Le Test dans la validation du logiciel

Bruno Mermet
2017

Exemples de "bugs" marquants (1)

http://www.mines.inpl-nancy.fr/~tisseran/cours/qualite-logiciel/qualite_logiciel.html

- La gestion comptable de l'Etat en panne...

  Chorus, le pharaonique logiciel de refonte de la gestion comptable de l'Etat sur les principes de la LOLF, basé sur SAP, n'est toujours pas opérationnel.

  Coût total 2006-2015 : 1,1 milliard d'euros.

- Une bête panne de site web...

  L'accès aux réservations et achats de billets de train sur le site internet voyages-sncf.com a été fermé plusieurs heures le 20 novembre 2008, à la suite de la mise en ligne d'"une nouvelle version du site". Le site avait déjà connu une longue panne fin juillet qui avait rendu les achats, réservations et impressions de billets impossibles pendant plus de trente heures et la SNCF avait promis d'en "tirer les enseignements". Voyages-sncf.com, premier site de commerce électronique en France, reçoit 700.000 visites par jour et 38 millions de billets y ont été vendus en 2007, soit le quart des tickets SNCF.

- Un bug informatique fait 9 morts et 14 blessés

  (octobre 2007, armée sud africaine) : un canon anti aérien Oerlikon, quatre cracheurs de balles de 35 millimètres, s'est retourné, tout en tirant, au hasard. Un bug informatique serait responsable.

Exemples de "bugs" marquants (2)

http://www.mines.inpl-nancy.fr/~tisseran/cours/qualite-logiciel/qualite_logiciel.html

- 12/10/2006 : irradiations : 5 morts et 721 complications

  La "mauvaise utilisation" d'un logiciel à l'origine d'accidents de radiothérapie à Epinal - Entre mai 2004 et août 2005, des patients traités aux rayons pour des cancers de la prostate ont subi des surdosages dus à des erreurs de paramétrage d'un logiciel. Conséquences actuelles : 5 décès et des complications chez 721 patients... Cause : "erreur humaine". Cause réelle : "mauvaise ergonomie d'un logiciel obsolète".

- 29/12/2005 Cotisation retraite

  Quelque 113 personnes ont eu la désagréable surprise de recevoir pour Noël un avis d'échéance de leurs cotisations retraite pour l'année 2006 de... deux milliards d'euros, un montant erroné dû à une erreur informatique.

- 17/11/2005 Affaire du Rootkit Sony : sur-accident logiciel

  Une infection nommée Backdoor.Win32.Breplibot.b tire parti du rootkit Sony DRM, système de protection contre les copies illicites Sony DRM, en fait un code empoisonnant le système d'exploitation des clients Sony dans le but innocent de prendre le contrôle des processus lancés.

- 01/11/2005 Bourse de Tokyo

  Gros bug à la bourse de Tokyo, la plus importante d'asie, et ce sont toutes les cotations qui sont bloquées toute la journée.

- 29/09/2005 Propos racistes dans des dictionnaires

  L'ensemble des dictionnaires Littré 2005 sont retirés des librairies après une coquille raciste dans ses colonnes : un "bug informatique" (un copier-coller mal maîtrisé) a conduit à la publication d'articles racistes tirés de l'édition de 1874 ! "Juif" était défini comme "Etre riche comme un juif, cherche à gagner de l'argent avec âpreté", nègre comme "La race des nègres".

Exemples de "bugs" marquants (3)

http://www.mines.inpl-nancy.fr/~tisseran/cours/qualite-logiciel/qualite_logiciel.html

- "C'est la faute de l'informatique". Arrêt de la distribution par écrit de leur évaluation aux élèves lors de la dernière séance de chaque cours dans une grande école

  Cause évoquée : mise en place d'un nouveau logiciel de gestion.

- Convocation de centenaires à l'école. Convocation à l'école primaire de personnes âgées de 106 ans.

  Cause : codage sur deux caractères.

- Mission Vénus : passage à 5 000 000 de Km de la planète, au lieu de 5 000 Km prévus.

  Cause : remplacement d'une virgule par un point (au format US des nombres).

- Mariner 1 : la première sonde spatiale du programme Mariner, envoyée par la NASA le 27 juillet 1962. La sonde fut détruite peu de temps après son envol. Coût : 80 millions de dollars.

  Cause : un trait d'union oublié dans un programme Fortran (« plus coûteux trait d'union de l'histoire », Arthur C. Clarke).

- Passage de la ligne. Au passage de l'équateur un F16 se retrouve sur le dos.

  Cause : changement de signe de la latitude mal pris en compte.

- Socrate. Les plantages fréquents du système de réservation de places Socrate de la SNCF, sa mauvaise ergonomie, le manque de formation préalable du personnel, ont amené un report important et durable de la clientèle vers d'autres moyens de transport.

  Cause : rachat par la SNCF d'un système de réservation de places d'une compagnie aérienne, sans réadaptation totale au cahier des charges du transport ferroviaire.

Exemples de "bugs" marquants (4)

http://www.mines.inpl-nancy.fr/~tisseran/cours/qualite-logiciel/qualite_logiciel.html

- Terminaux de paiement. Le 22 décembre 2001 les 750 000 terminaux de payement chez les commerçants ne répondaient plus, ce qui entraîné de longues files d'attente en cette période d'achats de Noël.
  - Cause : saturation des serveurs de la société Atos chargés des autorisation de paiements dépassant 600F. Les autorisation de débit prennent habituellement quelques dizaines de secondes, l'attente a frôlé la demi-heure.
  - Conséquence : des clients abandonnent leurs chariots pleins. Le groupe Leclerc a chiffré son préjudice à 2 millions d'euros.
- Echec du premier lancement d'Ariane V. Au premier lancement de la fusée Ariane V, celle ci a explosé en vol.

  La cause : logiciel de plate forme inertielle repris tel quel d'Ariane IV sans nouvelle validation. Ariane V ayant des moteurs plus puissants s'incline plus rapidement que Ariane IV, pour récupérer l'accélération dûe à la rotation de la Terre. Les capteurs ont bien détecté cette inclinaison d'Ariane V, mais le logiciel l'a jugée non conforme au plan de tir (d'Ariane IV), et a provoqué l'ordre d'auto destruction. En fait tout se passait bien... Coût du programme d'étude d'Ariane V : 38 milliards de Francs, pour 39 secondes de vol après 10 années de travail....

Hiérarchisation des tests

- Tests de recette (anciennement, tests fonctionnels)

  Tests visant à tester des fonctionnalités dans leur ensemble

- Tests d'intégration

  Tests servant à vérifier que des modules développés indépendamment fonctionnent bien une fois mis ensemble

- Tests unitaires

  Tests permettant de vérifier directement le fonctionnement de méthodes et de classes

Exemple du Triangle
Cahier des charges

Écrire en java un programme qui demande à l'utilisateur de rentrer 3 données correspondant aux longueurs des côtés d'un triangle. Le programme doit alors préciser si le triangle est équilatéral, isocèle ou scalène.

Exemple du Triangle
Génération des cas de test

Description	c1	c2	c3	Résultat Attendu

Exemple du Triangle Tests

Exemple du Triangle
Réécriture pour faciliter les tests

- Factoriser la saisie des nombres
- Ordonner les entiers
- Utiliser les exceptions

Psychologie du test

But du test

- Mauvaises idées
  - Etablir l'absence d'erreur
  - Montrer que le programme fait correctement ce qui lui est demandé
- Meilleures idées
  - Essayer de trouver des erreurs
  - Un test "à succès" est un test qui révèle quelque chose (c.f. Métaphore examen médical)

Conséquences psychologiques

- Montrer que le programme fait correctement ce qui lui est demandé
  - Impossible, donc non motivant
  - Amène à ne tester que les cas "valides"
- Test souvent vu comme une activité destructrice, moins motivante qu'une activité constructrice
- Mais test doit être vu comme une activité constructrice de la validité d'un programme

Aspects économiques du test

Exhaustivité du test

- Impossible car
  - Nombre d'entrées valides démesurément grand, voire infini
  - Nombre d'entrées possibles non valides essentiellement infini
- Choix d'une stratégie
  - Test "boîte noire"
  - Test "boîte blanche"

Tests "boîte noire"

- Principe
  - Tests "dirigés par les données"
  - Aucune connaissance du code
- Conséquence
  - Test exhaustif impossible
  - Choix de certaines valeurs supposées représentatives et du coup, risque de passer outre des "cas spéciaux" dans les traitements
  - Problème encore plus compliqué si le passé influence le comportement

Tests "boîte blanche"

- Principe
  - On connaît la structure interne du programme
  - Tenter un test exhaustif des chemins dans le code
- Conséquence
  - Test exhaustif impossible (pb des boucles)
  - On ne teste pas les cas oubliés par le développeur
  - Certains cas choisis pour les branches ne sont pas représentatifs de toutes les exécutions possibles

```
if (a-b < epsilon) {print("convergence réussie");}
```

Principes du test de logiciel
The Art of Software Testing
Glenford J. Myers

1. Une partie nécessaire d'un cas de test est la définition du résultat attendu
2. Un programmeur devrait éviter de tester son propre code
3. Une entreprise de développement de logiciels devrait éviter de tester ses propres programmes
4. Vérifier minutieusement les résultats de chaque test
5. Les cas de tests doivent être écrits pour des conditions d'entrées invalides ou inattendues, aussi bien que pour celles qui sont valides et attendues
6. Examiner un programme pour voir s'il ne fait pas ce qui devrait ne constitue que la moitié de la bataille ; l'autre moitié consiste à voir si le programme fait ce qu'il n'est pas supposé faire.
7. Eviter de jeter les cas de tests, sauf si le programme est lui-même jeté.
8. Ne pas planifier des tests en faisant implicitement l'hypothèse qu'aucune erreur ne sera trouvée
9. Le risque de trouver une erreur dans une partie du code est proportionnel au nombre d'erreurs déjà trouvées dans cette partie.
10. Tester est une tâche très créative et proposant d'intéressants défis intellectuels

1. Une partie nécessaire d'un cas de test est la définition du résultat attendu

- L'humain voit plus ce qu'il voudrait voir que ce qu'il voit
- => Etre précis dans les descriptions
- Un cas de test = description précise
    - Des données d'entrée
    - Du résultat attendu

2. Un programmeur devrait éviter de tester son propre code

- Analogie

  Quand on se relie, on ne voit pas ses erreurs (syntaxe, grammaire) même évidentes. De même, on comprend forcément ce qu'on a voulu dire

- Test = activité destructrice

  On n'aime pas détruire ce qu'on a construit (analogie du papier peint)

3. Une entreprise de développement de logiciels devrait éviter de tester ses propres programmes

- Une entreprise doit
  - Économiser du temps
  - Économiser de l'argent
- Or le test
  - Prend du temps
  - Est coûteux
- => L'entreprise n'est pas neutre dans le processus de test

4. Vérifier minutieusement les résultats de chaque test

On "loupe" des erreurs en regardant trop vite le résultats des tests

5. Les cas de tests doivent être écrits pour des conditions d'entrées invalides ou inattendues, aussi bien que pour celles qui sont valides et attendues

- On oublie de tester les cas "invalides"
- La plupart des erreurs détectées après la sortie officielle du logiciel sont liées à des cas d'utilisation non prévus au départ
- Les tests sur entrées invalides ont beaucoup plus de chance de détecter des erreurs

6. Examiner un programme pour voir s'il ne fait pas ce qui devrait ne constitue que la moitié de la bataille ; l'autre moitié consiste à voir si le programme fait ce qu'il n'est pas supposé faire.

- Corrolaire du point précédant

7. Eviter de jeter les cas de tests, sauf si le programme est lui-même jeté.

- Concevoir un cas de test prend du temps
  - Noter tous les tests que l'on fait/envisage
- Garder les tests pour les prochaines versions et éviter la "régression"

8. Ne pas planifier des tests en faisant implicitement l'hypothèse qu'aucune erreur ne sera trouvée

- c.f. Ce qui a été dit précédemment

9. Le risque de trouver une erreur dans une partie du code est proportionnel au nombre d'erreurs déjà trouvées dans cette partie.

- Les erreurs sont groupées en paquets
- Explication possible :
  - Les erreurs sont groupées dans les parties les plus compliquées
  - Les parties simples sont exemptes d'erreurs

10. Tester est une tâche très créative et proposant d'intéressants défis intellectuels

- Trouver des bons cas de test est certainement plus créatif qu'écrire un programme

Les Tests « boîte blanche »

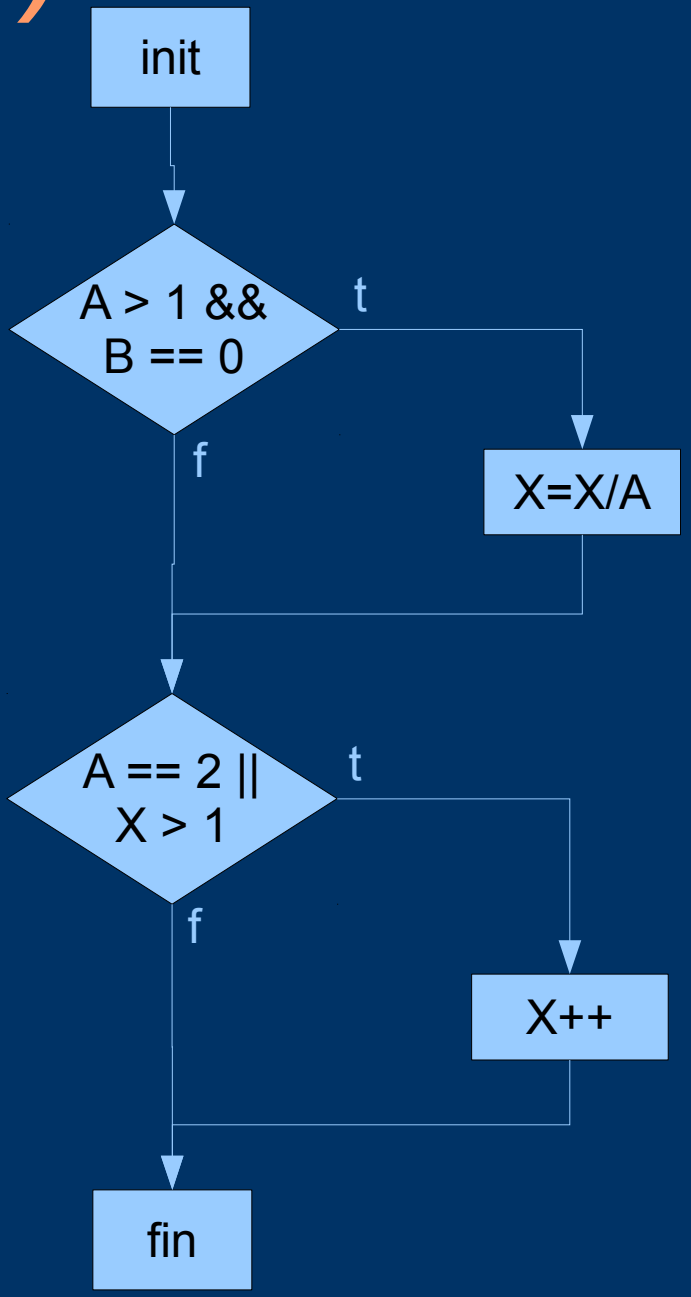
Tests BB = Tests de couverture logique

- Test exhaustif des chemins : Impossible
- Couverture des instructions
  - Non pertinent (on ne teste pas par exemple les cas de non-exécution comme un « if » simple faux sans « else »)
- Couverture des décisions
  - Pour chaque « test », avoir au moins un cas pour le choix « vrai » et un pour le choix « faux »
  - Pb : ne teste pas toutes les successions d'exécutions d'instructions pertinentes
- Couverture des conditions
  - On prend toutes les conditions intervenant dans les tests
  - On trouve un ensemble minimal de cas de tests permettant à chacune des conditions de prendre chacune de ses valeurs possibles
  - N'implique pas forcément la couverture des décisions (if a && b par exemple)
- Couverture décisions-conditions
  - On fait un « ET » des cas à tester par les 2 types précédents
  - Mais certaines conditions risquent d'en masquer d'autre (un faux dans un ET par exemple masque la valeur des autres critères)
- Couverture des conditions multiples
  - Pour chaque condition de chaque décision, envisager chaque valeur possible

Exemple applicatif (TAoST)

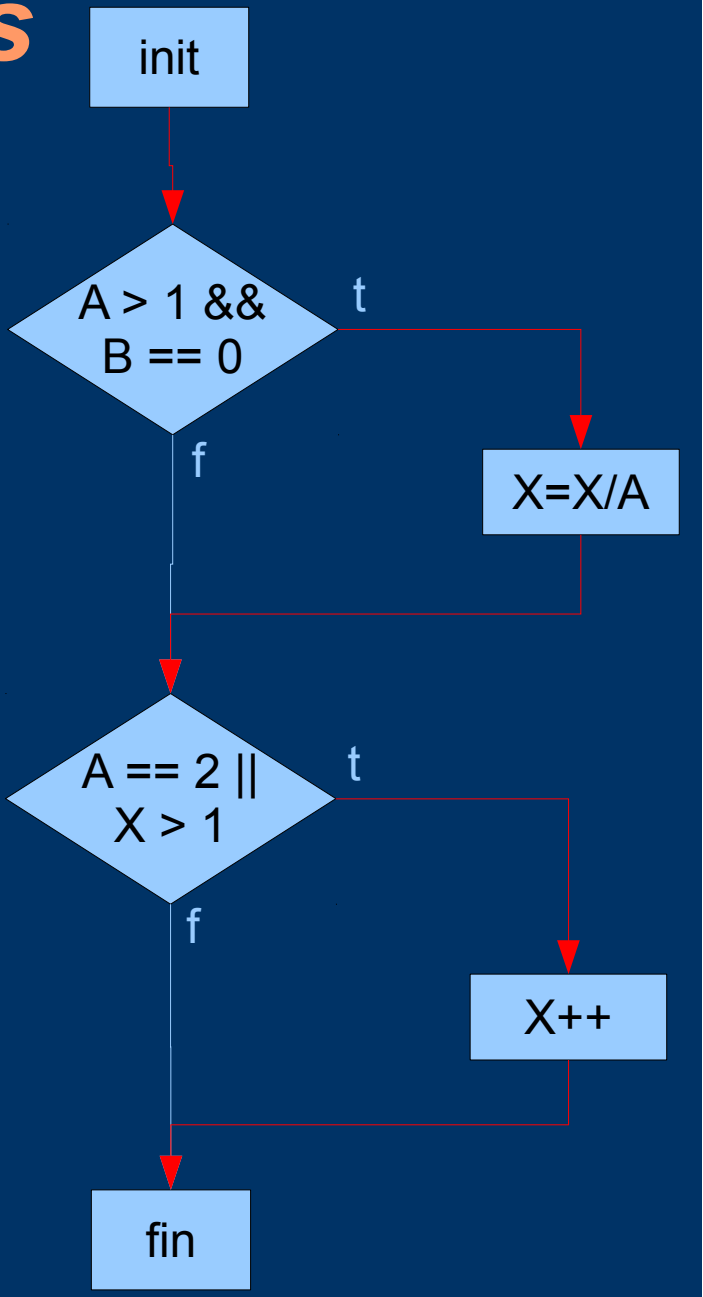
- Code java

```
If (A > 1 && B == 0) {
        X = X/A;
}
if A == 2 || X > 1) {
        X++;
}
```



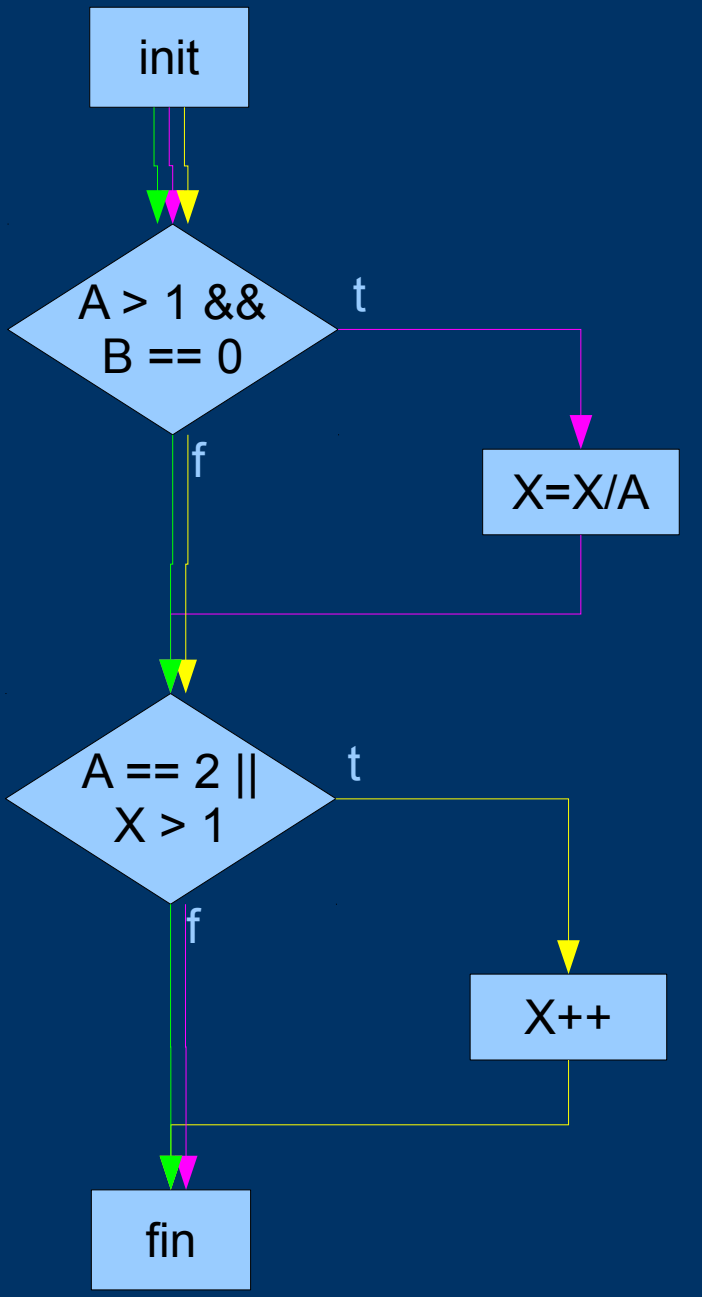
Exemple
Couverture des instructions

- On est loin de tout tester !



Exemple
Couverture des décisions

- Pour A>1 && B == 0
  - A = 3, B = 0 : true
  - A = 2, B = 1 : false
- Pour A == 2 || X > 1
  - A = 2, X = 1 : true
  - A = 3, X = 0 : false
- D'où les 2 cas :
  - A = 3, B = 0, X = 0
  - A = 2, B = 1, X = 1
  - Cheminement non testé

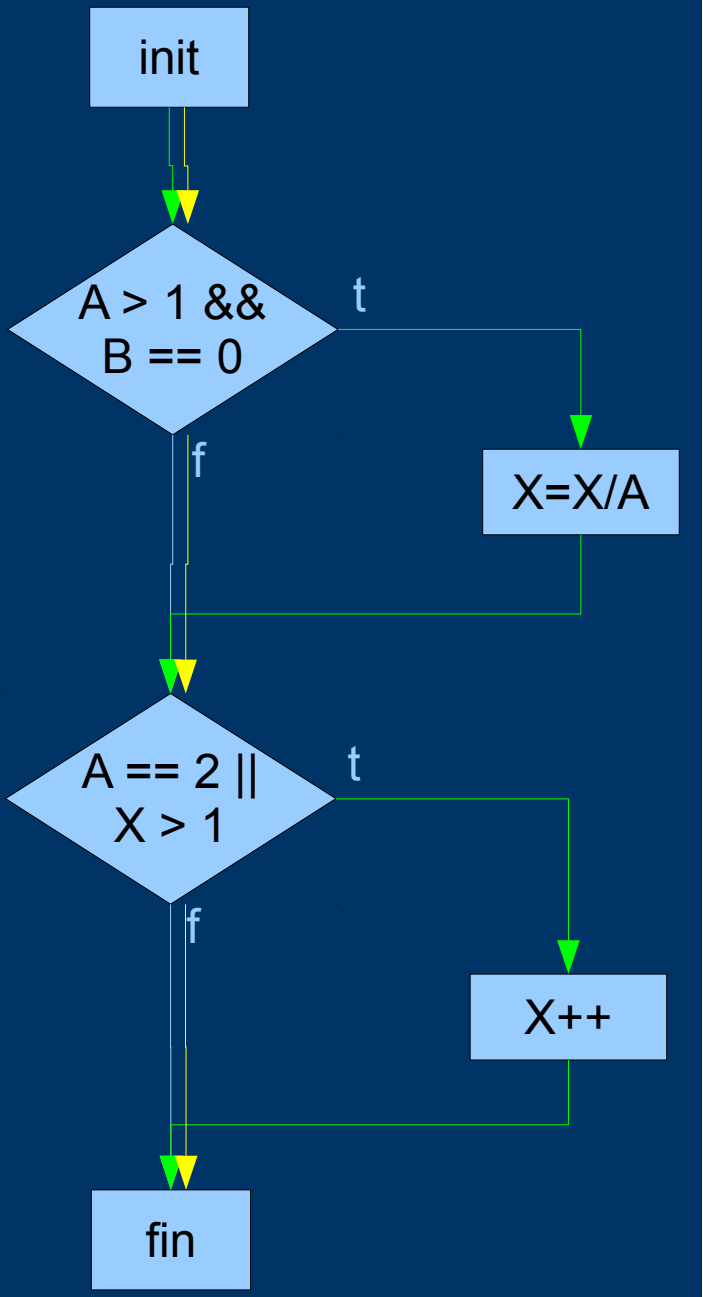


Exemple
Couverture des conditions

- Conditions :
  1. A > 1
  2. B == 0
  3. A == 2
  4. X > 1
- Cas choisis
  - A = 2, B = 0, X = 4

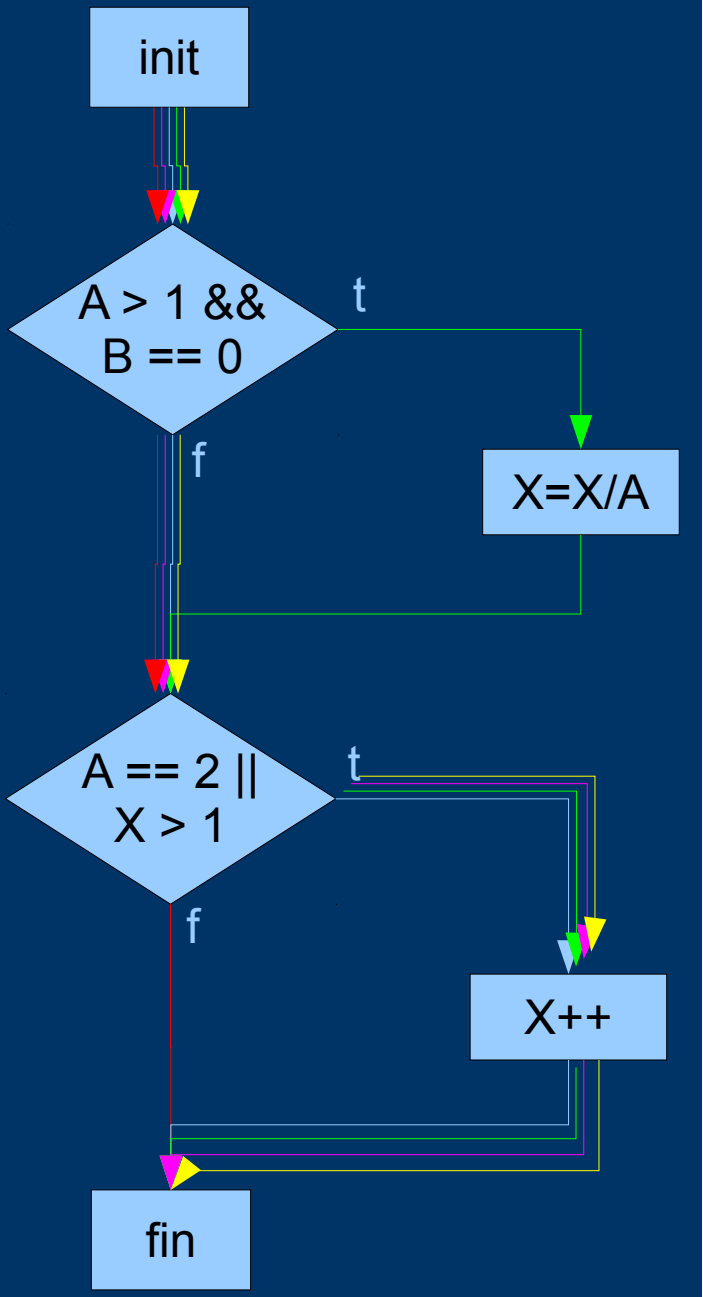
    1V, 2V,3V,4V
  - A = 1, B = 1, X = 1

    1F, 2F, 3F, 4F



Exemple : Couverture des conditions multiples

- Premier test, 4 cas
  - A > 1, B = 0
  - A > 1, B != 0
  - A <= 1, B =0
  - A <= 1, B != 0
- Deuxième test, 4 cas
  - A = 2, X > 1
  - A = 2, X <= 1
  - A != 2, X > 1
  - A != 2, X <= 1
- Cas de test choisis
  - A = 2, B = 0, X = 2
  - A = 2, B = 1, X = 0
  - A = 1, B = 0, X = 4
  - A = 1, B = 1, X = 1



Les Tests « boîte noire »

Différents types de tests « boîte noire »

- Partitionnement en classes d'équivalence
- Analyse des valeurs frontières
- (Graphique cause-effet)
- Prédiction des erreurs

Partitionnement en classes d'équivalence (PCE) : principe

- Rappel sur la pertinence d'un cas de test
  - Il réduit de plus d'une unité le nombre de tests à faire pour résoudre un but donné de test
  - Il couvre un large ensemble d'autres cas de test possibles
- Utilisation
  - D'après (2), on détermine un ensemble de conditions « intéressantes » à tester
  - Puis selon (1), on précise un ensemble minimal de cas de tests couvrant ces conditions

PCE :déterminer les classes d'équivalence

- A partir des spécifications
- Pour chaque entrée
- Déterminer aussi bien des classes pour les données valides que pour les données invalides
    - un entier *m* doit être entre 1 et 12 ; 3 classes

      1 <= m <= 12, m < 1, m > 12
    - Pour un type énuméré influençant un comportement, définir une classe par valeur

PCE :déterminer les cas de test

- Numéroter toutes les classes d'équivalence
- Pour les classes « valides »

  tant qu'elles ne sont pas toutes couvertes, générer un cas de test couvrant un maximum de classes non couvertes
- Pour les classes « invalides »

  tant qu'elles ne sont pas toutes couvertes, générer un cas de test couvrant exactement une classe d'équivalence

Analyse des valeurs frontières

- Motivation

  L'expérience montre que les tests aux valeurs frontières « rapportent plus » que les autres

- Valeur frontière :
  - borne d'une classe
  - Valeur immédiatement inférieure
  - Valeur immédiatement supérieure
- Différence par rapport au PCE
  - Plusieurs valeurs par classe
  - On s'intéresse aussi aux frontières des résultats

Prédiction des erreurs

- Essai de certaines valeurs types :
  - 0 pour les entiers
  - Liste vide, à un élément, à plusieurs éléments là où des listes de valeurs sont autorisées
  - Liste avec éléments identiques (2 personnes de même couple (nom, prénom) par exemple)
  - Cas clairement non précisés dans le cahier des charges

Synthèse
Stratégie de génération de cas de test

1. Pour les sorties dépendant de combinaisons de valeurs d'entrées, utiliser les graphiques cause-effet.
2. Procéder à une analyse des valeurs frontières.
3. Compléter par de la prédiction d'erreur.
4. Terminer par des tests « boîte blanche ».

Tests unitaires

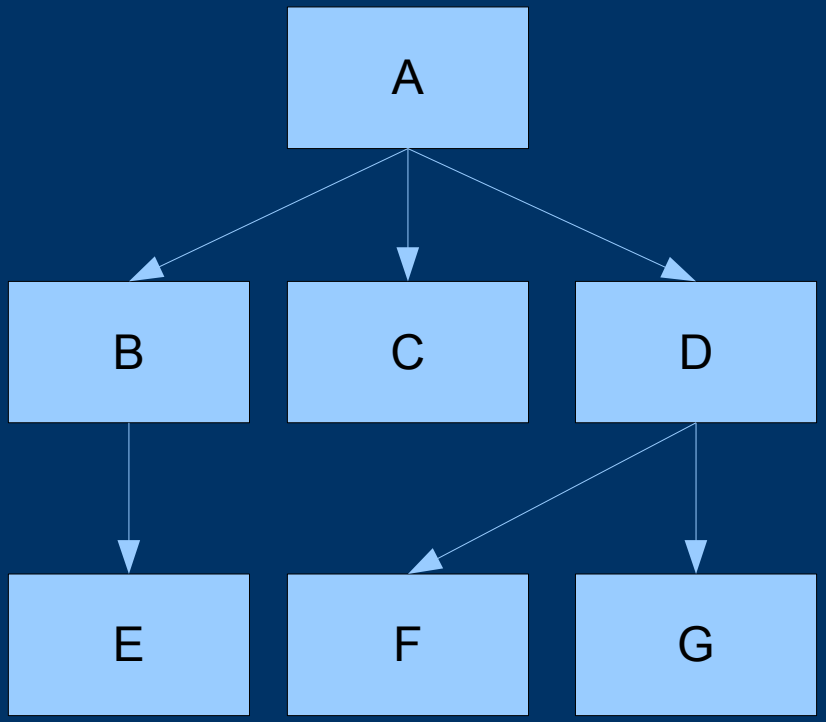
Tests unitaires
Introduction

- Motivations
  - Tests peuvent être faits relativement tôt
  - Seul endroit où les tests « boîte blanche » sont possibles
  - Permet de paralléliser les tests
- Pré-requis
  - Disposer du code du module
  - Disposer d'une spécification du module

Tests unitaires
Stratégie

- Commencer par des tests « boîte blanche »
- Compléter éventuellement par des tests « boîte noire »
- Choisir entre un test incrémental et un test non-incrémental
- Utiliser éventuellement des « mock objects » (objets leurres)

Tests unitaires
Test incrémental ascendant

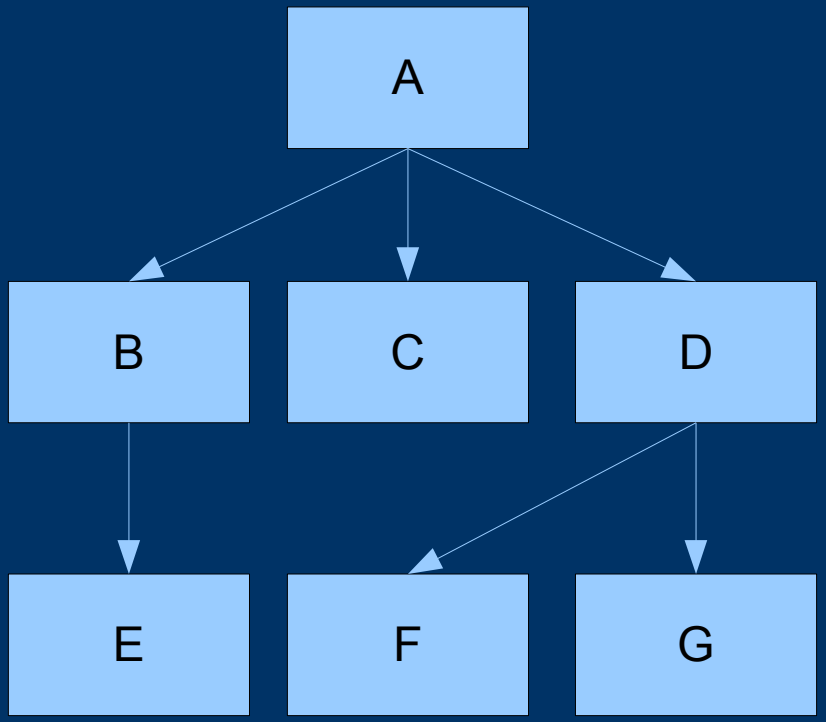


1. On teste E, C, F, G
2. On teste B, D
    1. Utilise E, F, G
3. On teste A
    1. Utilise B, C, D

➔ Nécessite

Une interface pour chacun des sous-modules (6)

Tests unitaires
Test incrémental descendant

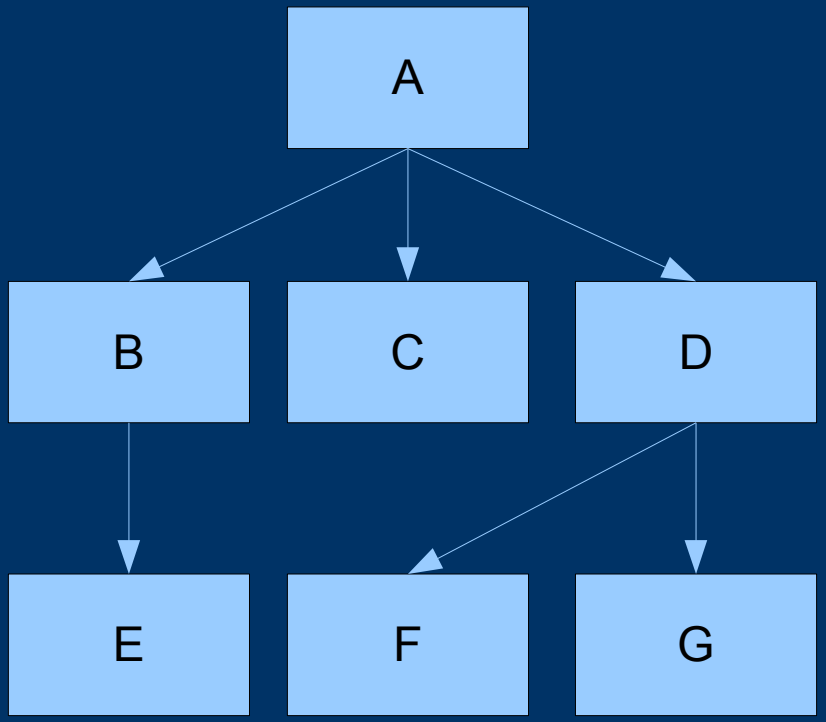


1. On teste A
2. On teste B, C, D
    1. Utilise A
3. On teste E, F, G
    1. Utilise B, D

➔ Nécessite

- Un objet leurre de B, C et D
- Un objet leurre de E, F, G

Tests unitaires
Test non-incrémental



Les modules sont testables indépendamment les uns des autres

➔ Nécessite

- Une interface pour B, C, D, E, F, G
- Un objet leurre de B, C, D, E, F, G

Tests unitaires incrémental vs non incrémental

- Plus de travail pour faire du test non incrémental (développer une interface/un leurre par module)
- Problèmes d'incompatibilité entre modules détectés plus tard en non incrémental
- Debuggage des erreurs d'incompatibilités plus compliqué
- Test incrémental requiert plus de temps machine
- Test non incrémental autorise plus de parallélisation des tests

Tests Unitaires
Cas d'échec ?

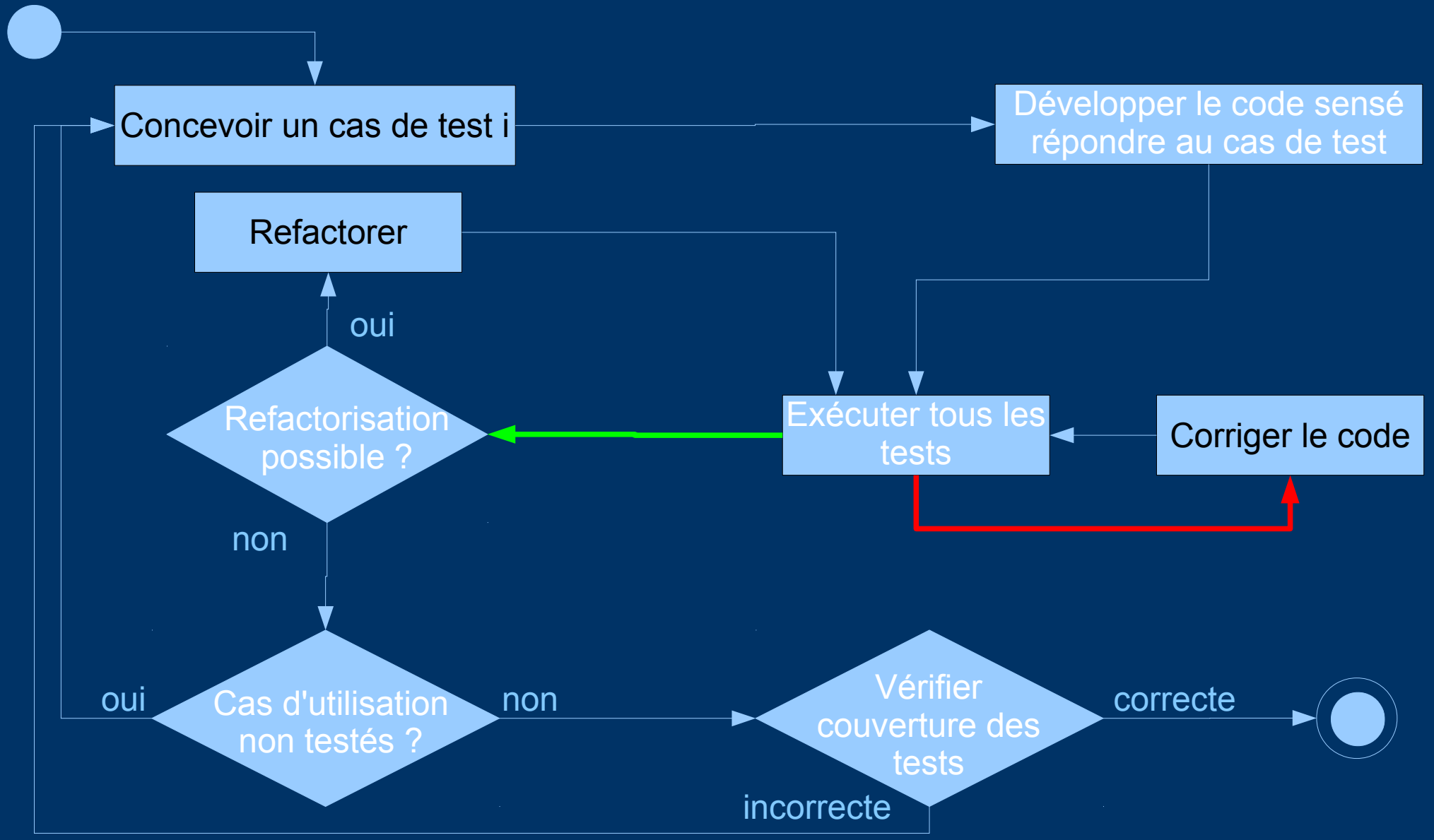
- Le résultat attendu dans le cas de test est-il correct ?
    - Tester les cas de tests
    - Ne pas tester les modules que l'on développe

Tests d'ordres supérieurs

Tests d'ordres supérieurs
Types de tests

- Tests fonctionnels
- Tests systèmes
- Tests des fonctionnalités implantées
- Tests en volume
- Tests de charge
- Tests d'ergonomie
- Tests de sécurité
- Tests de performance
- Tests de stockage
- Tests de configuration
- Tests de compatibilité
- Tests d'installabilité
- Tests de récupération
- Tests de documentation

Tests et méthodes Agiles
Test Driven Development (TDD)



Débuggage
Méthodes brutes

- 3 types
  - Debuggage à partir d'un « dump » mémoire
    - Difficile (taille potentiellement grande)
    - Ne prend en considération qu'un état (statique) du programme
  - Debuggage par l'ajout d'affichages
    - Aléatoire, long
    - Modifie le programme
      - Nécessite des recompilations
      - Peut masquer le bug
  - Debuggage via l'utilisation de debuggueurs
    - Aléatoire et long
- Inconvénient
  - Dispense de réfléchir

Program Inspections, Walkthroughs, et Reviews : Tests "humains"

Relecture du code
Histoire

- Longtemps négligé
- De plus en plus utilisé car
  - Détecte les erreurs plus tôt
    - Coûte moins cher
    - Réparation meilleure car moins de stress par rapport au planning
- Utilisée différemment suivant les entreprises :
  - Taille de l'équipe de dvlpt
  - Taille et complexité de l'application
  - Culture de l'entreprise

Inspections et pas-à-pas

- But = détecter (to test), pas corriger (to debug)
- Principes communs
  - Équipe de 3-4 développeurs, 1 seul ayant participé à l'écriture du code
  - Une erreur détectée ainsi correspond souvent à la correction de potentiels échecs de cas de test
  - Détecte essentiellement les erreurs de conception logique ou de codage (30% à 70% de ces erreurs sont corrigées), rarement les erreurs de conception de haut niveau
  - Ne détecte pas les mêmes types d'erreurs que les tests
  - Plus adapté au test de modifications de programme

Inspection : composition de l'équipe

- 4 personnes :
  - Un "modérateur"
  - L'auteur du programme
  - Le concepteur du programme
  - Un spécialiste des tests
- Modérateur
  - Caractéristiques
    - Compétent
    - Connaît bien le code
  - Rôle
    - Planifier et mener l'inspection
    - Mener la session
    - Noter les erreurs trouvées
    - Par la suite, s'assurer que les erreurs seront corrigées

Inspection : organisation

- Plusieurs jours avant la session, conception et code distribués aux membres de l'équipe par le modérateur
- Les documents doivent avoir été lus attentivement pour la session
- Organisation de la session
  - 1,5h – 2h
  - Environ 150 lignes/heure
- Pendant la session :
  - le programmeur commente la logique de son code instruction par instruction
    - Pendant ce temps, les autres participants soulèvent les erreurs qu'ils détectent
  - Puis le programme est vérifié par rapport à une check-list

Inspection : suite

- Après la session
  - Le programmeur corrige les bugs
  - Si beaucoup de bugs ou une grosse correction, le modérateur organisera une nouvelle inspection
- Etat d'esprit
  - Le programmeur ne doit pas se sentir agressé
  - Les résultats doivent être confidentiels pour éviter les jugements sur le programmeur
  - Doit profiter à tout le monde de part les échanges d'idées et styles de programmation

Check-list
Erreurs de référence à des données

- Une variable utilisée est-elle initialisée ?
- L'accès à un tableau est-il dans les bornes ? Les indices sont-ils bien entiers ?
- Un pointeur pointe-t-il sur une zone allouée ?
- En cas d'héritage, le "contrat" implicite est-il correctement implanté par les redéfinitions qui s'imposent ?

Check-list
Erreurs de référence à des données

- Une variable utilisée est-elle initialisée ?
- L'accès à un tableau est-il dans les bornes ? Les indices sont-ils bien entiers ?
- Un pointeur pointe-t-il sur une zone allouée ?
- En cas d'héritage, le "contrat" implicite est-il correctement implanté par les redéfinitions qui s'imposent ?

Check-list
Erreurs de déclaration de données

- Les variables sont-elles toutes déclarées (par exemple, si internes à un bloc/méthode, risque de confusion avec des variables externes au bloc/d'instance)
- Les valeurs par défaut sont-elles connues et justifiées ?
- Les types et taille (pour les tableaux) sont-ils corrects ?
- Si des variables ont des noms proches, est-ce normal ou bien s'agit-il de deux versions de la même donnée ?

Check-list
Erreurs de calcul

- Si mélange d'entiers et réels, les conversions automatiques sont-elles bien comprises ?
- En cas de division avec des entiers, le résultat est-il bien appréhendé ?
- Y a-t-il risque de perte d'information lors d'une affectation (par exemple un calcul en double vers un float) ? Si oui, est-ce normal et évalué correctement ?
- Y a-t-il risques de dépassement dans les calculs (overflow) ou d'approximation par zéro (underflow) ?
- Si les réels sont représentés en binaire, les conséquences sur les calculs sont-elles prises en compte ?
- Les intervalles de variation d'une variable sont-ils vérifiés (proba entre 0 et 1 par exemple)
- Si plusieurs opérateurs, les règles des priorités s'appliquant donnent-elles le résultat souhaité ?

Check-list
Erreurs de comparaison

- Les opérateurs de comparaison utilisés sont-ils les bons ? Notamment, sens large vs sens strict ?
- Les tests avec des opérateurs booléens et/ou/non sont-ils corrects ?
- Les évaluations paresseuses sont-elles maîtrisées ?
- Les comparaisons sur les réels doivent-elles utiliser l'égalité stricte ou avec une marge ? Celle-ci est-elle évaluée correctement ?

Check-list
Flux d'exécution

- Les boucles se terminent-elles toujours ?
- Un programme/fonction se termine-t-il ?
- Est-il possible qu'on ne rentre jamais dans le corps d'une boucle ? Est-ce normal ?
- Quand on peut sortir d'une boucle soit après un nombre fini d'itération connu au départ, soit en cours de route, les 2 types de sortie sont-ils correctement gérés ?
- Les compteurs initialisés à zéro sont-ils correctement utilisés ?
- En cas de tests non-exhaustif, cela est-il correct ? le choix gardé par défaut se justifie-t-il ?

Check-list
Erreurs d'interface

- Les paramètres d'une méthode sont-ils passés dans le bon ordre ?
- Les unités côté "appelant" sont-elles les mêmes que côté "appelé" (exemple : angle en degré/radian) ?
- Certains paramètres qui ne devraient être que des paramètres d'entrée risquent-ils d'être modifiés par une méthode ?

Check-list
Erreurs d'entrée/sortie

- Les références aux fichiers sont-elles créées avec les bons attributs ?
- Y a-t-il suffisamment de mémoire pour travailler sur le fichier ?
- Les fichiers sont-ils bien ouverts avant d'être utilisés ?
- Les fichiers sont-ils bien fermés après utilisation ?
- Les fins de fichiers sont-elles détectées et gérées correctement ?

Pas-à-pas
Principes

- Assez similaire à l'inspection
    - Réunion d'1 à 2 heures
    - 3-5 personnes
    - Documents fournis quelques jours avant
- Composition de l'équipe
    - Un modérateur
    - Un secrétaire (note les erreurs trouvées)
    - Le programmeur
    - Le testeur : arrive à la réunion en ayant préparé quelques cas de test

Pas-à-pas
Organisation

- L'équipe simule l'exécution du programme sur chacun des cas de tests prévus par le "testeur"
- L'état du programme est maintenu à jour sur un tableau

Autres méthodes d'analyse statique
Vérification de Bureau

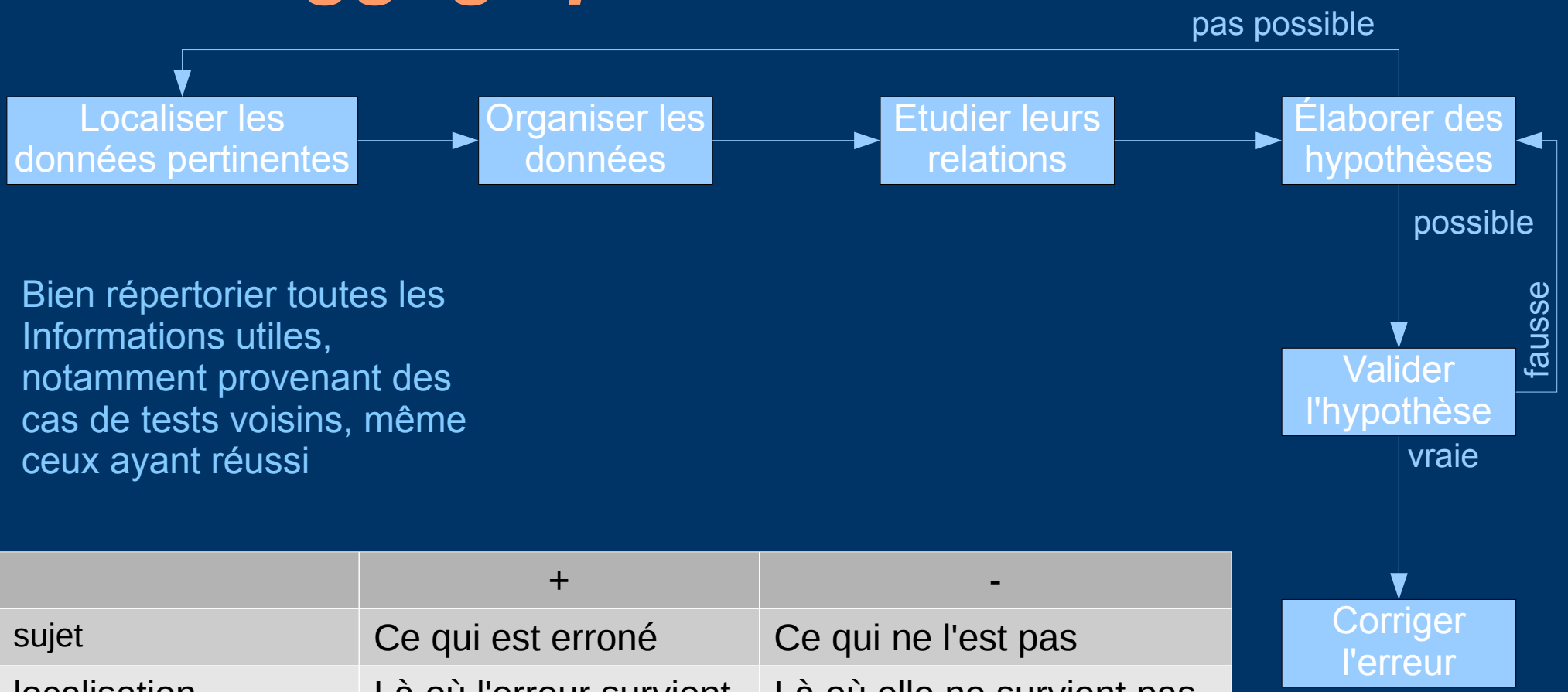
- Méthode plutôt ancienne
- Une personne seule analyse le code
- Inconvénients
    - Si c'est le programmeur, viole un des principes de base
    - Sinon, ne connaît pas le code, donc peu efficace

=> Préférer l'inspection ou le pas-à-pas.

Autres méthodes d'analyse statique
Evaluation par ses pairs

- Principe
  - Réunion de 6 à 20 programmeurs, donnant chacun de façon anonyme un de leur meilleur code et un moins bon
  - Les codes sont mélangés, et chaque programmeur doit analyser 2 codes "bon" et 2 code "moyens" de ses pairs
- Critères d'évaluation (note de 1 à 7 ; 1 = oui, 7 = non)
  - Le programme est-il facile à comprendre ?
  - La conception de haut niveau est-elle visible et raisonnable ?
  - La conception de bas niveau est-elle visible et raisonnable ?
  - Le programme semble-t-il facile à modifier ?
  - Seriez-vous fier d'avoir écrit ce programme ?
  - Commentaires généraux et suggestions d'amélioration
- Retour
  - Évalution de ses programmes et positionnement par rapport aux autres
  - Comparaison de ses évaluations des autres programmes par rapport aux autres évaluations faites sur les mêmes programmes ?

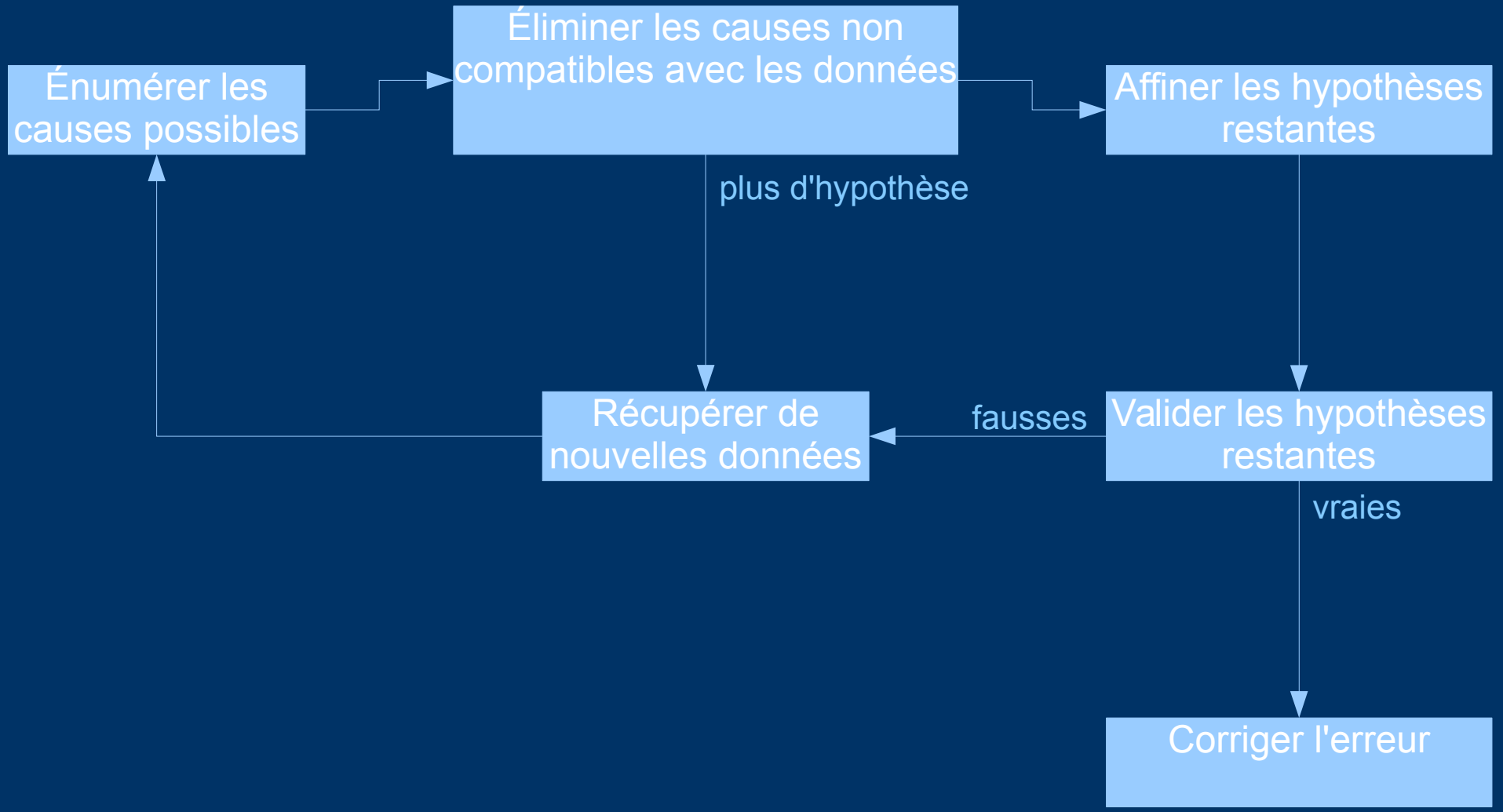
Débuggage par induction



Bien répertorier toutes les Informations utiles, notamment provenant des cas de tests voisins, même ceux ayant réussi

	+	-
sujet	Ce qui est erroné	Ce qui ne l'est pas
localisation	Là où l'erreur survient	Là où elle ne survient pas
quand	Particularité des cas avec erreur	Particularité des cas corrects
Dans quelle mesure	Portée et ampleur	

Débuggage par déduction



Débuggage par back-tracking

Analyse du code source

- Partir du lieu où l'erreur survient
- Remonter dans le code en pas-à-pas

Débuggage par test

- Générer des cas de tests spécifiques au débuggage
    - Partir du/des cas de tests ratés
    - Concevoir des variantes de ce/ces cas de tests pour mieux cerner les conditions engendrant l'erreur
- A utiliser souvent conjointement avec le débuggage par induction ou le débuggage par déduction

Principes du débuggage
1. localisation de l'erreur

- Réfléchir
- Si on sèche, mettre de côté et y revenir plus tard
- Si on sèche, décrire le problème à quelqu'un d'autre
- N'utiliser les outils de débuggage qu'en deuxième ressort
- Éviter les expérimentations hasardeuses

Principes du débuggage
2. correction de l'erreur

- Là où il y a un bug, il y en a surement d'autres
- Corriger l'erreur et pas un de ses symptômes
- La probabilité que la réparation soit correcte n'est pas de 100%
- La probabilité que la réparation soit correcte diminue lorsque la taille du code augmente
- La réparation d'une erreur doit amener éventuellement à revoir la conception
- Changer le code source, pas le code objet

Débuggage
Retour d'expérience

Se poser les questions suivantes

- Où l'erreur a-t-elle eu lieu ?
- Qui a fait l'erreur ?
- Qu'est-ce qui a été mal fait ?
- Comment l'erreur aurait-elle pu être évitée ?
- Pourquoi l'erreur n'a-t-elle pas été détectée plus tôt ?
- Comment l'erreur aurait-elle pu être détectée plus tôt ?