing.
- Orders a lizard.
- Orders a sdfdsd.
- Orders a bar.
- Orders a ”; drop table orders ;”.
- Tries to leave without paying.