

Vérification et validation du logiciel

Introduction Généralités, Problèmes, Contexte

Cours LOG3430, Polytechnique Montréal
(Auteur Principal: Giulio Antoniol)

Définitions de base

- ❑ **Logiciel:** Un ensemble de programmes, de procédures, de documentation associée et les données qui concernent le fonctionnement d'un système informatique (ISO 24765)
 - **Ensemble de programmes:** code source ayant fait l'objet de spécifications, conception, revues, essais, ...;
 - **Données:** inventoriées, modélisées, normalisées, créées lors de la production du logiciel;
 - **Processus:** processus d'affaires des utilisateurs, décrits, étudiés et optimisés;
 - **Règles:** règles d'affaires décrites, validées, implantées et testées;
 - **Documentation:** véhicule de communication, réalisée à toutes les étapes clé du cycle de vie.

Assurance Qualité – Vérification vs. Validation (IEEE Std 829)

❑ **Groupe d'assurance-qualité logicielle**

- Équipe de personnes avec les compétences et la formation nécessaires pour s'assurer que toutes les actions nécessaires sont prises durant le processus de développement afin que le logiciel résultant soit conforme aux exigences techniques établies.

❑ **Vérification**

- Évaluation d'un système ou composant logiciel pour déterminer si les produits d'une phase donnée du développement sont conformes aux conditions imposées au début de cette phase. Activités: inspections et revues par les pairs de divers livrables associés au logiciel, etc.

❑ **Validation**

- Évaluation d'un système ou composant logiciel durant ou à la fin d'un cycle de développement, dans le but de déterminer si les exigences spécifiées sont satisfaites. La validation est habituellement associée à des tests traditionnels basés sur l'exécution, c'est-à-dire en sollicitant le logiciel avec des cas de test.

Missions de l'Assurance Qualité

- ❑ Découvrir les fautes dans les documents *où elles sont introduites*, d'une manière systématique, afin d'éviter les effets de propagation. Les *études structurées et systématiques* de documentation logicielle sont appelées **inspections**.
- ❑ Dériver, d'une manière *systématique*, des ***cas de test*** efficaces pour découvrir les anomalies.
- ❑ Automatiser et étendre les activités de test et d'*inspection* à leur *maximum* possible.
- ❑ Surveiller et contrôler la *qualité*, ex., fiabilité, facilité de maintenance, sécurité, à travers *toutes* les phases et les activités du projet.
- ❑ Tout ceci implique de mesurer par une métrique appropriée le produit logiciel et ses processus de même qu'une *évaluation empirique* du test et des technologies d'inspection.

Défaillances logicielles: Histoires « drôles »



- ❑ Hounslow, West London (2009). **Un distributeur automatique Tesco donne deux fois le retrait demandé après une "erreur opérationnelle", un "computer glitch"**
- ❑ "It makes a change for us to have the joke on the banks for once."
- ❑ **Bank:** "If the people using the ATM see it as a bit of fun, so be it."
- ❑ Des événements similaires en 2008, 2011, Europe, Amérique du Nord

Défaillances logicielles: Histoires « secrètes »

????????????????

????????????????

????????????????

Défaillances logicielles: Histoires « d'horreur »



Ariane 5, Vol 501 (1996)

← 4000 m et 40 s →

- Pertes: 370 millions \$ US
- Cause (entre autres):

Mauvaise gestion d'exceptions

PRE: $-32768 \leq x \leq +32767$

POST: $y = \text{int16}(x)$



Machine de radio-thérapie Therac-25 (85-87)

- Sur-Radiations (jusqu'à 100 fois la dose)
- Au moins 5 morts
- Cause: Aucun test sur les entrées non-standard

Mars Climate Orbiter (1999)

- Écrasement du vaisseau sur Mars
- Pertes: 300 millions \$US
- Cause (entre autres):
Mélange des systèmes impérial et métrique

Le cas ARIANE 5

Communiqué de presse, Juin 96

❑ **Paris, 4 Juin 1996, Communiqué de presse de ESA/CNES, ARIANE 501**

Le premier lancement d'Ariane 5 n'a pas résulté à une validation du nouveau missile européen. C'était le premier test de vol d'un véhicule entièrement nouveau dont chacun des éléments avait été testé au sol durant les années et mois passés.

D'une conception entièrement nouvelle, le missile utilise des moteurs dix fois plus puissants que la série d'Ariane-4. Son cerveau électronique est cent fois plus puissant que ceux utilisés pour les fusées Ariane précédentes. **Les nombreuses études de qualification et essais terrestres ont imposé des vérifications extrêmement strictes sur l'efficacité de tous les choix faits. Il n'y a, toutefois, aucune garantie absolue. La capacité d'une fusée ne peut être démontrée qu'en vol sous des conditions réelles de lancement.**

Un second test qui est déjà prévu sous le plan du développement aura lieu dans un intervalle de quelques mois. Avant cela, tout devra être entrepris pour établir les raisons de ce contretemps et faire les corrections nécessaires pour un second test réussi. Une commission d'enquête sera organisée dans les prochains jours. Il sera requis de soumettre, pour la mi-juillet, un rapport entièrement indépendant identifiant les causes de cet incident et proposant les modifications envisagées afin de prévenir tout autre incident.

Ariane 5 est un défi majeur pour les activités spatiales en Europe. L'habileté de toutes les équipes impliquées dans le programme, jointe à la détermination et à la solidarité de toutes les autorités politiques, techniques et industrielles, nous assurent d'une issue réussie.

Le cas ARIANE 5

Origine de l'échec

- ❑ Source : Échec d'ARIANE 5 Vol 501,
Reportage de la revue Inquiry Board

Un segment de programme pour convertir un nombre en virgule flottante en un nombre entier de 16 bits a été exécuté avec une valeur de donnée d'entrée en dehors de la zone représentable par un nombre entier signé de 16 bits. Cette erreur de version exécutable (hors limite, surcapacité), qui s'est produite dans les deux ordinateurs actifs et de sauvegarde à peu près en même temps, a été détectée et les deux ordinateurs se sont eux-mêmes éteints. Ceci a résulté en la perte totale du contrôle d'altitude. Ariane 5 a tourné de manière incontrôlée et des forces aérodynamiques ont pulvérisé le véhicule en morceaux. Cette détérioration a été détectée par un moniteur à bord qui a déclenché les charges explosives qui ont détruit le véhicule dans les airs. **Ironiquement, le résultat de cette conversion de format n'était plus nécessaire après le décollage.**

Le cas ARIANE 5

Leçons à tirer.

- ❑ Les procédures réutilisées (depuis Ariane 4 dans ce cas-ci) doivent l'être testées de manière rigoureuse, incluant les tests *basé sur l'utilisation (des profils opérationnels)*.
- ❑ Stratégies adéquates de gestion des exceptions (sauvegarde, gestion de procédures dégradées).
- ❑ Spécifications documentées de manière claire et complète (ex., pré-conditions, post-conditions)

Ce n'était pas un problème informatique complexe, mais une déficience de pratiques de l'ingénierie logicielle mises en place ...

Problèmes dominants

Rappel: Nature du développement logiciel:

Pas une production. Utilisation intensive de l'effort humain.

Ingénierie, mais aussi processus social. Systèmes logiciels de plus en plus complexes, impliquant diverses industries.

- ❑ Le logiciel est le plus souvent livré en retard, avec des surcoûts et une qualité insatisfaisante.
- ❑ La validation et la vérification de logiciel sont rarement systématiques et ne sont pas souvent basées sur des techniques sûres et bien définies.
- ❑ Les processus de développement de logiciel sont habituellement instables et incontrôlés.
- ❑ La qualité du logiciel est naïvement mesurée, surveillée et contrôlée.

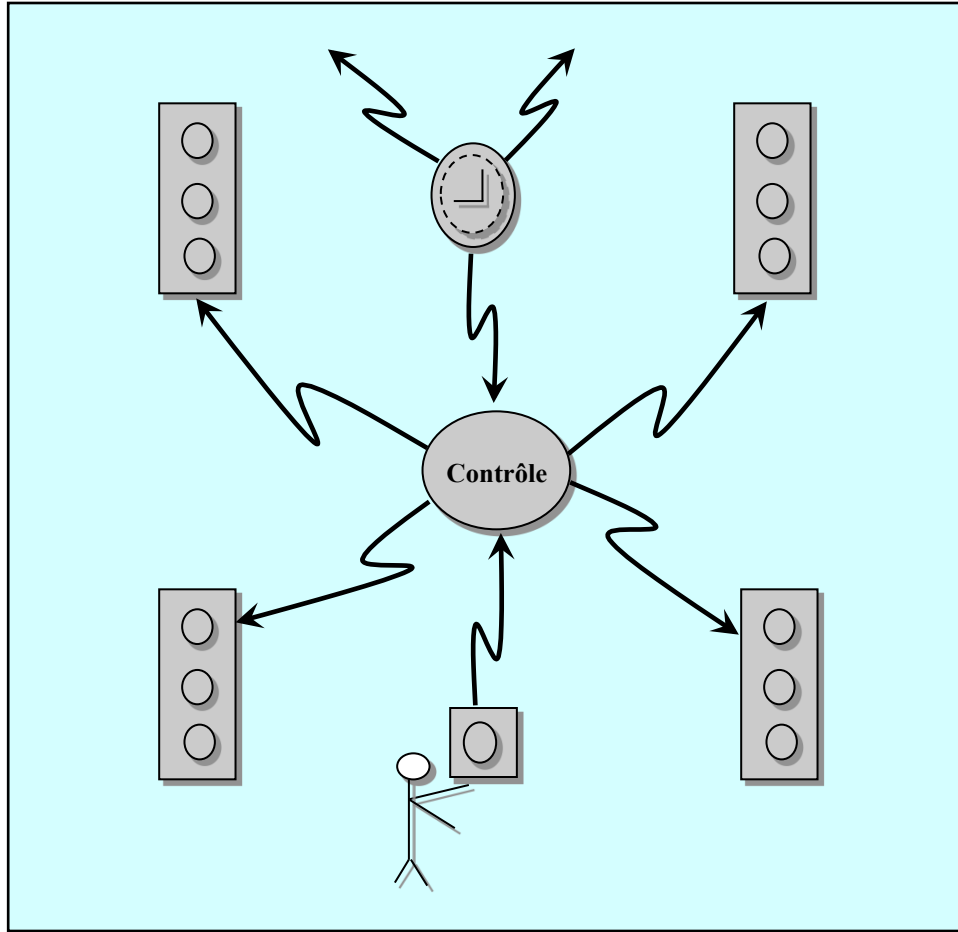
Conséquences chiffrées d'une qualité faible

- ❑ L'enquête du groupe Standish (1994)
 - 350 compagnies, plus de 8000 projets.
 - 31% des projets ont été annulés, seulement 9-16% ont été livrés dans les coûts et budgets prévus
- ❑ Étude américaine (1995) :
81 milliards \$US dépensés par année pour des projets de développement de logiciel qui ont échoué.
- ❑ Étude NIST (2002) :
 - les bogues coûtent 59.5 milliards \$ par année.
 - Une détection anticipée aurait pu sauver 22 milliards \$.

Qualités des produits logiciels

- ❑ Exactitude
- ❑ Fiabilité
- ❑ Robustesse
- ❑ Performance
- ❑ Convivialité
- ❑ Facilité de vérification
- ❑ Facilité de maintenance
- ❑ Facilité de réparation
- ❑ Facilité d'évolution
- ❑ Facilité de réutilisation
- ❑ Portabilité
- ❑ Facilité de compréhension
- ❑ Interopérabilité

Exemple : feu de signalisation



Exactitude, fiabilité :
laisse le trafic passer
selon un modèle correct
et une programmation
centralisée.

Robustesse, sécurité :
fournit une fonction
dégradée quand c'est
possible ; ne signale
jamais des verts en
conflit.

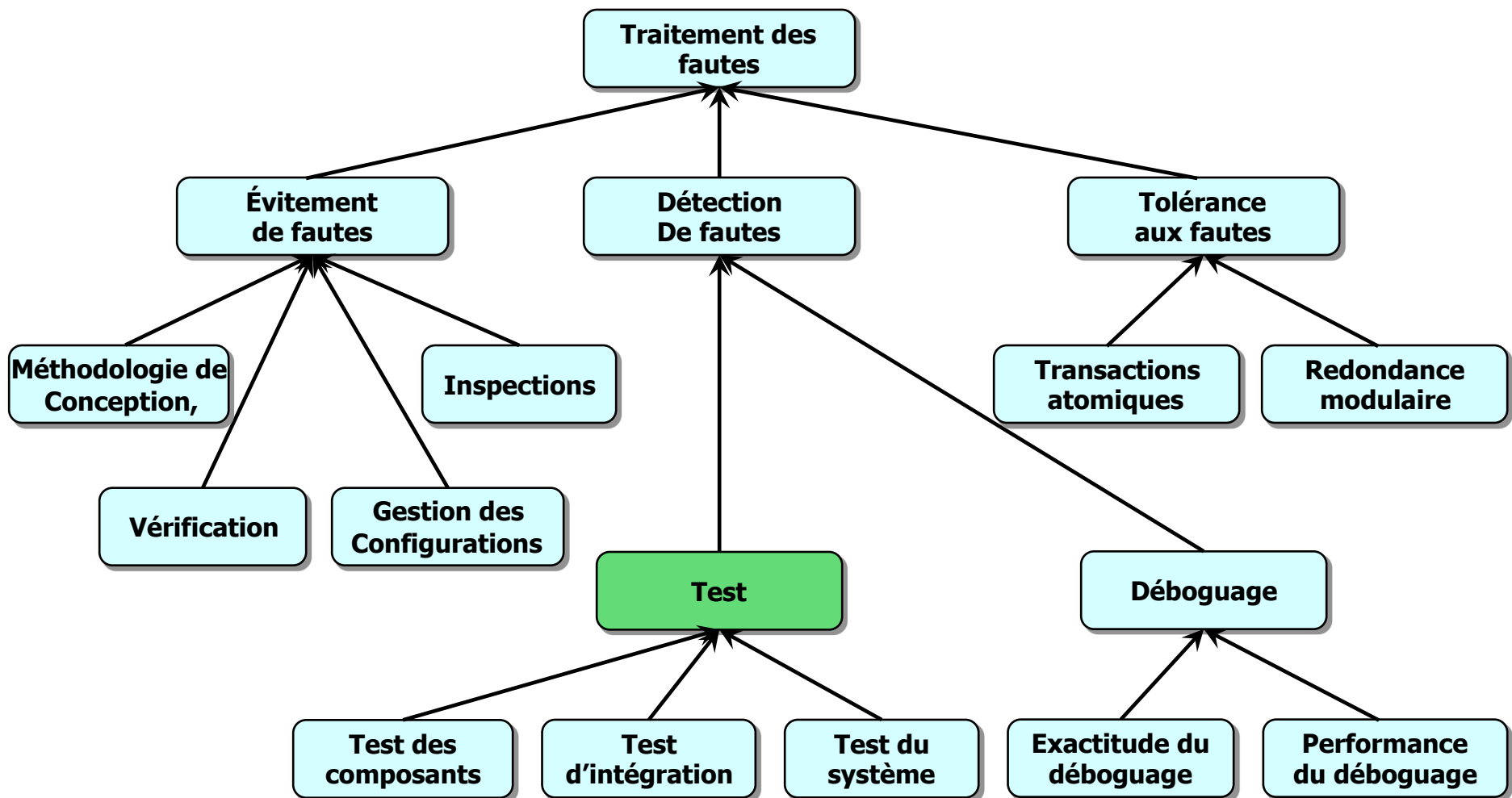
Subtilités de la sûreté de fonctionnement du logiciel

- ❑ Un programme est correct s'il obéit à sa spécification. On ne peut établir qu'un programme est correct mais on peut en évaluer la fiabilité (approximation statistique)
- ❑ Un programme est robuste s'il agit raisonnablement sous des conditions sévères, inhabituelles ou illégales.
- ❑ Spécification inadéquate ou partielle → correct mais pas sécuritaire ou robuste.
- ❑ Défaillances Rares → Fiable mais pas correct
- ❑ Défaillances « gênantes » → Sécuritaire mais pas correct
- ❑ Défaillances catastrophiques → peut-être Robuste mais pas sécuritaire

Les besoins de sûreté de fonctionnement varient

- ❑ Les applications à sécurité-critique :
 - les systèmes de contrôle aérien ont des besoins stricts en matière de sécurité;
 - les systèmes de télécommunications ont des besoins stricts en matière de robustesse.
- ❑ Les produits de masse :
 - la fiabilité est moins importante que le délai de livraison au marché.
- ❑ Cela peut varier dans la même classe de produits :
 - la fiabilité et la robustesse sont des aspects clés pour les systèmes d'exploitation multi-utilisateurs (e.g., UNIX) mais moins importants pour les systèmes d'exploitation mono-utilisateur (e.g., Windows or MacOS)

Traitement des fautes logicielles



Définitions & Objectifs du test

Erreurs, Défauts, Défaillances (1/3)

❑ **Erreur (*Error*)**

- Action humaine qui produit un résultat incorrect (ISO 24765).

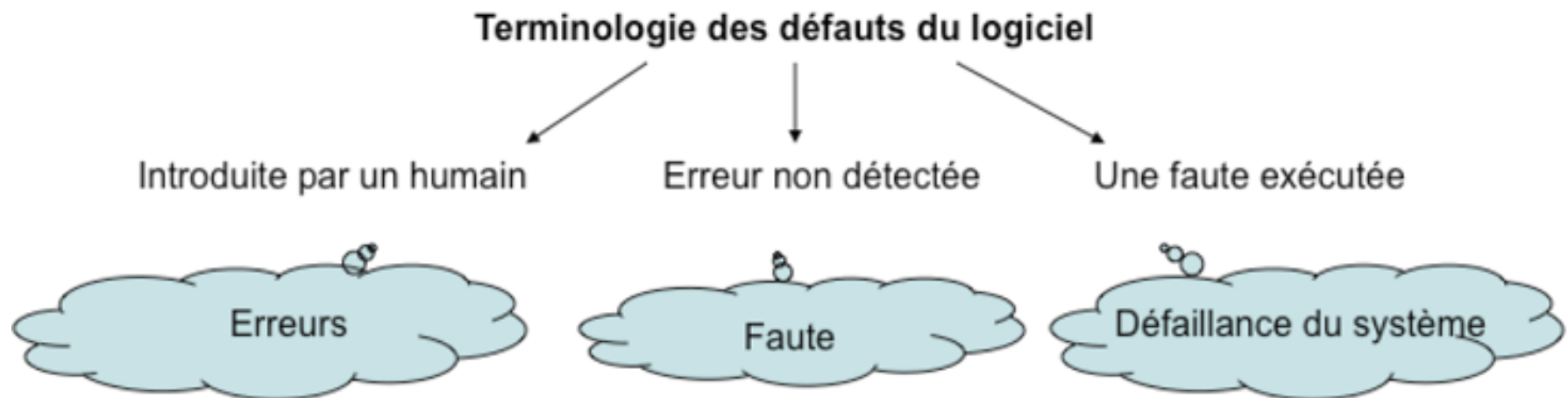
❑ **Défaut, faute ou bug (*Defect, Fault, or Bug*)**

- Une erreur qui, si elle n'est pas corrigée, pourra causer une défaillance (*failure*) ou produire des résultats incorrects (ISO 24765).
- Un défaut est introduit dans le logiciel comme **conséquence** d'une erreur.

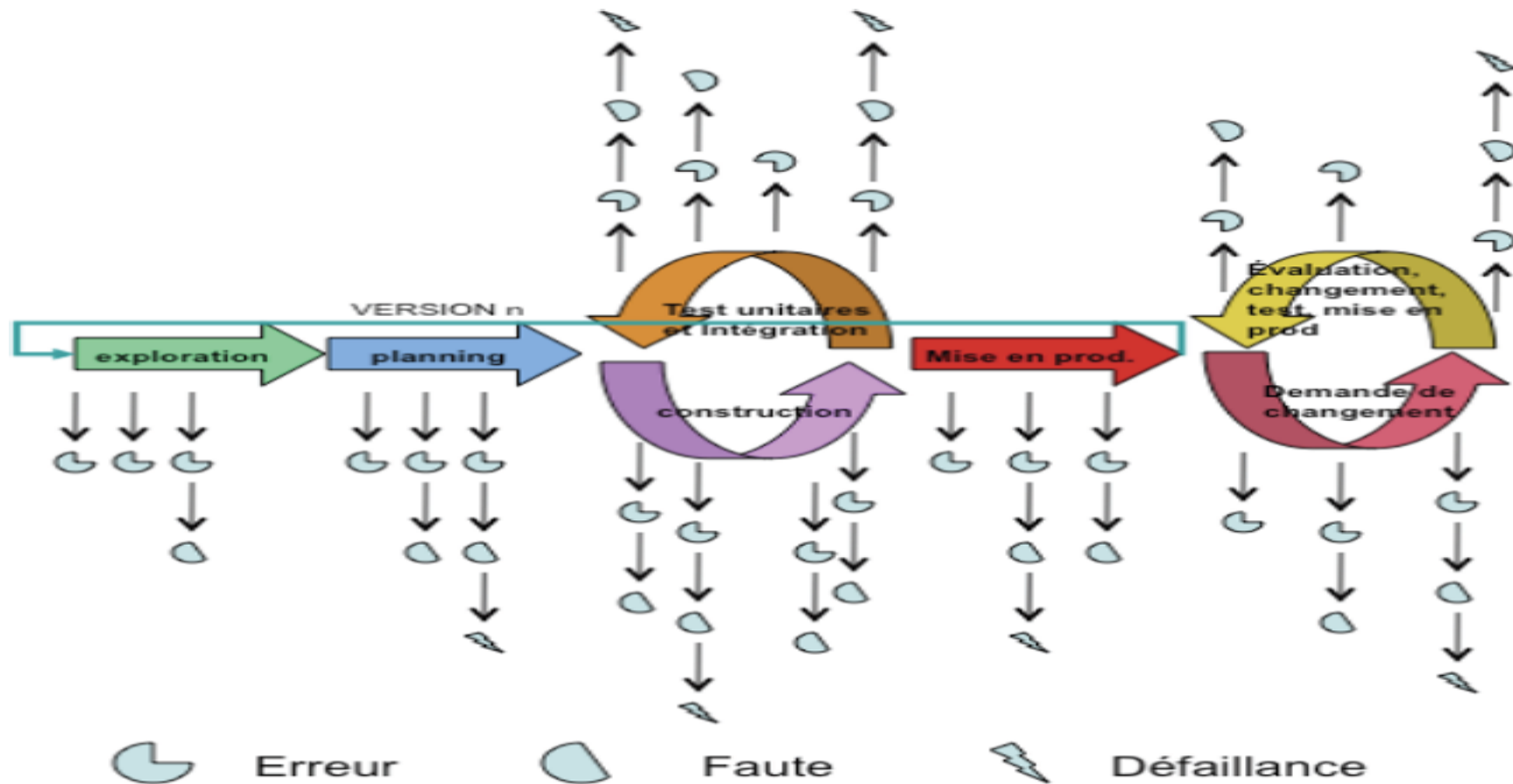
❑ **Défaillance ou panne (*Failure*)**

- Cessation de l'aptitude d'un produit à accomplir une fonction requise ou de son incapacité à s'en acquitter à l'intérieur des limites spécifiées précédemment (ISO 25000).

Erreurs, Défauts, Défaillances (2/3)



Erreurs, Défauts, Défaillances (3/3)



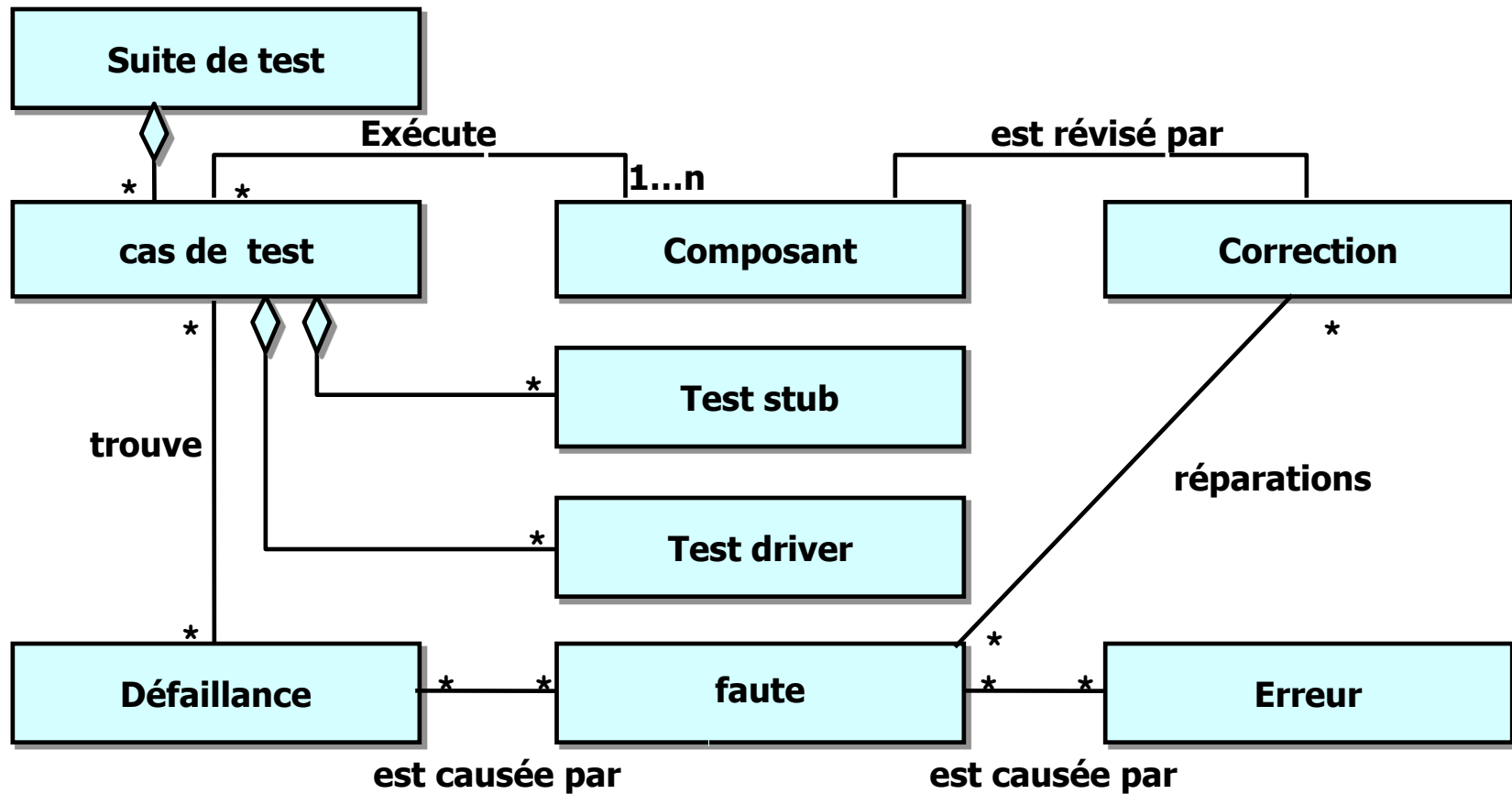
Tests vs. Debogage

- ❑ **Test de logiciel** : Exécuter le logiciel pour trouver les fautes ou renforcer la confiance que l'on a dans le système.
- ❑ **Cas de test**:
 - Un ensemble de données d'entrée de test: reçues d'une source externe (matérielle, logicielle, ou humaine) par le code testé.
 - Des conditions d'exécution: requises pour exécuter le test, par exemple une base de données qui se trouve dans un certain état, ou une configuration particulière d'un dispositif matériel.
 - Les sorties attendues. Il s'agit des résultats spécifiés que le code testé doit produire.
- ❑ **Jeu (Suite) de tests** : Ensemble de cas de tests.
- ❑ **Débogage (localisation de défauts)**
 - localiser le défaut; réparer le code; tester le code à nouveau.

Harnais de tests: Substituts (stubs) et Drivers

- ❑ *Test Stub* : implémentation partielle d'un composant duquel dépend le composant testé.
- ❑ *Test Driver*: implémentation partielle d'un composant qui dépend du composant testé.
- ❑ Test stubs et test drivers permettent aux composants d'être isolés du reste du système pour être testés.

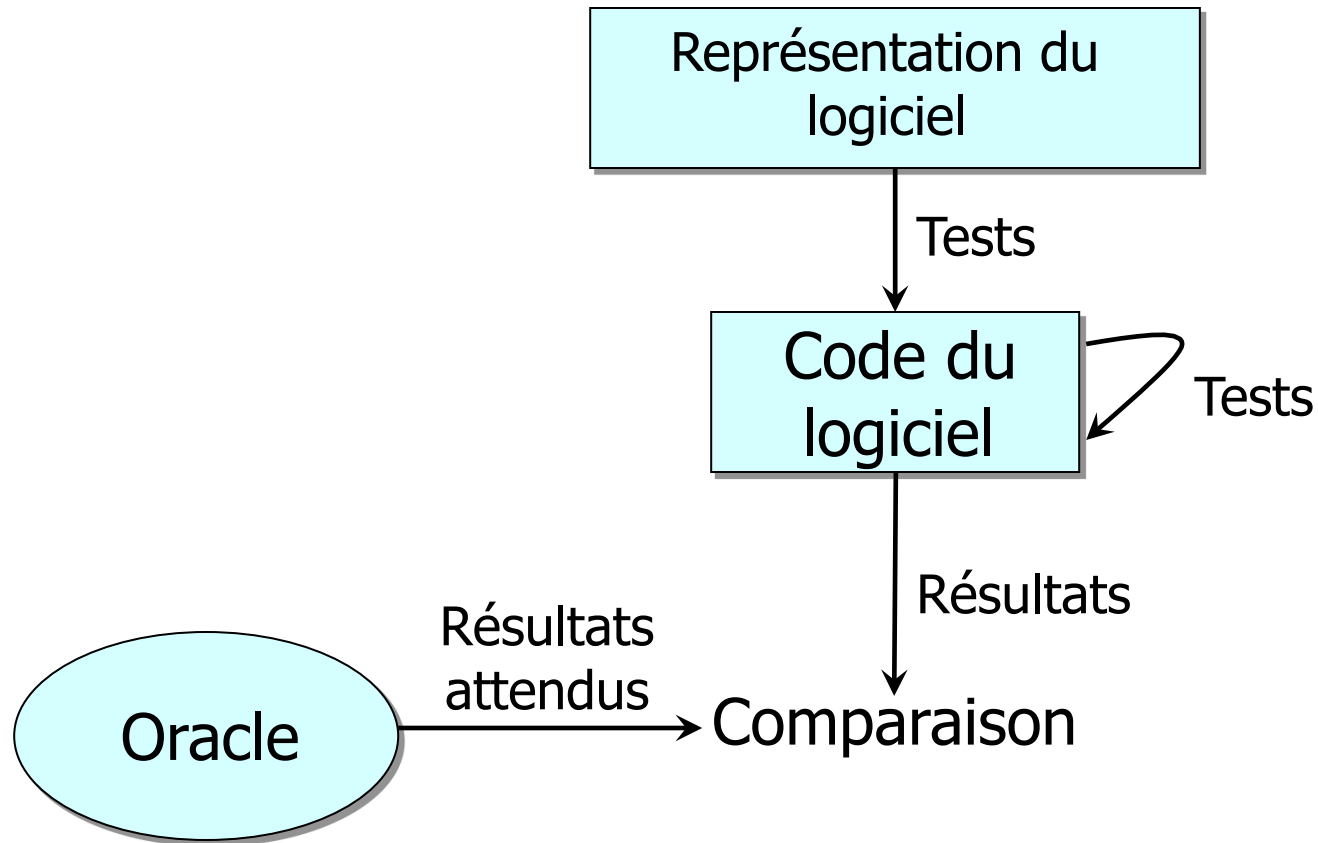
Sommaire des définitions



Buts du test

- ❑ Dijkstra, 1972 : “Le test de programme peut être utilisé pour montrer la présence de bugs, mais jamais pour montrer leur absence.”
- ❑ Aucune certitude absolue ne peut être déduite du test.
- ❑ Le test devrait être intégré avec d’autres activités de vérification, e.g., les inspections.
- ❑ But principal : démontrer que le logiciel a une *sûreté de fonctionnement suffisante*.

Vue d'ensemble du processus du test



Oracle : n'importe quel moyen permettant de prédire le résultat.

Tests de haute qualité

- ❑ Un nombre suffisant (le plus petit possible) de cas significatifs de test sont exécutés pour révéler des erreurs ou augmenter la confiance qu'on a dans le système.

- ❑ *Qualités recherchées*
 - *Efficace* pour découvrir des fautes
 - Aide à *localiser* les fautes à débbugger
 - *Répétable* de telle façon qu'une compréhension précise de la faute peut être déduite
 - *Automatisé* afin d'abaisser le coût et la durée
 - *Systématique* afin d'être prévisible

Principes fondamentaux

Propriété de la continuité

- ❑ Tester la capacité d'un pont à soutenir un certain poids.
- ❑ Si un pont peut soutenir un poids égal à $W1$, alors il soutiendra tout poids $W2 \leq W1$.
- ❑ La même règle ne peut pas être appliquée au logiciel ...
- ❑ La propriété de continuité, i.e., petites différences dans les conditions d'exploitation ne résultera pas en un comportement significativement différent -> totalement faux dans le cas d'un logiciel.

Test exhaustif

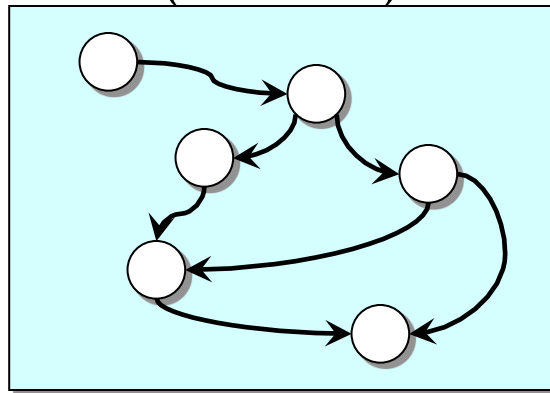
- ❑ Le test exhaustif, i.e., test d'un système logiciel utilisant toutes les données possibles, est la plupart du temps impossible.
- ❑ Exemples :
 - Un programme qui calcule la fonction factorielle ($n! = n.(n-1).(n-2)...1$).
 - ✓ Test exhaustif = exécuter le programme avec $n = 0, 1, 2, \dots, 100, \dots!$
 - Un compilateur (e.g., javac)
 - ✓ Test exhaustif = exécuter le compilateur de Java avec tous les programmes (Java) possibles (i.e., code source)!!!.
- ❑ Technique utilisée pour réduire le nombre de données :
 - Le critère de test groupe les éléments d'entrée en classes d'équivalence.
 - ✓ Une entrée est sélectionnée dans chaque classe (notion de couverture des entrées de test).

Test Intelligent

- ❑ Peu importe les moyens pour l'éviter, il reste des défauts dans tout composant logiciel
- ❑ Responsabilité des testeurs: concevoir des tests qui
 - révèlent des défauts
 - peuvent être utilisés pour évaluer divers attributs de qualité du logiciel (performance, fiabilité,...)
- ❑ Pour atteindre ces objectifs, le testeur doit **intelligemment choisir un sous-ensemble des entrées** de test possibles, ainsi que des **combinaisons de ces entrées**, de telle sorte que la **probabilité de révéler des défauts soit maximisée**, le tout en respectant les contraintes (budget, temps, ressources) du processus de test.

Stratégie de conception / Couverture des tests

Représentation du logiciel
(Modèle)



Critère associé

Le jeu de test doit couvrir
toutes les éventualités
dans le modèle

Données
de test

Représentation de

- la spécification \Rightarrow Test Boîte noire
- l'implémentation \Rightarrow Test Boîte blanche

Tests Boite Noire

- ❑ Aussi nommés "test basés sur les spécifications" ou "tests fonctionnels"
- ❑ le logiciel à tester est considéré comme une boîte opaque
- ❑ le testeur sait **ce que fait** le logiciel, mais pas **comment**
- ❑ applicable à tous les niveaux (fonction, module, système complet)
- ❑ sources d'information pour la conception des tests:
 - spécifications (formelles ou non)
 - ensemble bien défini de pré- et post-conditions
- ❑ particulièrement utile pour **révéler des défauts au niveau des spécifications**

Tests Boite Noire - Exemple

❑ **Spécification du calcul de la factorielle d'un nombre:**

Si la valeur d'entrée n est < 0 , alors un message d'erreur approprié devrait être imprimé. Si $0 < n < 20$, alors la valeur exacte de $n!$ devrait être imprimée. Si $20 < n < 200$, alors une valeur approximative de $n!$ devrait être imprimée en format virgule flottante, e.g., utilisant une méthode approximative de calcul numérique. L'erreur admissible est 0.1% de la valeur exacte. Finalement, si $n \geq 200$, l'entrée peut être rejetée en imprimant le message d'erreur approximatif.

- ❑ À cause des variations attendues en comportement, il est naturel de diviser le domaine d'entrée en classes $\{n < 0\}$, $\{0 < n < 20\}$, $\{20 < n < 200\}$, $\{n \geq 200\}$. Nous pouvons utiliser un ou plusieurs cas de test de chacune des classes dans chaque jeu de test. Les résultats corrects d'un tel jeu de test supportent l'assertion que le programme se comportera correctement pour n'importe quelle autre valeur, mais il n'y a pas de garantie!

Tests Boite Blanche

- ❑ aussi nommés "boite transparente" ou "boite de verre"
- ❑ concentrés sur la structure interne du code
 - le code (ou un pseudo-code "fidèle") doit être disponible au testeur
- ❑ cas de tests conçus pour exercer certaines structures spécifiques
- ❑ plus longs à concevoir, coder, exécuter et analyser les résultats, donc typiquement appliqués à des "petits" éléments logiciels
- ❑ utiles pour révéler des défauts liés à la **conception**, ou au **code (contrôle, logique, séquences, initialisation, flux de données)**

Tests Boite Blanche - Exemple

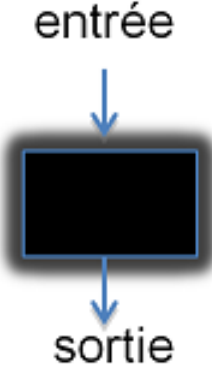

```
if x > y then
    Max := x;
else
    Max := x ;    // faute!
end if;
```

{x=3, y=2; x=2, y=3} peut détecter l'erreur, plus de "couverture"

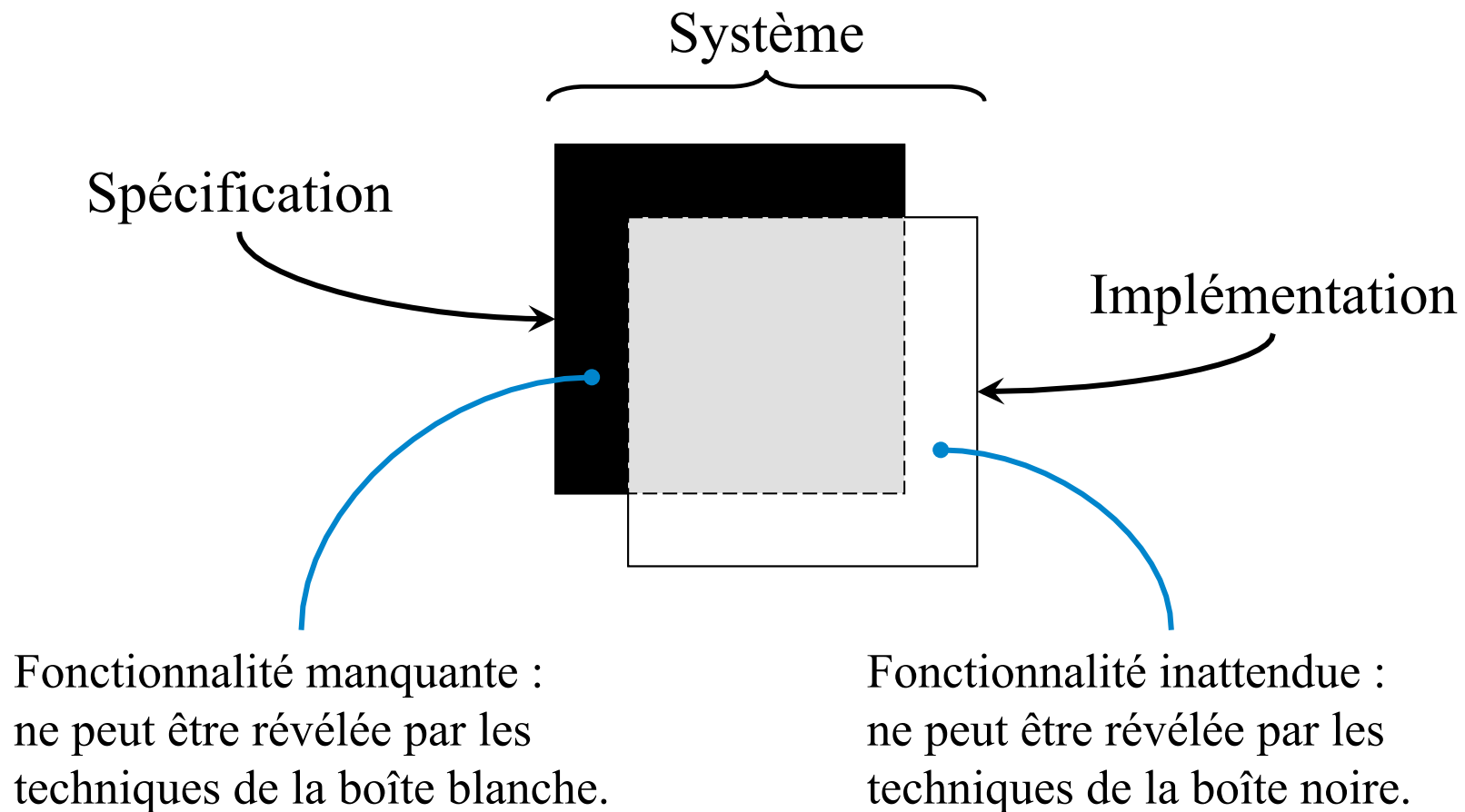
{x=3, y=2; x=4, y=3; x=5, y=1} est plus étendu mais ne peut pas la détecter.

- ❑ Le critère de test groupe les domaines d'entrée en classes d'équivalence (ici, les chemins dans le graphe de flot de contrôle).
- ❑ La couverture complète essaie d'exécuter les cas de test de chacune des classes.

Test boîte noire et test boîte blanche

Stratégie de test	Point de vue du testeur	Sources	Techniques/Approche
Boîte noire		<ul style="list-style-type: none">-Exigences-Spécifications-Expertise du domaine-Analyse des défauts-Données	<ul style="list-style-type: none">-Exploratoire-Classes d'équivalence-Analyse des valeurs frontalières-Transition d'états-Analyse cause à effet-Cas d'utilisation-Table de décision
Boîte blanche		<ul style="list-style-type: none">-Architecture-Conception-Graphes du code source-Structures internes-Complexité	<ul style="list-style-type: none">-Test d'énoncé-Test de branche-Test de chemin-Test de flot de données-Test de structure

Test boîte noire vs test boîte blanche



Test boîte blanche vs test boîte noire

■ Boîte noire

- + Vérifier la conformité avec les spécifications.
- + Il s'adapte aux niveaux de granularité (différentes techniques à différents niveaux de granularité).
- Il dépend de la notation des spécifications et de leur degré de détail.
- On ne sait pas jusqu'à quel point le système a été testé.
- Quoi faire si le logiciel effectue des tâches non spécifiées, indésirables?

□ Boîte blanche

- + Il permet d'être certain au sujet de la couverture du test.
- + Il est basé sur la couverture du flot de contrôle ou de données.
- Il ne s'adapte aux niveaux de granularité (le plus souvent applicable à l'unité et aux niveaux du test d'intégration).
- À la différence de la technique de la boîte noire, il ne peut pas révéler les fonctionnalités manquantes (partie de la spécification qui n'est pas implémentée).

Fondements théoriques

Critère d'adéquation

- ❑ Étant donné un critère C pour un modèle M
 - Le *ratio* de la couverture de l'ensemble de tests T est la proportion des éléments de M définis par C qui auront été couverts par l'ensemble de tests T.
 - Un ensemble de tests T est dit adéquat pour C, ou simplement C-adéquat, quand le ratio de la couverture atteint 100% pour un critère C.
- ❑ Exemple :
 - M est le graphe de flot de contrôle d'une fonction.
 - C est l'ensemble de tous les arcs dans le graphe.

Hiérarchie des critères

La relation subsomption (inclusion) entre les critères associés au même modèle.

- ❑ Étant donné un modèle M et deux critères C1 et C2 pour ce modèle :
 - C1 inclut C2 si n'importe quel ensemble de tests C1-adéquat est aussi C2-adéquat.
- ❑ Exemple : Considérons les critères « tous les arcs » et « tous les chemins » pour des graphes de flux de contrôle, « tous les chemins » inclut « tous les arcs ».
- ❑ Si C1 inclut C2, nous supposons :
 - Satisfaire C1 est plus coûteux (e.g., # de cas de tests) que satisfaire C2.
 - C1 permet la détection de plus d'anomalies que C2.

Fondements théoriques

- ❑ Soit P un programme.
- ❑ Soit D le domaine des valeurs d'entrée.
- ❑ OR dénote les besoins en valeurs de sorties (Oracle).
- ❑ P est dit *correct* pour $d \in D$ si $P(d)$ satisfait OR – sinon, nous avons une défaillance.
- ❑ Un cas de tests est un élément d de D .
- ❑ Un ensemble (jeu) de tests T est un ensemble fini de cas de test – un sous-ensemble fini de D .

Fondements théoriques II

- ❑ Un ensemble de tests T est dit *idéal* si, chaque fois que P est incorrect, il existe $d \in T$ pour laquelle P est incorrect (pour d).
- ❑ Si T est un ensemble idéal de tests et T est réussi pour P , alors P est correct.
- ❑ Un *critère d'adéquation* C est un sous-ensemble de P_D , l'ensemble de tous les sous-ensembles finis de D .
- ❑ T satisfait C s'il appartient à C (adéquat).

Critère de sélection de Test

- ❑ C est *cohérent* si, pour n'importe quelle paire d'ensembles de tests T1 et T2 qui satisfont C, T1 est réussi si, et seulement, si T2 est réussi.
- ❑ C est *complet* si, chaque fois que P est incorrect, il y a un ensemble de tests non réussis qui satisfait C.
- ❑ Si C est cohérent *et* complet, tout ensemble de tests T satisfaisant C est *idéal* et pourrait être utilisé pour décider que P est correct.
- ❑ Le problème est qu'il n'est pas possible de dériver des algorithmes qui aident à déterminer si un critère, un ensemble de tests ou un programme a l'une des propriétés mentionnées ... ce sont des problèmes indécidables.

Principe empirique de test

- ❑ Impossible de déterminer les critères de cohérence et de complétude d'une manière théorique.
- ❑ Un test exhaustif ne peut pas être effectué en pratique.
- ❑ Par conséquence, nous avons besoin de *stratégies* de tests qui ont été investiguées empiriquement.
- ❑ Un cas significatif de test est un cas de test avec un grand potentiel de détection d'erreur – cela augmente notre confiance dans l'exactitude d'un programme (« correctness »).
- ❑ L'objectif est d'exécuter un nombre suffisant de cas significatifs de test – ce nombre pourrait être le plus petit possible.

Aspects pratiques

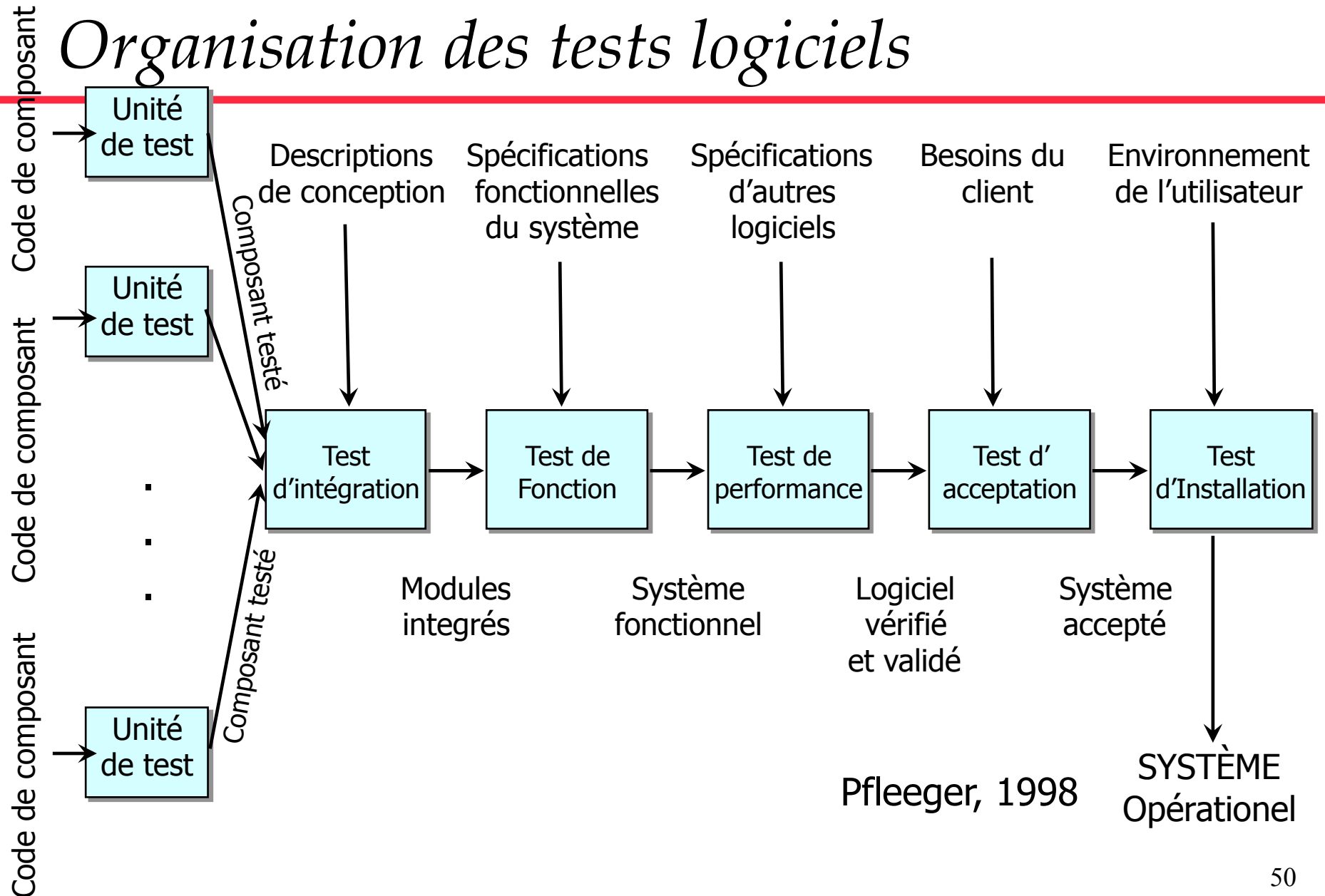
Plusieurs causes de défaillances

- ❑ La spécification peut être incorrecte ou avoir un besoin manquant.
- ❑ La spécification peut contenir un besoin qui est impossible à implémenter étant donné le logiciel et le matériel prescrits.
- ❑ La conception du système peut contenir une défaillance.
- ❑ Le code du programme peut être inexact.

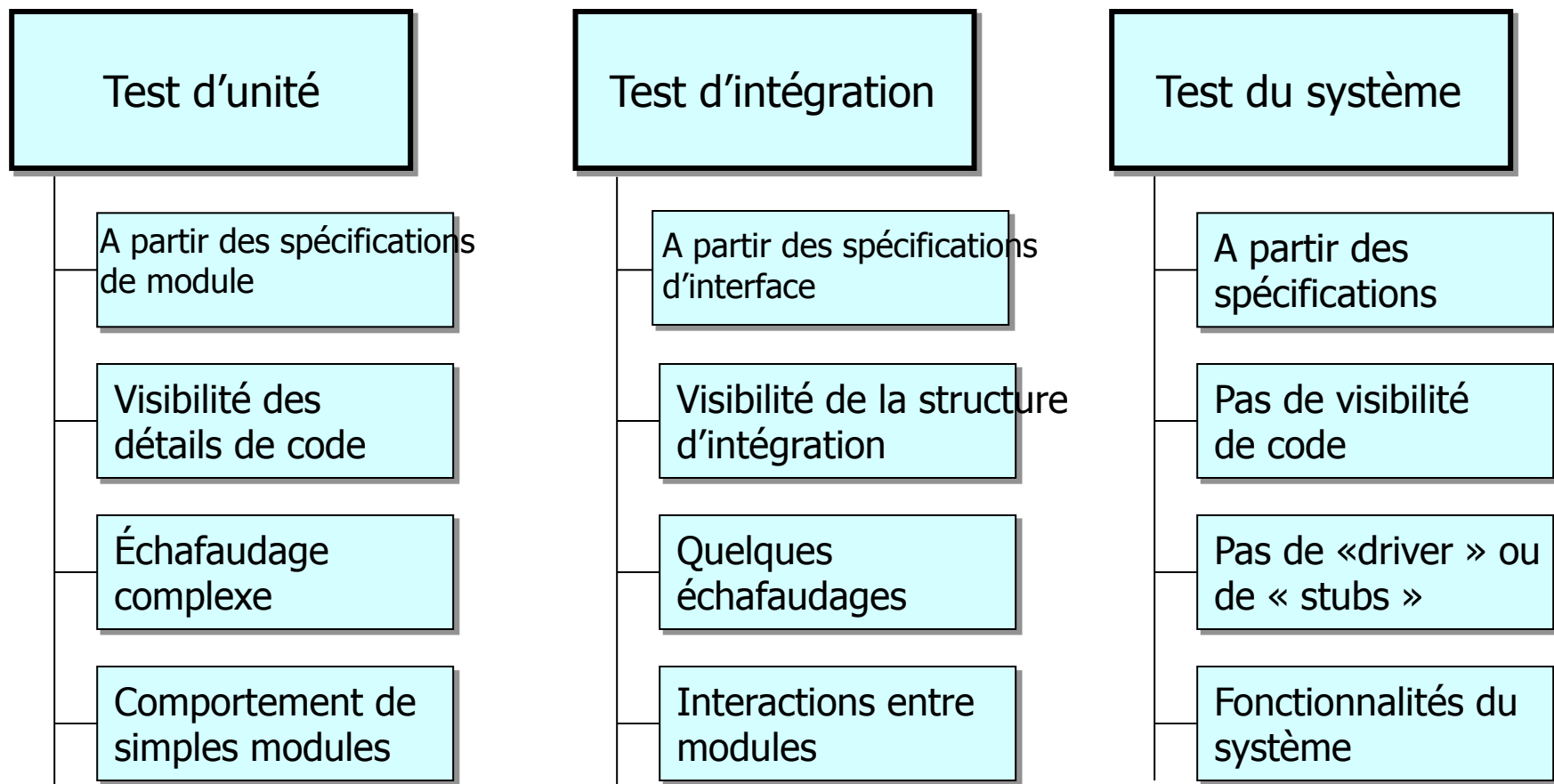
Organisation de test

- ❑ Il peut y avoir différentes causes potentielles de défaillance. Dans les gros systèmes, le test implique plusieurs étapes :
 - Module, composant ou unité de test
 - Test d'intégration
 - Test de fonction
 - Test de performance
 - Test d'acceptation
 - Test d'installation

Organisation des tests logiciels



Différences entre les activités de tests



Pezze and Young, 1998

Test d'intégration

L'intégration de composants bien testés peut mener à une défaillance due à :

- ❑ Mauvaise utilisation des interfaces (mauvaises spécifications d'interface / implémentation) ;
- ❑ hypothèse incorrecte sur le comportement/état de modules reliés (mauvaise spécification de fonctionnalité / implémentation), e.g., supposition incorrecte concernant la valeur de retour.
- ❑ utilisation de drivers / stubs faibles : un module peut se comporter correctement avec des drivers/stubs (simples), mais peut avoir des défaillances quand il est intégré avec des modules réels (complexes).

Test de système vs. test d'acceptation

❑ Test de système

- Le logiciel est comparé avec les spécifications des besoins (vérification).
- Toujours effectué par des développeurs qui connaissent le système.

❑ Test d'acceptation

- Le logiciel est comparé avec les spécifications de l'utilisateur final (validation).
- Toujours effectué par le client (acheteur) qui connaît l'environnement où le système est utilisé.
- Parfois on distingue entre « α - β - testing » pour les produits à usage général.

Test durant le cycle de vie

- ❑ Beaucoup d'artefacts de développement durant le cycle de vie fournissent une source riche de données de test.
- ❑ L'identification anticipée des besoins de test et des cas de tests aide à réduire la durée de développement.
- ❑ Ils peuvent aider à faire connaître les fautes.
- ❑ Cela peut aussi aider à identifier tôt la faible testabilité des spécifications ou de la conception.

Organisation du cycle de vie

- ❑ besoins => Test d'acceptation
- ❑ Spécifications/Analyses => Test de système
- ❑ Conception => Test d'intégration
- ❑ Diagramme de classe, méthodes, pré- et post-conditions, structure => test de classe.

Exemple : besoins du système

- ❑ Les erreurs à cette étape auront des effets dévastateurs comme toutes les autres activités qui en sont dépendantes.
- ❑ Langage naturel : flexible mais ambigu, testabilité faible.
- ❑ Est-il possible développer un test pour vérifier si les besoins ont été satisfaits ?
 - ◆ E.g., non testable : le système devrait être convivial, le temps de réponse devrait être raisonnable.
- ❑ Le développement anticipé des tests d'acceptation à partir des besoins nous permet d'évaluer si ces tests sont testables et de durée raccourcie.
 - ◆ E.g., testable : le temps de réponse est moins de 1.5 seconde pour 95% du temps moyen du chargement du système.

Exemple : besoins du système

- ❑ Le niveau le plus bas du test des besoins consiste à générer les données de test pour chaque besoin au moins une fois.
- ❑ Mais nous voulons utiliser des techniques qui sont un peu plus exigeantes : limites et interactions des besoins.
- ❑ Les techniques de test fonctionnel (boîte noire) peuvent être appliquées.
- ❑ Partitionnement d'équivalence, analyses des limites, partition par catégorie, tables de décision, graphe de cause à effet.
- ❑ Les besoins de test d'acceptation sont identifiés.

Activités de test

- ♦ **Établir les objectifs des tests**
- ♦ **Concevoir les cas de tests (jeu de test)**
- ♦ **Écrire les cas de tests**
- ♦ **Tester des cas de tests**
- ♦ **Exécuter les tests**
- ♦ **Évaluer les résultats de tests**
- ♦ **Comprendre la cause des défaillances**
- ♦ **Faire des tests de régression**

Les activités de test AVANT codage

- ❑ Le test est une activité consommatrice du temps.
- ❑ Développer une stratégie de test et identifier les besoins de test représentent une partie substantielle du test.
- ❑ Planifier est essentiel.
- ❑ Les activités de test font subir une énorme pression si elles sont exécutées vers la fin du projet.
- ❑ Afin de raccourcir le délai de livraison (vers le marché) et d'assurer un certain niveau de qualité, beaucoup d'activités d'AQ (incluant le test) doivent être réalisées tôt dans le cycle de vie du développement.

Rôle du testeur

- ❑ Pour développer des tests efficaces, il faut avoir :
 - ✓ une compréhension détaillée du système ;
 - ✓ une bonne connaissance des techniques de test ;
 - ✓ les compétences nécessaires pour appliquer ces techniques d'une manière efficace.
- ❑ Le test est souvent perçu comme une activité destructrice:
 - révéler des défauts
 - trouver des points faibles ou des comportements incohérents
- ❑ Le programmeur restera souvent fidèle à l'ensemble des données qui font fonctionner le programme.
- ❑ Il est préférable que les tests soient faits par des testeurs hiérarchiquement indépendants des développeurs.
- ❑ **Souvent, un programme ne fonctionne pas quand il est essayé par quelqu'un d'autre.**