CONDUITE DE PROJET

INTEGRATION, TESTS ET VALIDATION

Définitions

Tester, c'est exécuter un programme pour y trouver des erreurs.

Myers - The Art of Software Testing - 1979

Autres définitions

- Le test est l'exécution ou l'évaluation d'un système ou d'un composant, par des moyens automatiques ou manuels, pour vérifier qu'il répond à ses spécifications ou identifier les différences entre les résultats attendus et les résultats obtenus.

  IEEE - Standard Glossary of Software Engineering Terminology
  IEEE STD 729 - 1983

- Technique de contrôle consistant à s'assurer, au moyen de l'exécution d'un programme que son comportement est conforme à des données pré-établies.

  AFCIQ - Norme expérimentale de terminologie - 1985

- Technique de contrôle consistant à s'assurer, grâce à l'exécution d'un constituant logiciel, que son comportement est conforme à un comportement de référence préalablement formalisé dans la procédure de test.

  DGA - Méthodologie de développement des logiciels intégrés
  GAM-T17 / V2 - 1988

Vocabulaire

Le test permet de détecter une ANOMALIE

due à un DEFAUT du logiciel

lui même du a une ERREUR du programmeur.

- L'anomalie fait l'objet d'un rapport.
- Le défaut doit être corrigé.
- Il est préférable d'éviter l'erreur du programmeur.

Efficacité comparée des tests et de l'analyse de code

Une simple lecture du code permet d'éliminer jusqu'à 80% des anomalies (avant les tests)

TECHNIQUES:

- INSPECTION DE CODE:

  Examen méthodique des programmes (sélection/échantillon)

- WALKTHROUGH

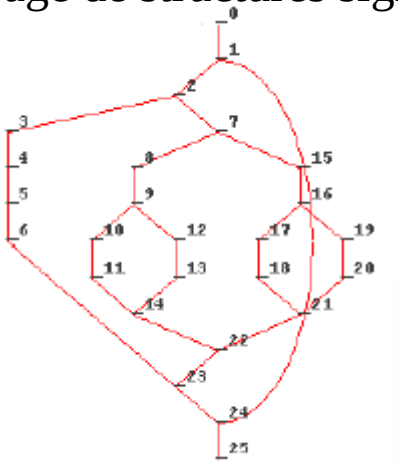
  Réunion de lecture en commun de sections de code critiques

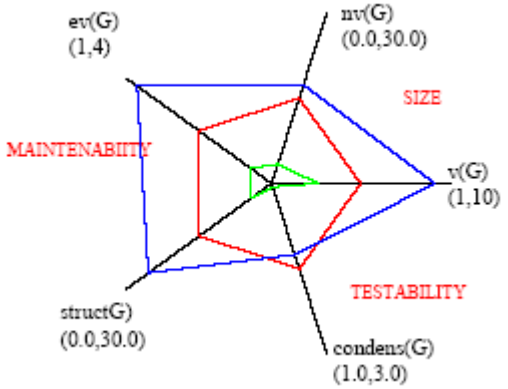
- ANALYSE STATIQUE

  Parcours automatique du code pour effectuer des mesures

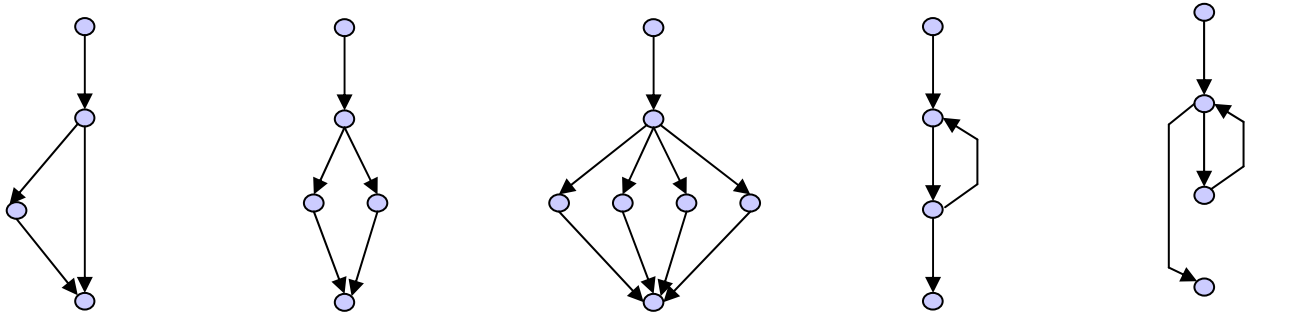
Analyse statique de code

Repérage de structures significatives/ Comptages → Défaut/Mesure de complexité





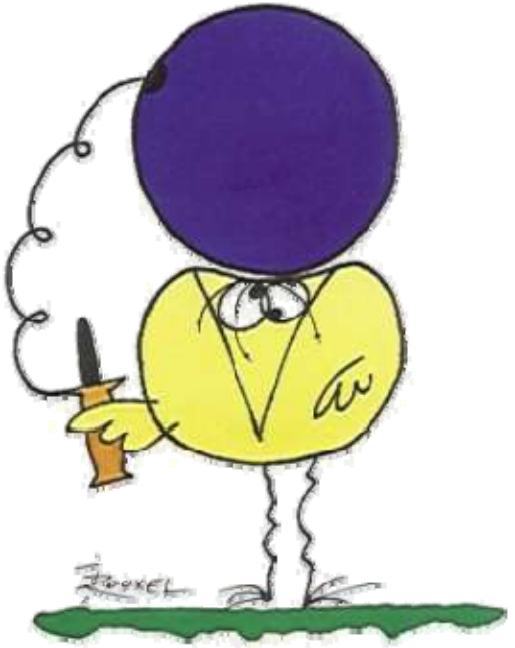
Exemple: Nombre cyclomatique = nombre d'arcs – nombre de sommets + 2



`If/then` `If/then/else` `Switch / case` `Do while` `While` ou `for`

Le mauvais testeur

- « C'est la dernière compilation et après c'est fini ».
- Ne repasse pas le test qui lui a permis de découvrir une erreur après la correction.
- Ne sait pas trop si le résultat qu'il obtient est correct ou non.
- Ne peut exprimer clairement ce qui est et n'est pas testé quand il livre un produit.
- Ne sait pas re-tester automatiquement un logiciel qu'il a déjà testé une fois.
- Pour tester son module, lance le debugger et exécute pas à pas (avec 2 ou 3 jeux d'essai), pour regarder si « tout est bon ».



Influence des tests mal fait sur les coûts et délais

LES TESTS MAL FAITS NE SONT PAS MAITRISABLES

- **On ne sait pas apprécier l'avancement réel (Qu'a-t-on fait? Que reste-t-il à faire?).**
- **On ne sait pas distribuer la charge de test.**
- **On ne sait pas quand les tests sont terminés.**
- **On ne sait pas rejouer les tests déjà exécutés.**
  - **Le coût du logiciel et le délai de développement sont imprévisibles.**
  - **La maintenance est coûteuse.**
  - **Les tests sont finalement négligés pour pouvoir respecter le planning général.**

Influence des tests mal faits sur la Qualité

LES TESTS MAL FAITS ENTRAINENT LA DEGRADATION…

- **DU LOGICIEL**
  - **Sa structure se complexifie car les corrections d'anomalies sont faites de façon désorganisée (verrues).**
  - **Les règles de codage ne sont plus respectées.**
  - **La documentation ne suit plus les évolutions.**
  - **Des impasses sont faites au niveau des exigences initiales.**
- **DES RELATIONS**
  - **dans l'équipe.**
  - **avec le client .**

Validation et vérification

- VALIDATION
  - Activités permettant de s'assurer qu'un produit correspond aux besoins.
  - Objectif : **"FAIRE LE BON PRODUIT".**

- VERIFICATION
  - Activités permettant de s'assurer qu'un produit est réalisé conformément à ses spécifications.
  - Objectif : **"BIEN FAIRE LE PRODUIT".**

Testabilité d ’un logiciel

CAPACITE A EXERCER UN LOGICIEL AFIN D’EN VERIFIER LE BON FONCTIONNEMENT

Un logiciel est **TESTABLE** si il est

- **REPETITIF:** Il donne toujours le même résultat dans des conditions de sollicitation identiques.
- **ACCESSIBLE:** Il peut être mis en œuvre sans excès de difficulté.
- **AUTODESCRIPTIF:** Il contient en lui-même toutes les informations nécessaires à sa compréhension.
- **STRUCTURE:** Son organisation interne est modulaire.

B.W.Boehm - 1980

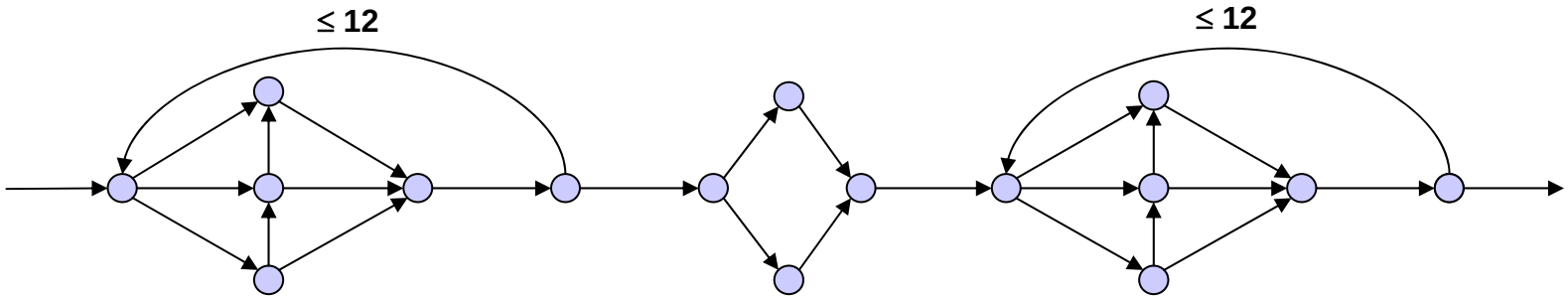
Peut-on tester tous les cas possibles?

1. EXHAUSTIVITE PAR RAPPORT AU DOMAINE D'ENTREE

- Combinatoire très grande voire infinie
- Combinatoire encore plus importante si le logiciel utilise en entrée des résultats établis précédemment pour traiter les données

  Exemples:
  - Tests d'un compilateur
  - Tests d'un superviseur de processus

2. EXHAUSTIVITE DU TEST PAR RAPPORT A L'OBJET TESTE



N^{bre} de chemins = $(6 + 6^2 + 6^3 + ... + 6^{12}) \times 2 \times (6 + 6^2 + 6^3 + ... + 6^{12}) \approx 10^{19}$

Tests de tous les chemins ≈ 400 ans de travail à raison d'un test toutes les nano-secondes

Efficacité et rendement des tests

- **L'exhaustivité des tests est rarement possible.**
- **L'exhaustivité des tests n'est pas toujours utile (le nombre de situations réelles est souvent inférieur au nombre de situations théoriquement possibles).**

IL FAUT S'ATTACHER A LA REPRESENTATIVITE DES TESTS PAR RAPPORT AUX SITUATIONS THEORIQUEMENT POSSIBLES.

Cette représentativité peut se mesurer afin de connaître la complétude des tests.
Cette mesure s'appelle **LA COUVERTURE DES TESTS.**

Effort de test - Distribution par phase

	Spécifications & Conception	*Codage & Mise au Point*	*Tests & Intégration*
Commande/Contrôle	35%-46%	17%-20%	48%-34%
Spatial	46%	20%	34%
Système/OS	34%	20%	46%
Scientifique	44%	26%	30%
Gestion	44%	28%	28%
TOTAL	**40%**	**20%**	**40%**

Tests unitaires

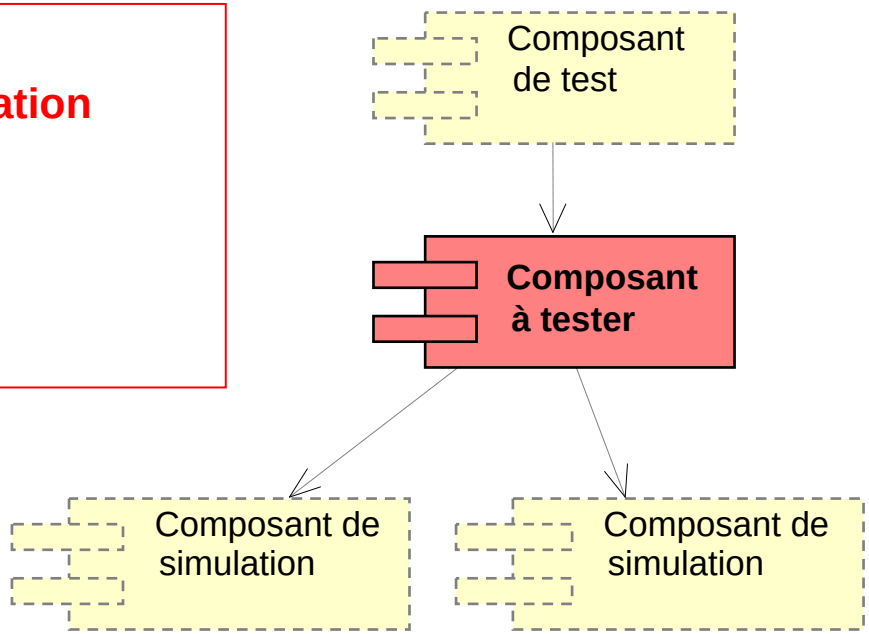
- **Ce sont les tests d'un COMPOSANT ISOLE à l'issue de sa production.**
- **Ils peuvent être exécutés sur une machine de développement avec simulation de l'environnement cible.**

Il doivent

- toujours être **exécutés avant intégration**
- toujours être **planifiés**
- si possible, être **automatisés**
- être **"tracés"** (passage + résultats)

Ils peuvent être

- **FONCTIONNELS** (boite noire)
- **STRUCTURELS** (boite blanche).



Démarches possibles

- **PROGRAMMATION DEFENSIVE**
  - Instrumentation automatique du code
  - Vérification systématique de clauses d'invariant, pré-conditions, post-conditions.
  - Génération systématique d'une version de mise au point

- **TESTS FONCTIONNELS SUR CHAQUE COMPOSANT**
  - Permet de détecter les erreurs ou "les oublis" de codage
  - Sous la responsabilité du réalisateur
  - Formalisation minimum des procédures et comptes-rendus

- **TESTS STRUCTURELS SUR LES FONCTIONS "ESSENTIELLES"**
  - Réalisés en intégration sur un ensemble cohérent de composants
  - Permet de vérifier l'instanciation des classes et le comportement des objets
  - Permet de vérifier le bon fonctionnement des algorithmes
  - Permet de mesurer la couverture de test

Mesure de couverture

Objectif	*Nombre d'objets*	*Nombre minimal de jeux d'essai*	*Efficacité*
Toutes les instructions	Nombre d'instructions du programme	?	insuffisant
Tous les segments	Nombre d'arcs du graphe de programme	= n^{bre} cyclomatique = n^{bre} arcs – n^{bre} sommets +2	Le minimum (implique toutes les instructions)
Tous les chemins	A calculer d'après le graphe Séquence = A × B Alternative = A + B Itération = A × e^{n+1}	= n^{bre} chemins	D'ordre 2 sinon trop long (implique tous les segments des chemins faisables)
Tous les prédicats	?	?	Difficile à mettre en œuvre (implique tous les segments)

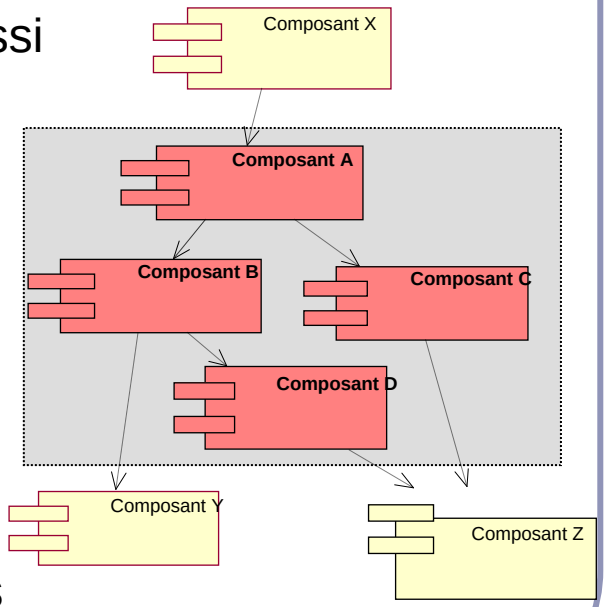
Mesure de couverture - Exemple

```
Sorting order:                                                                  _____FUNCTIONS_____  ________LINES________
Unused lines                                                          Runs  Calls  unused  used  used%  unused  used  used%
   Total Coverage                                                                  5269  1444   21%   13450  1938   12%
      /home/brevel/modele/S_DONNEES_PLAN_VOL/code/CHECK_OBJ/                         737    77    9%    3581   259    6%
      /home/brevel/modele/S_DONNEES_CONTRAINTES/code/CHECK_OBJ/                      218     3    1%    2724     0    0%
      /home/brevel/modele/S_DONNEES_ENVIRONNEMENT/code/CHECK_OBJ/                    320     3    0%    1859     0    0%
      /home/brevel/modele/S_IHM_NOYAU/code/CHECK_OBJ/                                79    32   28%     369    85   18%
      /home/brevel/modele/S_DIALOGUE/code/CHECK_OBJ/                                 61    26   29%     140    63   31%
      /home/brevel/modele/S_RESSOURCE_POP/code/CHECK_OBJ                             60   116   65%     126   316   71%
        POP.crc                                                                       2     3   22   88%      68   114   62%
            POP::~POP(void)                                                           6         X            65    88   57%
            POP::_numero_POP(void) const                                              0         X             1     0    0%
            POP::accessibilite_appli(int)                                             0         X             1     0    0%
            POP::maj_contexte_distant(char,unsigned int)                              0         X             1     0    0%
            POP::POP(char, unsigned int)                                              6         X             0     1  100%
            POP::_contexte_travail(int) const                                       188         X             0     1  100%
            POP::_invariant(void) const                                             126         X             0     2  100%
            POP::card_configuration(void) const                                     252         X             0     2  100%
            POP::card_contexte_travail(void) const                                  357         X             0     2  100%
            POP::card_distant(void) const                                           305         X             0     2  100%
            POP::declarer_poste_distant(char, unsigned int)                           4         X             0     1  100%
            POP::maj_contexte_hote(unsigned int, unsigned int, int)                   5         X             0     1  100%
            POP::premiere_appli_dispo(unsigned int)                                   1         X             0     1  100%
            POP::tester_disponibilite(int, unsigned int)                             47         X             0     2  100%
            set_of_CONTEXTE_DE_TRAVAIL::append(const CONTEXTE_DE_TRAVAIL*)            7         X             0     1  100%
            set_of_CONTEXTE_DE_TRAVAIL::card(void) const                            357         X             0     1  100%
            set_of_CONTEXTE_DE_TRAVAIL::erase(const int, const int)                   1         X             0     1  100%
            set_of_CONTEXTE_DE_TRAVAIL::operator ()(const int) const                224         X             0     1  100%
            set_of_CONTEXTE_DE_TRAVAIL::~set_of_CONTEXTE_DE_TRAVAIL(void)             6         X             0     1  100%
            set_of_POP::card(void) const                                            305         X             0     1  100%
            set_of_POP::operator ()(const int) const                                286         X             0     1  100%
            set_of_POP::operator <<(const POP*)                                       4         X             0     1  100%
            set_of_POP::set_of_POP(const int, const int, const int)                   6         X             0     1  100%
            set_of_POP::~set_of_POP(void)                                             6         X             0     1  100%
      REF_VOL.crc                                                             2             4   28   87%      40    49   55%
      CONFIGURATION.crc                                                       2             4   11   73%       4    41   91%
      BD_VOL.crc                                                              2             2    8   80%       2     8   80%
      ENREGISTREMENT.crc                                                      2             1    1   50%       1     1   50%
```

Intégration

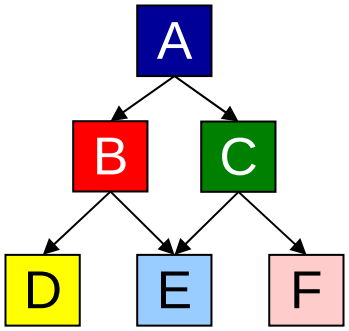
CONSTRUCTION PROGRESSIVE DU LOGICIEL A PARTIR DE COMPOSANTS TESTES UNITAIREMENT

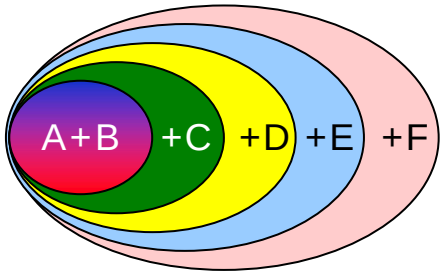
- Pour vérifier les choix architecturaux (minimisation des risques)
- Pour vérifier que le composant intégré “fonctionne aussi bien dans son environnement que isolément”
- Pour exercer tous les points d’entrée du composant
- Pour exercer tous les points d’appel des composants
- Pour les appels concurrents d’un même composant
- Pour vérifier le bon emploi des données partagées
- Pour vérifier les mécanismes de synchronisation
- Pour vérifier la conformité des résultats intermédiaires
- Pour identifier les composants morts



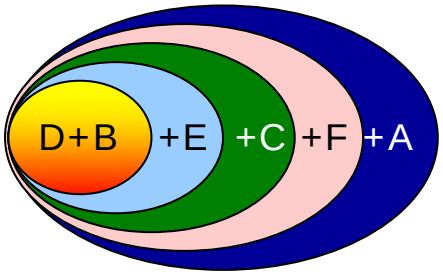
Stratégies d'intégration

- **INTEGRATION MASSIVE (BIG-BANG)**
  - Assemblage en 1 seule étape de tous les composants
  - Suicidaire sur les projets de plus de 5 composants
- **INTEGRATION PAR AGREGATS**
  - Assemblage des composants en sous-ensembles puis assemblage des sous-ensembles
  - Nécessite la conception des sous-ensembles (Sous-systèmes, Lots, Packages,...)
- **INTEGRATION PAR INCREMENTS**
  - Ajout d'un ou plusieurs composants à l'assemblage existant (base d'intégration)
  - Généralement bien adaptée aux architectures logicielles
  - Peut être réalisée suivant différentes approches

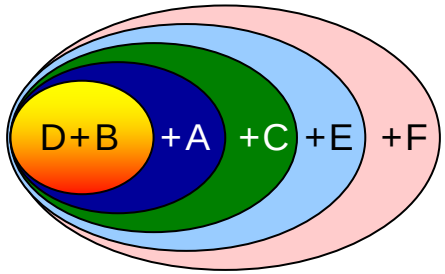




DESCENDANTE



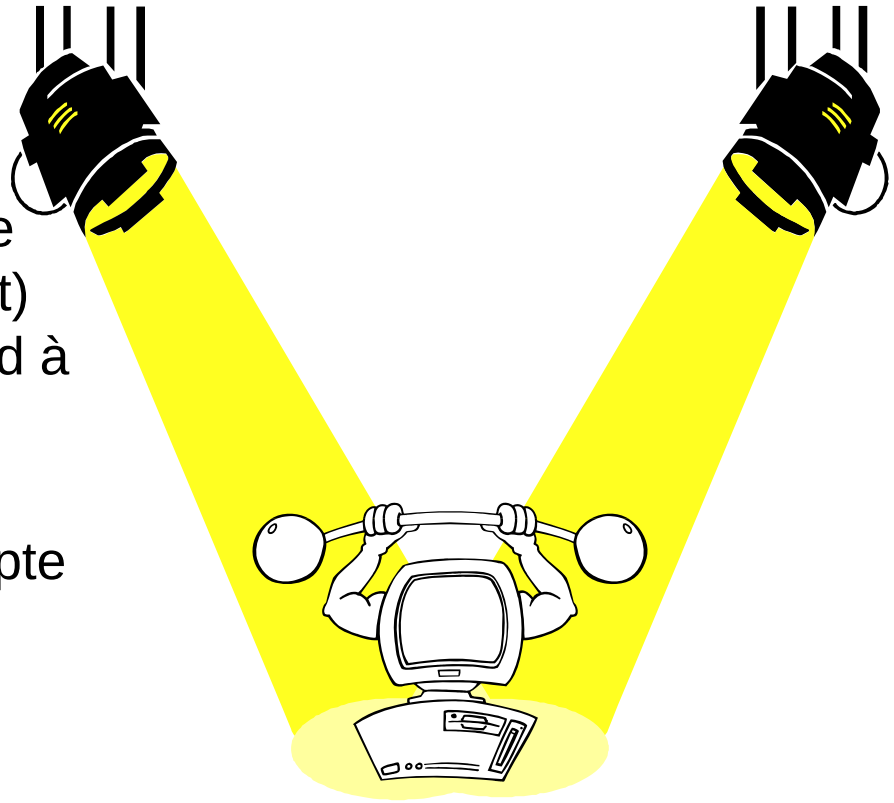
ASCENDANTE



MIXTE

Validation du logiciel

- La validation a pour but
  - de vérifier que le logiciel est conforme à sa spécification.
  - de démontrer au commanditaire (client, utilisateur, chef de projet) que le logiciel développé répond à ses besoins.
- La validation considère le logiciel comme un tout et ne tient pas compte de sa structure interne.
- La validation est parfois appelée «recette provisoire».



La validation par étapes

1. INSTALLATION
- Présence et conformité du produit
- Procédure d'installation

2. INTERFACES
- Présence et conformité de toutes les interfaces

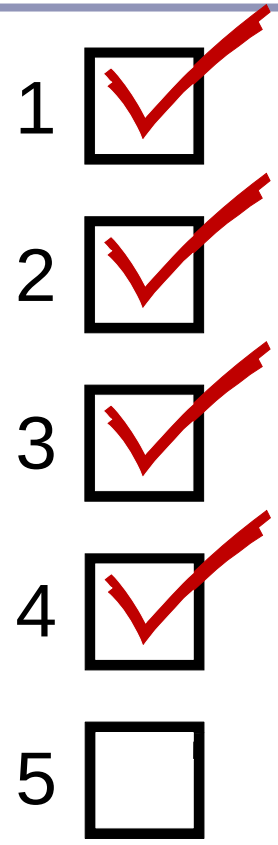
3. DIALOGUE
- Ergonomie
- Enchaînements corrects
- Résistance aux erreurs

4. FONCTIONS
- Conformité aux spécifications fonctionnelles
- Enchaînement des fonctions

5. PERFORMANCES
- Temps, Espace, Débit, Capacité

6. ROBUSTESSE
- Résistance aux défaut des supports
- Modes dégradés

7. SECURITE
- Sécurité d'accès, de fonctionnement, intégrité des données

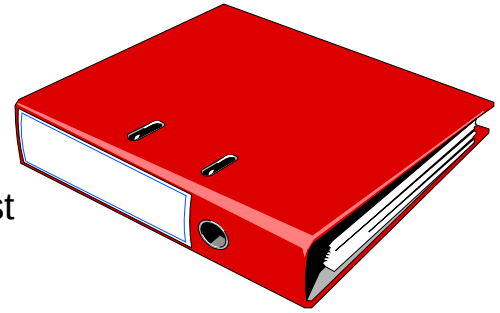


Le plan de validation

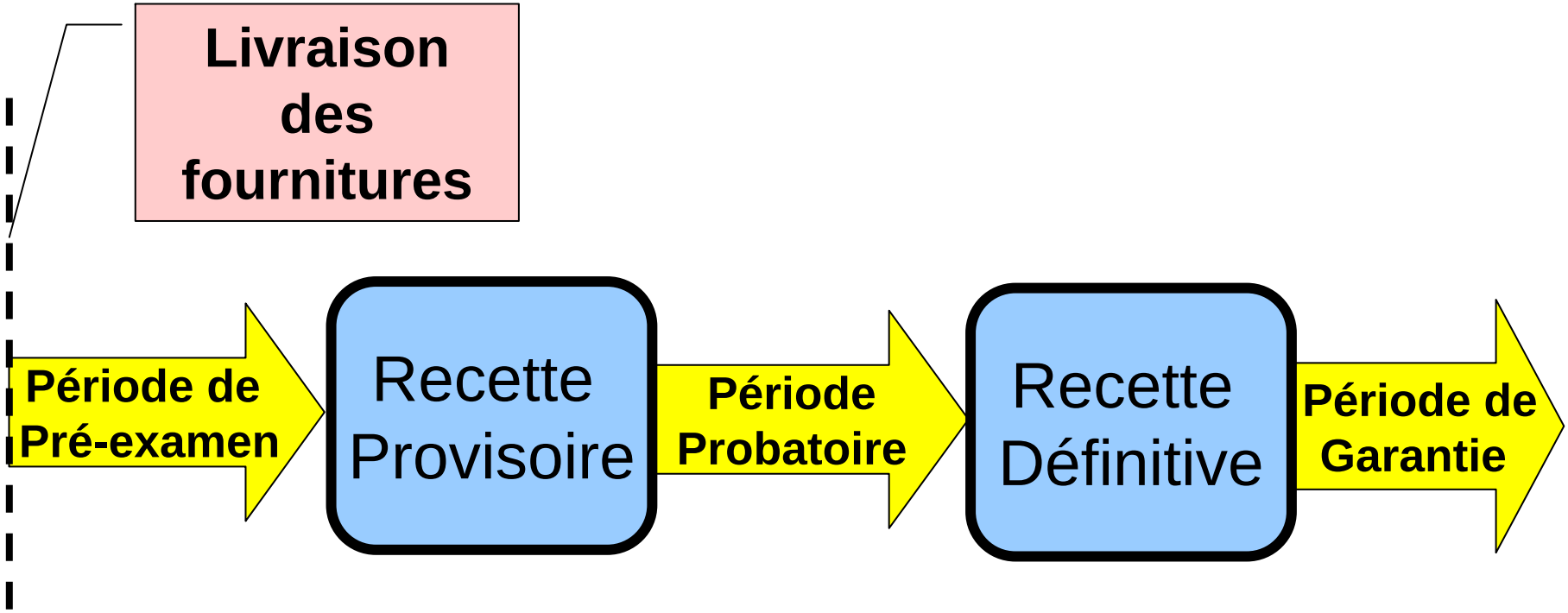
1. Objet
2. Documents de référence
3. Terminologie et sigles utilisés
4. Dispositions générales
- 4.1. Démarche d'élaboration des procédures de test
- 4.2. Planification de la validation
- 4.3. Fournitures et documentation
- 4.4. Organisation des essais
- 4.5. Responsabilités

5. Dispositions détaillées
- 5.1. Etape 1 (titre de l'étape)
  - 5.1.1. Liste des objectifs de test
  - 5.1.2. Scénarios de tests
  - 5.1.3. Ressources nécessaires
- 5.2. Etape 2 (titre de l'étape)
  - .....
- 5.n. Etape n

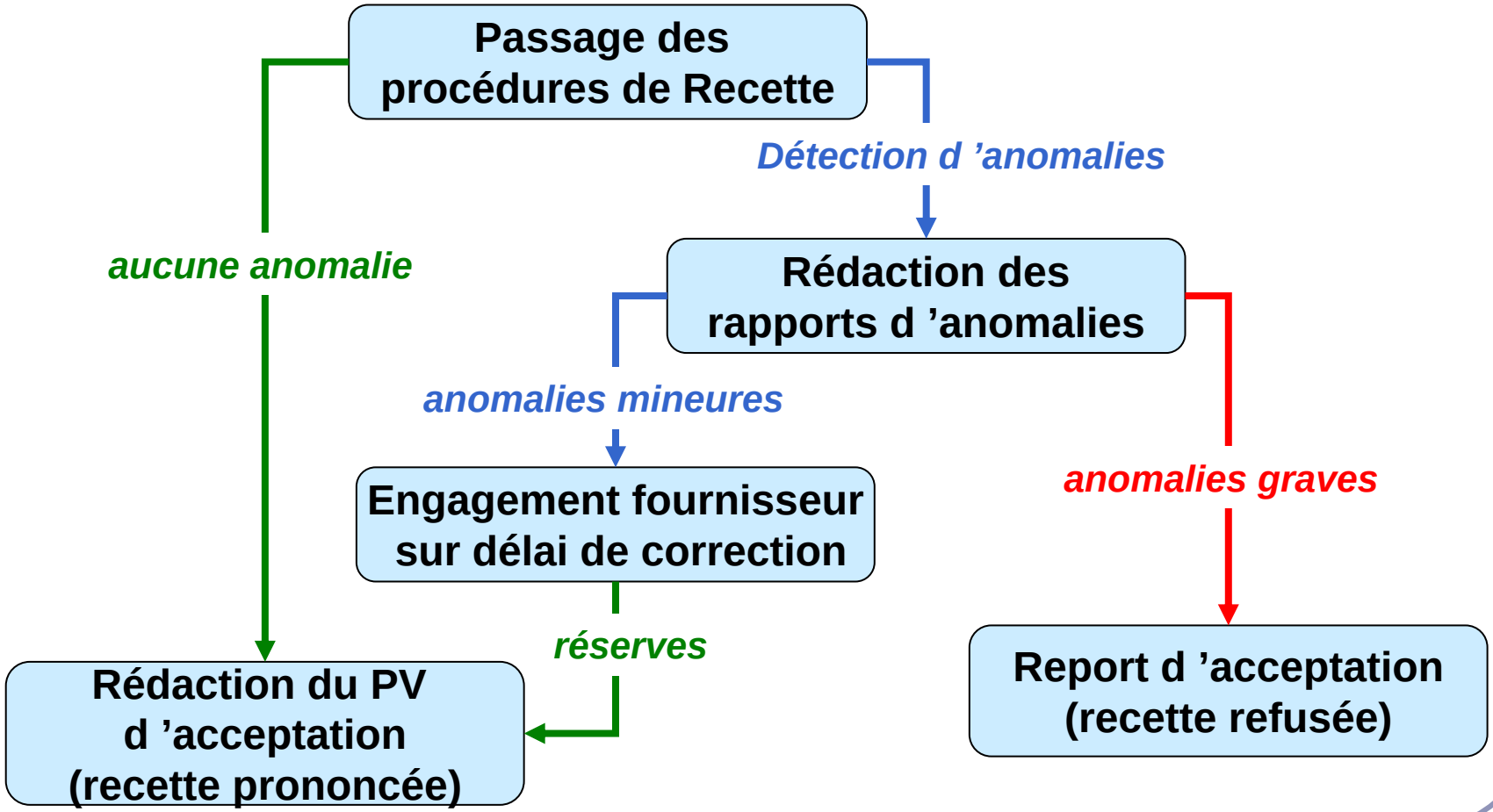
6. Annexes



Procédure d ’acceptation



Déroulement d'une séance de recette



Vocabulaire du test

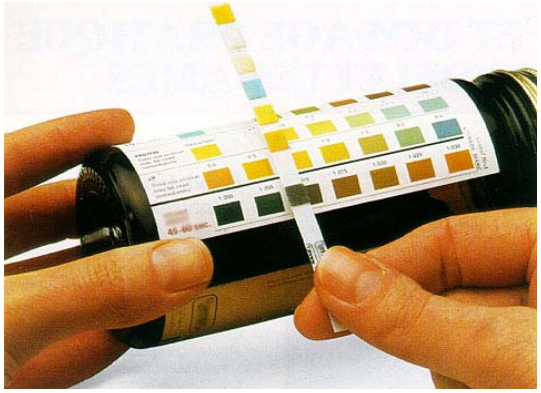
- **OBJET TESTE:** Produit logiciel, Module, Fonction, Composant, etc. que l'on exécute dans le but de vérifier que son comportement est conforme à un comportement de référence.
- **OBJECTIF DE TEST:** Caractéristique que l'on veut vérifier sur l'objet testé.
- **CAS DE TEST:** Configuration de données d'entrée et de sortie permettant de vérifier une partie d'un objectif.
- **PROCEDURE DE TEST:** Liste ordonnée des actions à exécuter par le testeur pour passer un groupe de tests.
- **DONNEES DE TEST:** Données en entrée + résultats attendus
- **JEU D'ESSAI:** Suite de données de test utilisées dans la même procédure
- **ETAPE DE TEST:** Unité de planification de la validation correspondant à l'atteinte d'un ou plusieurs objectifs et à un ensemble cohérent de procédures.

L ’oracle de test

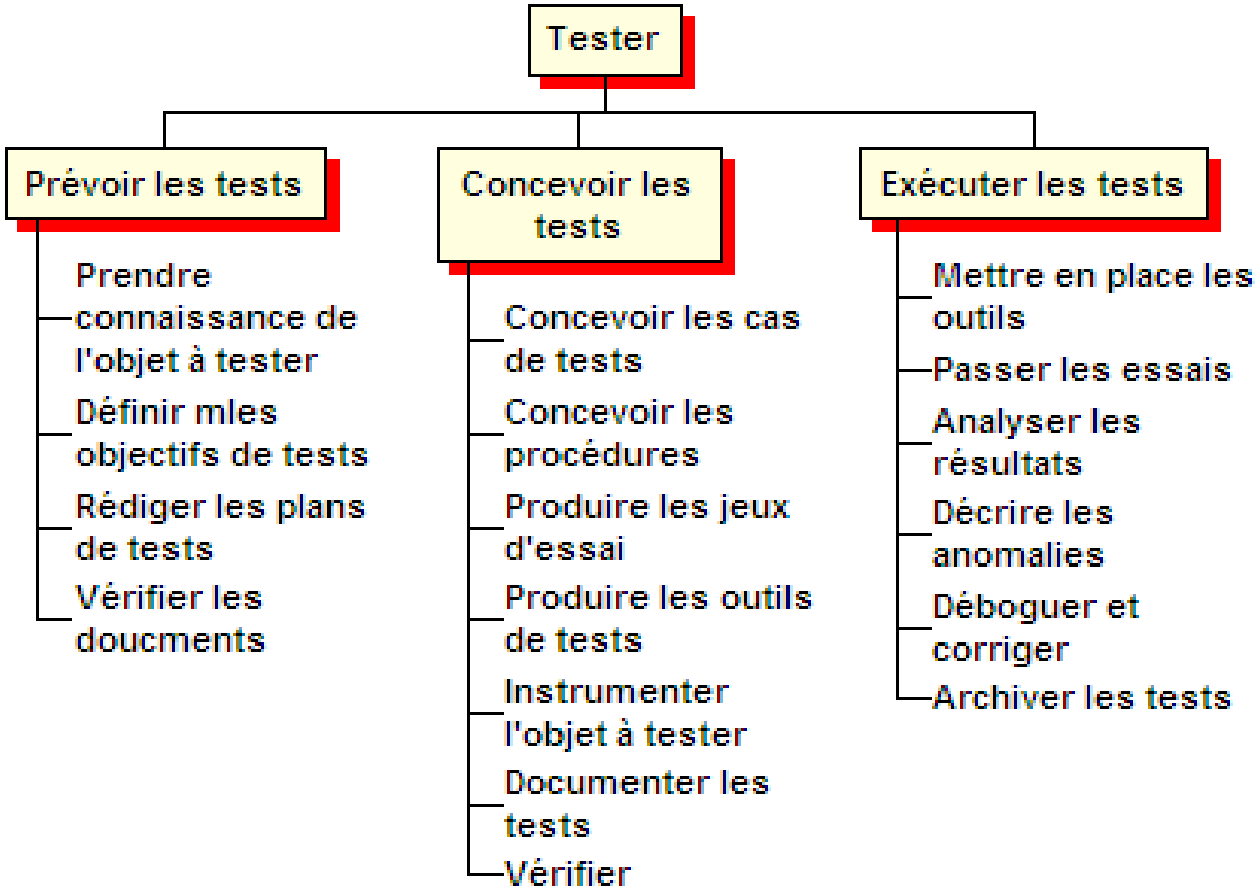
ORACLE = ce qui permet de dire si un test est bon ou non

2 METHODES

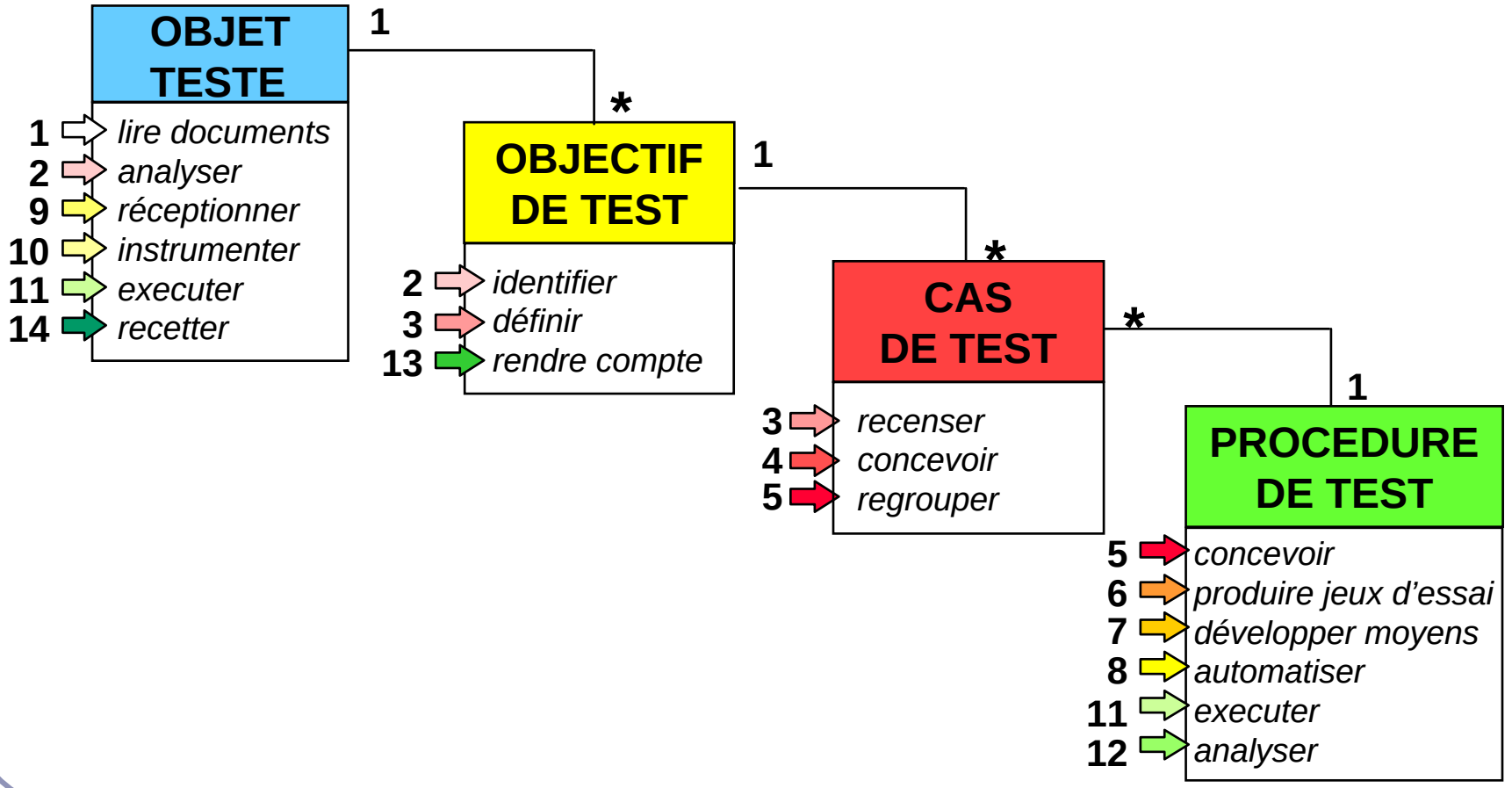
- **Définition, à priori, des résultats attendus pour comparaison avec les résultats obtenus**
  - **calcul « manuel » des résultats corrects**
  - **technique formelle (données d ’entrée = fonction inverse sur résultat attendu, logiciel symétrique, spécifications exécutables,...)**
- **Validation, à posteriori, des résultats obtenus**
  - **technique formelle (fonction inverse, démonstration = preuve)**
  - **technique empirique (analyse et vraisemblance = probabilité d ’erreur)**
  - **statistique (enregistrement/comparaison des résultats observés ou “capturés”)**



Un O.T. de test


Tester
Prévoir les tests
Prendre connaissance de l'objet à tester
Définir mles objectifs de tests
Rédiger les plans de tests
Vérifier les doucments
Concevoir les tests
Concevoir les cas de tests
Concevoir les procédures
Produire les jeux d'essai
Produire les outils de tests
Instrumenter l'objet à tester
Documenter les tests
Vérifier
Exécuter les tests
Mettre en place les outils
Passer les essais
Analyser les résultats
Décrire les anomalies
Déboguer et corriger
Archiver les tests

Processus de test



Procédures et scénarios

OBJET TESTE: Editeur Graphique **VERSION:** 1.3

ETAPE: Tests des fonctions d'édition

IDENTIFICATION PROCEDURE: *E-7.18* **VERSION:** 2

OBJECTIFS DE TEST: Vérifier la fonction de copier-coller

N°	Actions opérateur	Résultats attendus	Note
1	Sélectionner un objet graphique en le désignant avec la souris.	des «poignées» apparaissent sur les contours de l'objet	
2	Ouvrir le menu édition dans la barre d'outils, vérifier que l'option «coller» est inaccessible puis sélectionner l'option «copier».	l'option coller se «dégrise» dans le menu. les poignées disparaissent	
3	Ouvrir de nouveau le menu puis sélectionner l'option «coller»	l'objet initialement sélectionné est dupliqué dans la zone de dessin	

Journal de tests

OBJET TESTE: Editeur Graphique
VERSION: 1.3
ETAPE: Tests des fonctions d'édition
ITERATION: 2
NOM DU TESTEUR: Paul DURAND
DATE ET HEURE DE DEBUT: 21/05:97 à 8h45

N° Procédure	Résultats / Constats / Observations
E7-15	OK - RAS
E7-16	Correction du jeu d'essai *ET-16.4*
E7-17	Résultat correct en 12 secondes
E7-18	Rapport d'anomalie *FT-23*
FIN	*La procédure ET-19 ne peut être exécutée (fonction indisponible)*

Date et heure de fin: 21/05:97 à 11h15
Cause de l'arrêt des tests: fin des procédures
Nombre de rapports d'anomalies: *1*

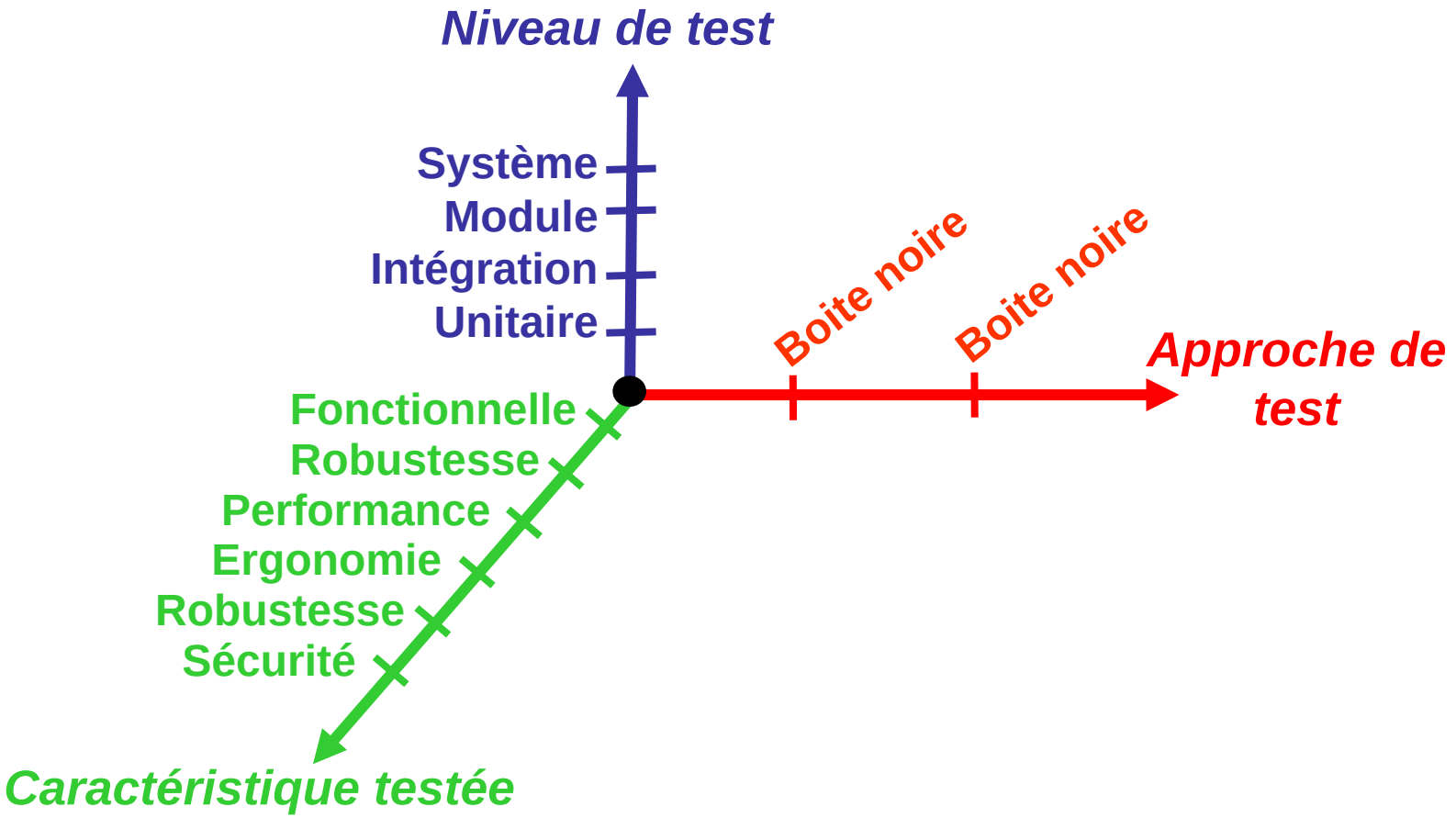
Test de charge

On exécute l'application dans des conditions d'exploitation particulières pour valider le système.

Objectifs :

- Mesures de performance
- Observation de la dégradation des transactions
- Simulation de l'activité maximale (Test de stress)
- Evaluation de l'endurance, de la robustesse, de la fiabilité
- Estimation de la capacité (Test de montée en charge)

Méthodologie de test



Tests de non-régression

Les tests de non-régression permettent de s'assurer que:

- Les modifications effectuées n 'ont pas dégradé le logiciel.
- Les parties non modifiées fonctionnent aussi bien qu'avant l'intervention.

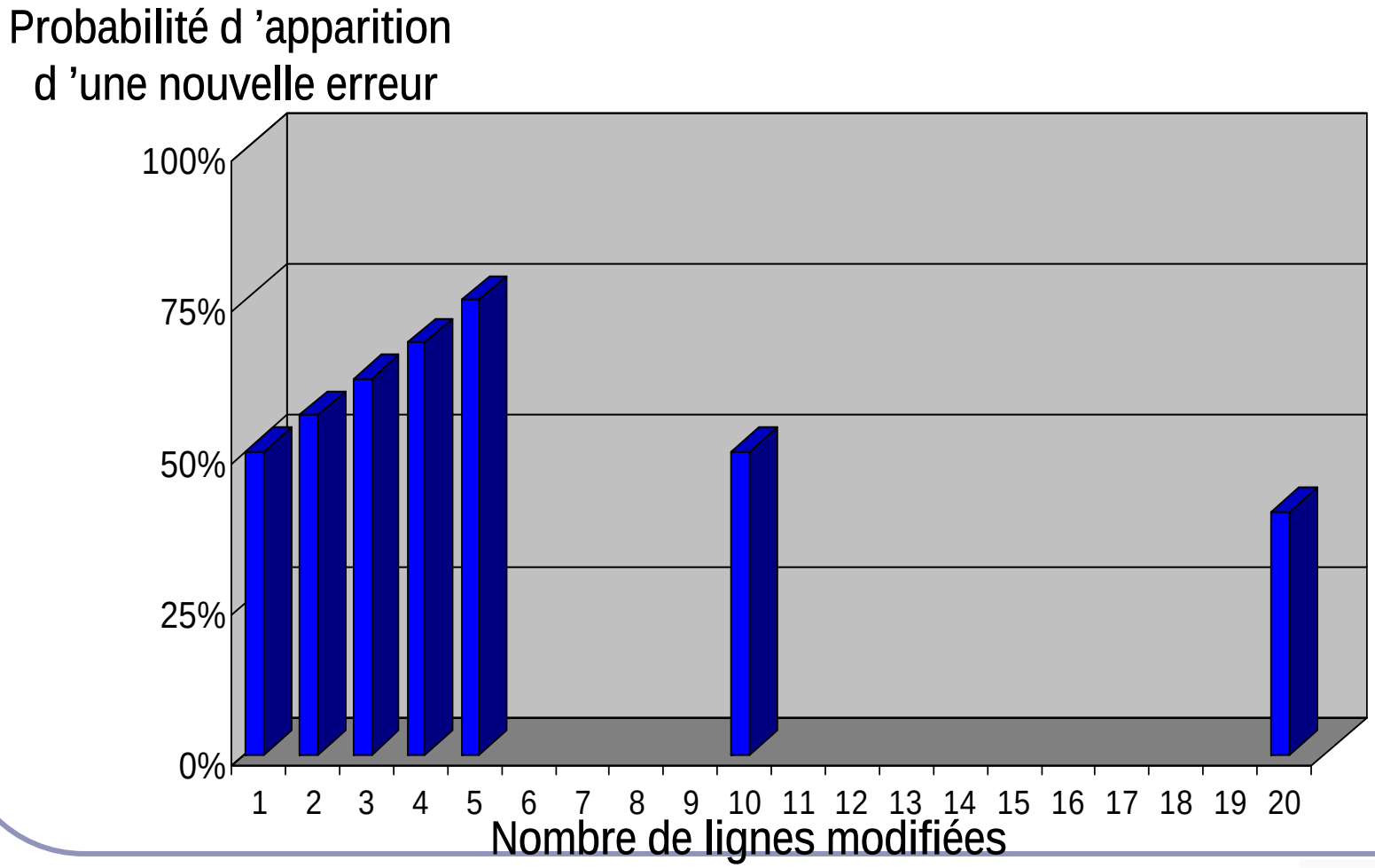
Les tests de non-régression doivent faire le compromis entre 2 objectifs contradictoires:

1. MINIMISER L'EFFORT en ne repassant pas les tests déjà passés
2. MINIMISER LE RISQUE de laisser échapper de nouveaux défauts.
   - Vision minimaliste: Repasser uniquement les tests modifiés.
   - Vision maximaliste: Repasser tous les tests portant sur l'objet modifié.

Un moyen pour atteindre les objectifs 1 et 2 :

*Prévoir, **dès le début du projet**, une stratégie globale de non-régression en ORGANISANT les tests.*

Probabilité d ’erreur après modification



Méthode d'organisation des tests

- IDENTIFIER LES TESTS
  - EX : fonction <=> jeu d'essai /composant <=> batterie de tests
- GERER DES VERSIONS DE TEST
  - EX : nouvelle version d'un composant => nouvelles versions des jeux d'essai
- ETABLIR DES REGLES :

Niveau 0	Repasser les tests modifiés
Niveau 1	Niveau 0 + Repasser les tests de cas de fonctionnement simples
Niveau 2	Niveau 1 + Repasser les tests de l'ensemble des instructions ou segments
Niveau 3	Tous les tests du composant

- AUTOMATISER LE PASSAGE DES TESTS

Développement piloté par les tests avec XP

- Les tests explicite la spécification

  Que veut-on vérifier ? = Quel est le résultat attendu?

- Les tests guident la conception

  Ce qui est facile à tester est facile à utiliser et à intégrer

- Les tests permettent de mesurer l'avancement

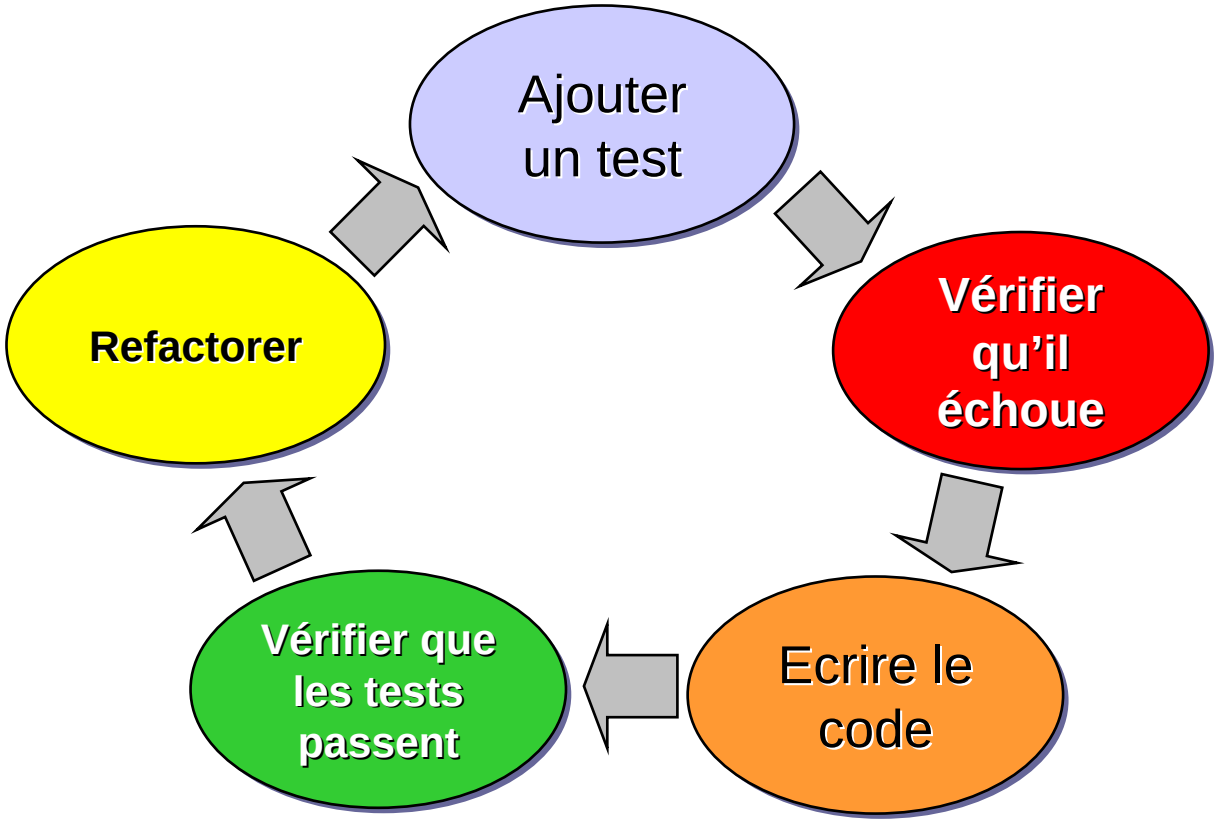
  « On sait ce qui marche et ce qui ne marche pas »

- Les tests améliorent la qualité

  La conception des tests prévient l'introduction de défauts

  L'exécution des tests met en évidence des défauts résiduels

Test Driven Development



Conception préalable des tests

Les tests sont conçus et écrits avant le code qu'ils sont supposés vérifier

- Par le programmeur ou par le client
- Pour être certain qu'ils existent
- Pour évaluer l'effort à fournir
- Pour identifier les données et les ressources nécessaires
- Pour prévoir leur exécution «automatique »
- Pour pouvoir tester dès que le programme existe

Qui doit tester ?

- **LE REALISATEUR DE L'OBJET A VERIFIER?**
  - Le plus courant, le moins cher
  - Le plus rapide
  - Le plus facile à organiser
  - Le plus difficile à maîtriser (avancement, efficacité des tests)
- **UN AUTRE REALISATEUR?**
  - Plus objectif sur ce que doit faire l'objet et sur son fonctionnement
  - Connaît le projet et le contexte
  - Peut communiquer aisément avec le réalisateur
  - Demande une organisation et une planification rigoureuse
- **UN "SPECIALISTE" DES TESTS ET DE LA VALIDATION?**
  - Le plus cher (il doit apprendre à connaître l'objet avant)
  - Le plus compétent du point de vue de la technique de test
  - Le plus rigoureux du point de vue de la qualité du produit
  - Demande une planification rigoureuse et la définition d'une mission

Source des jeux d ’essais

1. DONNEES REELLES

- Nécessitent souvent une sélection et un reformatage
- Doivent être examinées pour savoir ce qui est testé

2. GENERATION AUTOMATIQUE

- Selon une loi de probabilité représentative des données
- Génération pseudo-aléatoire
- Génération à partir d’une grammaire de données
- Génération à partir des spécifications
- Simulateur d’environnement

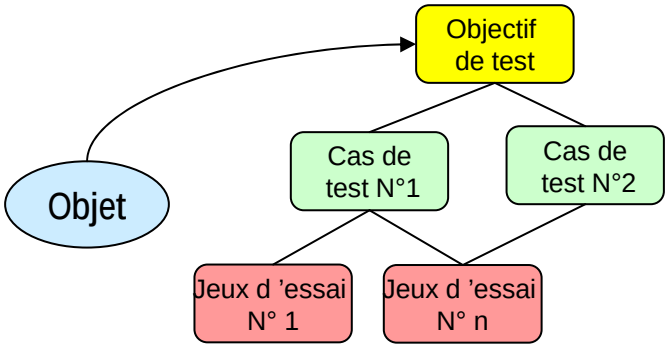
3. CONSTRUCTION MANUELLE

- Après correction des cas d’essai par tables de décision, combinaison de valeurs aux limites, etc..
- «Intuitive»: les cas difficiles, spéciaux, courants, les plus fréquents, etc..

La traçabilité des tests

La traçabilité des tests permet de

- mettre en relation les objectifs de test
  - les cas de test
  - les jeux d'essais et les procédures
  - les objets testés
- localiser les défauts (désigne l'objet testé)
- évaluer l'avancement du test
- gérer les modifications et les versions

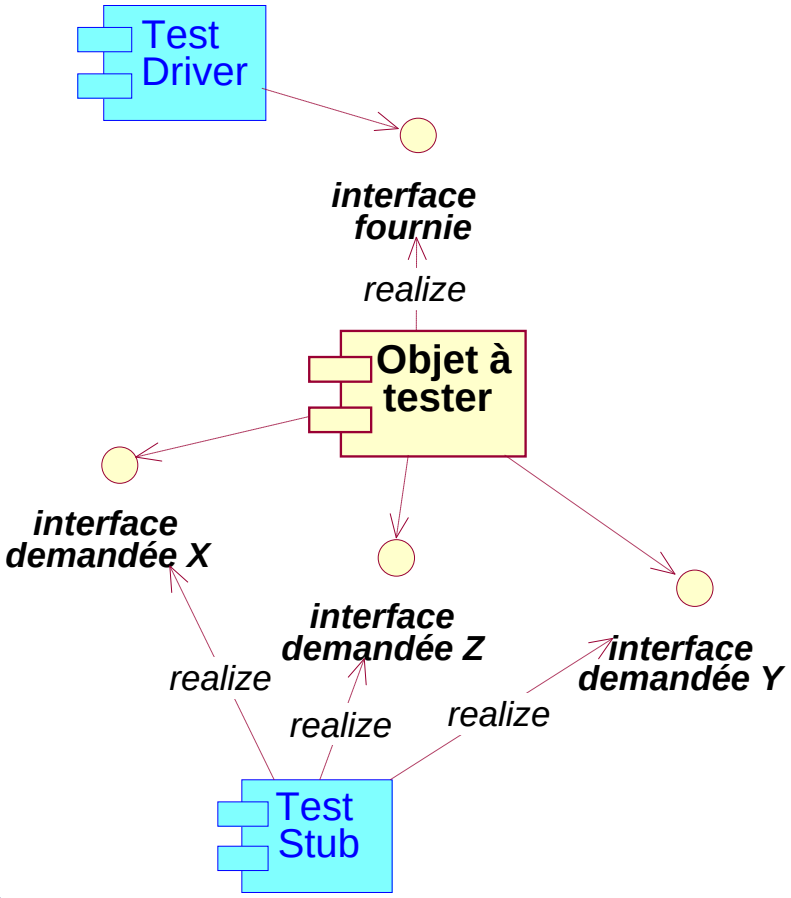


Organisation Hiérarchique

	O1	O2	O3	O4
J1	▤			
J2		▤		
J3	▤			▤
J4				▤

Matrices

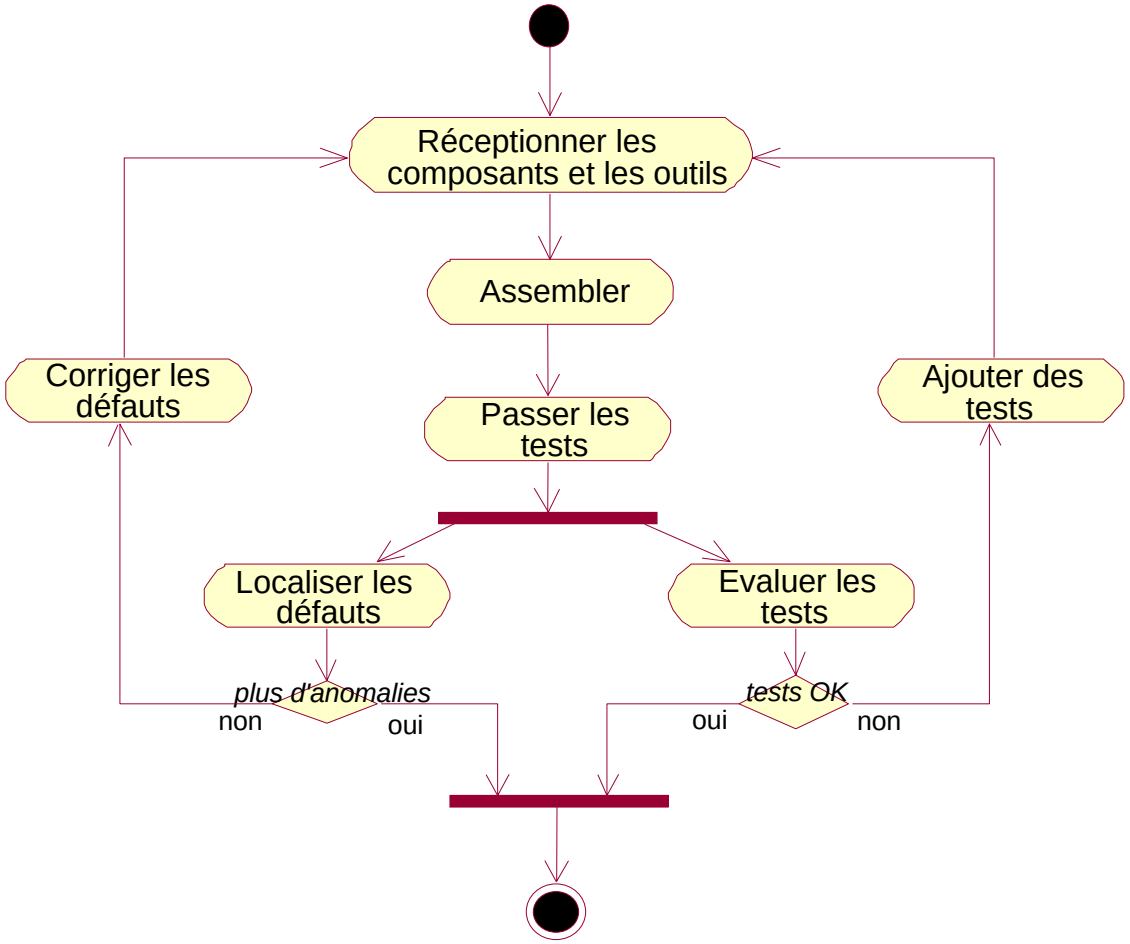
« Drivers » et « stubs »



- DRIVER = LANCEUR
- STUB = MUET = BOUCHON

- BUT : Permettre de tester l'objet dans un environnement simulé.
- INCONVENIENT : Leur réalisation peut nécessiter un effort important.

Déroulement des tests



Recommandations

- Rédiger un journal de test par séance.
- Passer tous les tests prévus.
- Passer les tests de non régression.
- N'arrêter les tests que sur un critère explicite.
- Observer sans chercher à déboguer.
- Rédiger des rapports d'anomalie.
- Corriger exclusivement les défauts du procédé de test.
- Automatiser autant que possible.

Automatiser les tests

Objectifs

- Améliorer l'efficacité et minimiser le risque d'erreur humaine
- Pouvoir répéter un test pour localiser un défaut
- Rejouer les tests pour identifier les éventuelles régressions

Moyens

Développer ou utiliser des outils permettant de

1) « Dérouler » des scénarios prédéfinis

2) Enregistrer les données de test

3) Comparer les résultats obtenus avec les résultats attendus

Tester une classe

- Définir en priorité les opérations publiques et écrire immédiatement les tests associés
- Tenir compte du scénario de test pour planifier le codage
- Tester une opération le plus tôt possible (dès qu'elle « marche »)
- Nettoyer le programme dès que le test est OK (suppression du code de debug, commentaires ou compilation conditionnelle)
- Fournir le programme de test en même temps que la classe (documentation)
- Ajouter des tests chaque fois que la classe évolue

En Java

- écrire systématiquement un « main » qui contient les tests des opérations de la classe
- Utiliser les assertions (mot clef « assert » depuis java 1.4)
- Utiliser les exceptions (gestion des erreurs)

Tester une classe – Exemple (1)

Étape 1:
Écriture
des tests

exécution
⇒Tous les tests
sont KO

```
/**
 * <p>Titre : Discriminant </p>
 * <p>Description : Permet de calculer le discriminant d'un binome </p>
 * <p>Projet : AlgebriX </p>
 * @author PGX
 * @version 1.0
 */

public class Discriminant {

  public Discriminant(double _a, double _b, double _c){
     }

  public double getValeur() {
     return 0.0;
     }

  public static void main(String[] args) {
     Discriminant d_test = new Discriminant(4,4,1);
     assert(d_test.getValeur() == 0);
     d_test = new Discriminant(2,2,2);
     assert(d_test.getValeur() == -12);
     d_test = new Discriminant(4,0,-1);
     assert(d_test.getValeur() == 16);
     d_test = new Discriminant(0,0,-0);
     assert(d_test.getValeur() == 0);
  }
}
```

Tester une classe – Exemple (2)

Étape 2:

Écriture du code des méthodes

Exécution et mise au point jusqu'à ce que tous les tests soient OK.

```
/**
 * <p>Titre : Discriminant </p>
 * <p>Description : Permet de calculer le discriminant d'un binome </p>
 * <p>Projet : AlgebriX </p>
 * @author PGX
 * @version 1.0
 */

public class Discriminant {
  private double delta = 0.0;

  public Discriminant(double _a, double _b, double _c){
     delta = (_b*_b) - (4*_a*_c);
     }

  public double getValeur() {
     return delta;
     }

  public static void main(String[] args) {
     Discriminant d_test = new Discriminant(4,4,1);
     assert(d_test.getValeur() == 0);
     d_test = new Discriminant(2,2,2);
     assert(d_test.getValeur() == -12);
     d_test = new Discriminant(4,0,-1);
     assert(d_test.getValeur() == 16);
     d_test = new Discriminant(0,0,-0);
     assert(d_test.getValeur() == 0);
  }
}
```

Tester une classe – Exemple (3)

```
public class Binome
{
    public Binome (double _a, double _b, double _c){
    }

    public double[] calculerRacines()
    {
      double[] racines = null;
      return racines;
    }
    public static void main(String[] args) {
      double racines[] = null;
      Binome binome;
      // test 1: Un binome avec racine double
      binome = new Binome(4,4,1);
      racines= binome.calculerRacines();
      assert ((racines.length ==1) && (racines[0] == -0.5));
      // test 2: Un binome sans racine
      binome = new Binome(2,2,2);
      racines= binome.calculerRacines();
      assert (racines == null);
      // test 3: Un binome avec 2 racines
      binome = new Binome(4,0,-1);
      racines= binome.calculerRacines();
      assert ((racines.length ==2) && (racines[0] == -0.5) && (racines[1] == 0.5));
    }
}
```

Tester une classe avec JUnit

Framework « open source » www.junit.org

- **Dédié aux tests unitaires**
- **Conçu par Kent Beck (auteur de XP) et Erich Gamma (créateur de Design Patterns célèbres).**
- **Cours: http://selab.fbk.eu/swat/slide/3_JUnit.ppt**

- **Remplace l'utilisation de la méthode main()**
- **Consiste à dériver une classe "TestCase" (incluie dans le framework) :**

```
import junit.framework.TestCase;
public class TestMyClass extends TestCase {
```

- **Plugin Eclipse**

Nom de classe de la forme TestMyClass or MyClassTest pour permettre au TestRunner de trouver automatiquement les classes de tests.

existe dans de nombreux autres langages

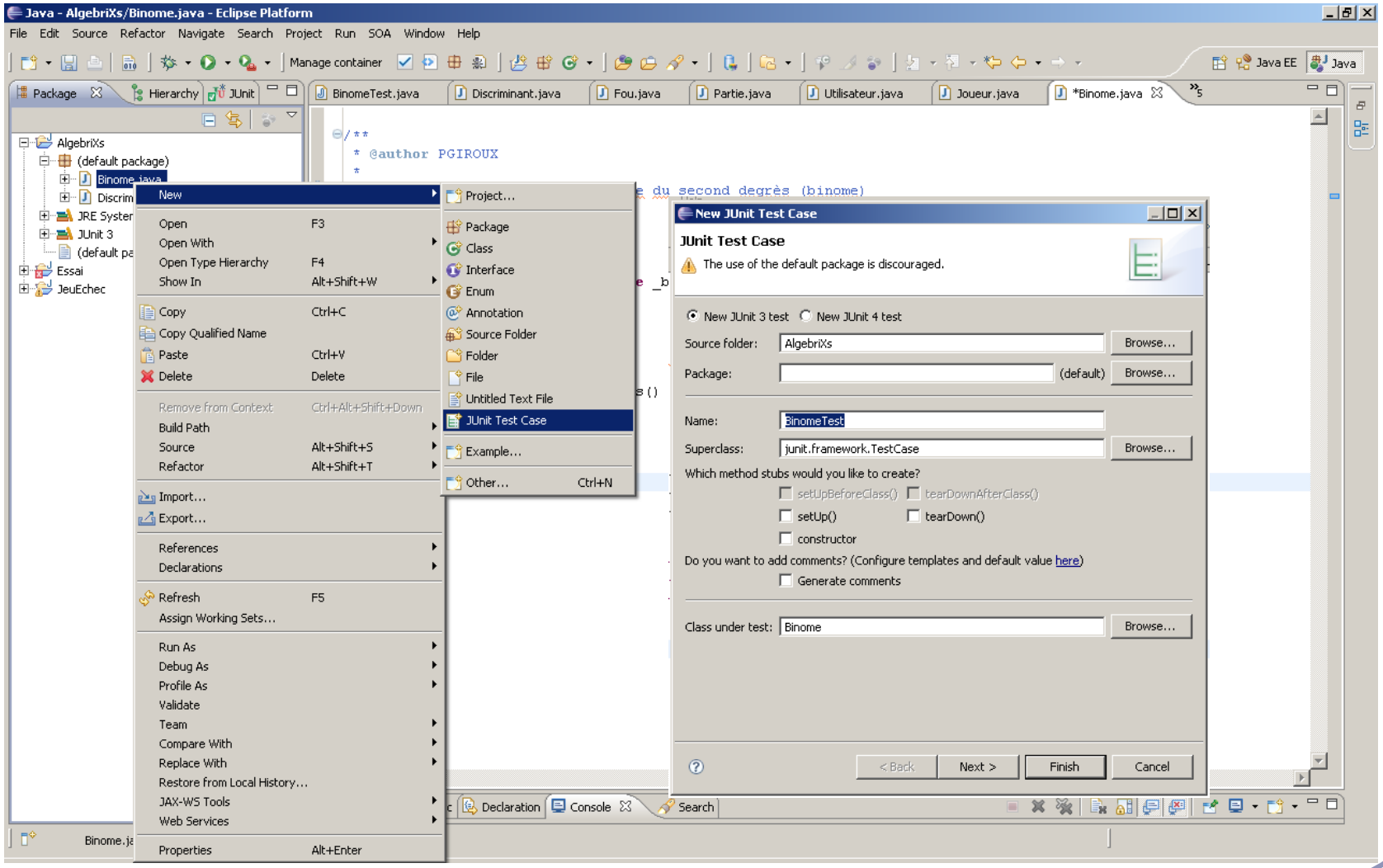
Classe de test

```
import junit.framework.*;

public class TestBinome extends TestCase{

    public static void main (String[] args) {
        junit.textui.TestRunner.run (suite());
    }

    public static Test suite() {
        return new TestSuite(TestBinome.class);
    }

    public void testDeuxRacines(){
      double racines[] = null;
      Binome binome = new Binome(4,0,-1);
      racines= binome.calculerRacines();
      assertEquals(2, racines.length);
      assertEquals(-0.5, racines[0], 0);
      assertEquals(0.5, racines[1], 0);
    }

    public void testRacineDouble(){
      double racines[] = null;
      Binome binome = new Binome(4,4,1);
      racines= binome.calculerRacines();
      assertEquals(1, racines.length);
      assertEquals(-0.5, racines[0], 0);
    }

    public void testPasDeRacine(){
      double racines[] = null;
      Binome binome = new Binome(2,2,2);
      racines= binome.calculerRacines();
      assertEquals(1, racines.length);
    }
}
```

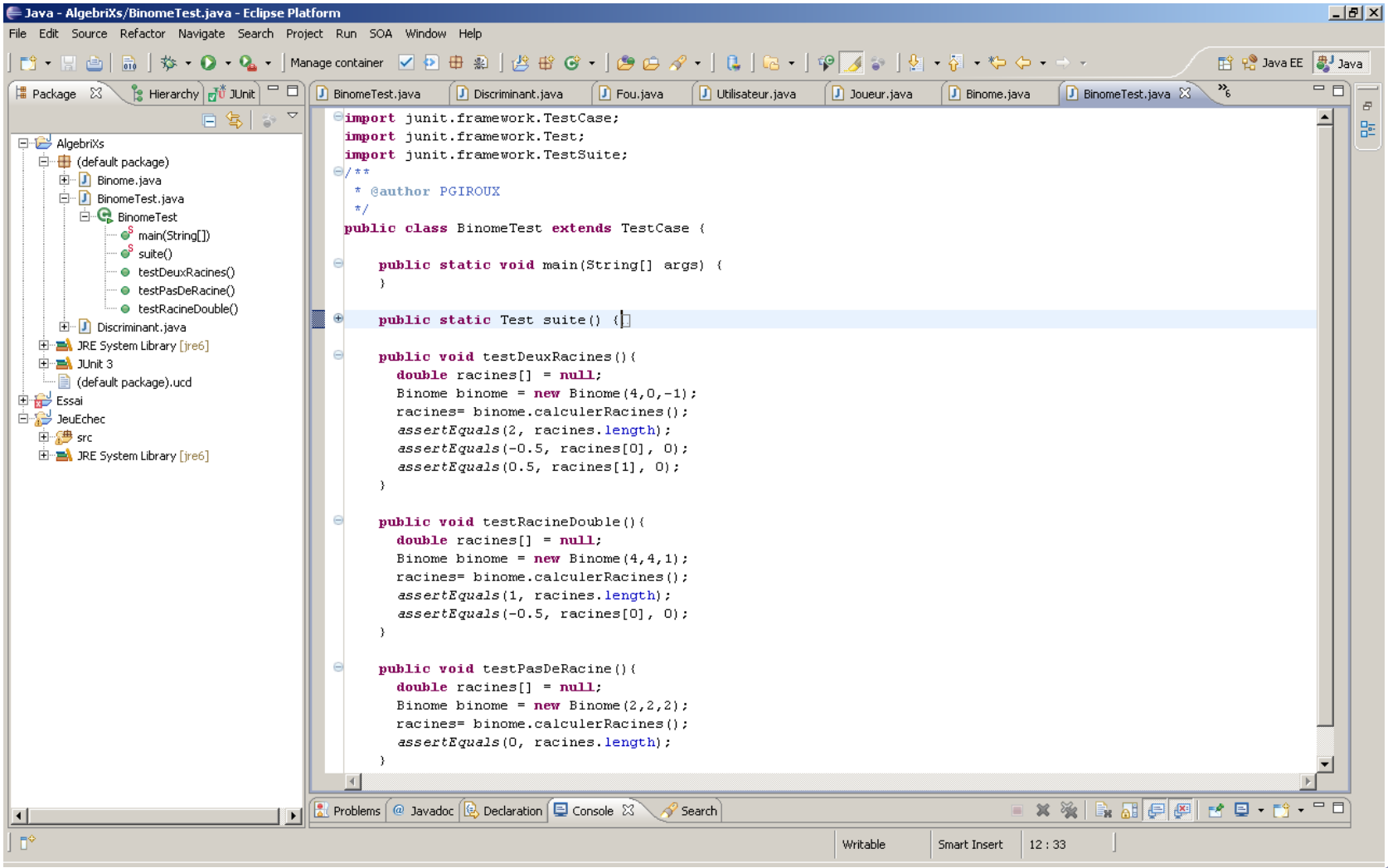
Les assertions de test

- assertTrue(String *msg*, Boolean *test*)
- assertFalse(String *msg*, Boolean *test*)
- assertNull(String *msg*, Object *object*)
- assertNotNull(String *msg*, Object *object*)
- assertEquals(String *msg*, Object *expected*, Object *actual*)
- assertSame(String *msg*, Object *expected*, Object *actual*)
- assertNotSame(String *msg*, Object *expected*, Object *actual*)

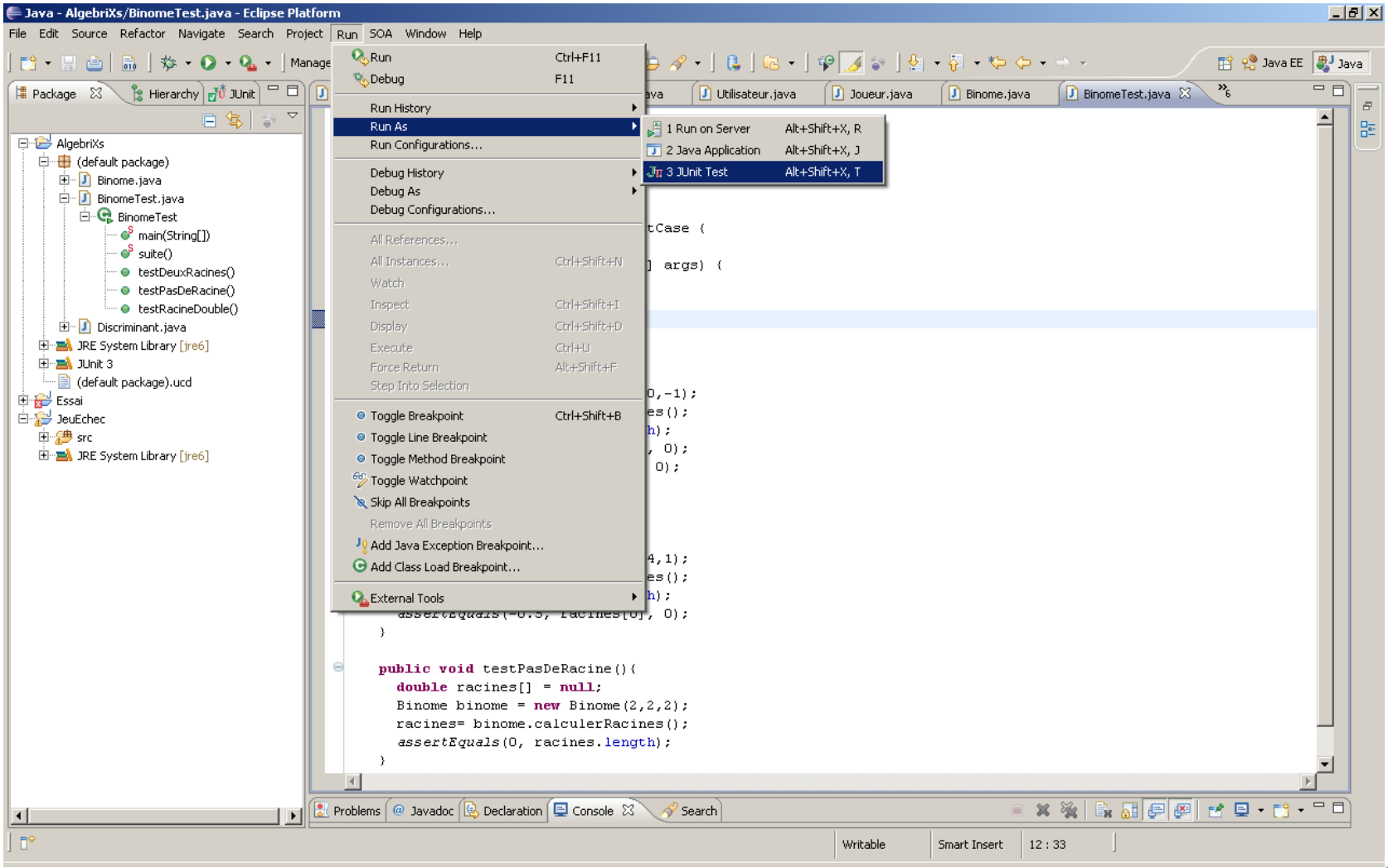
Utilisation du plugin Eclipse



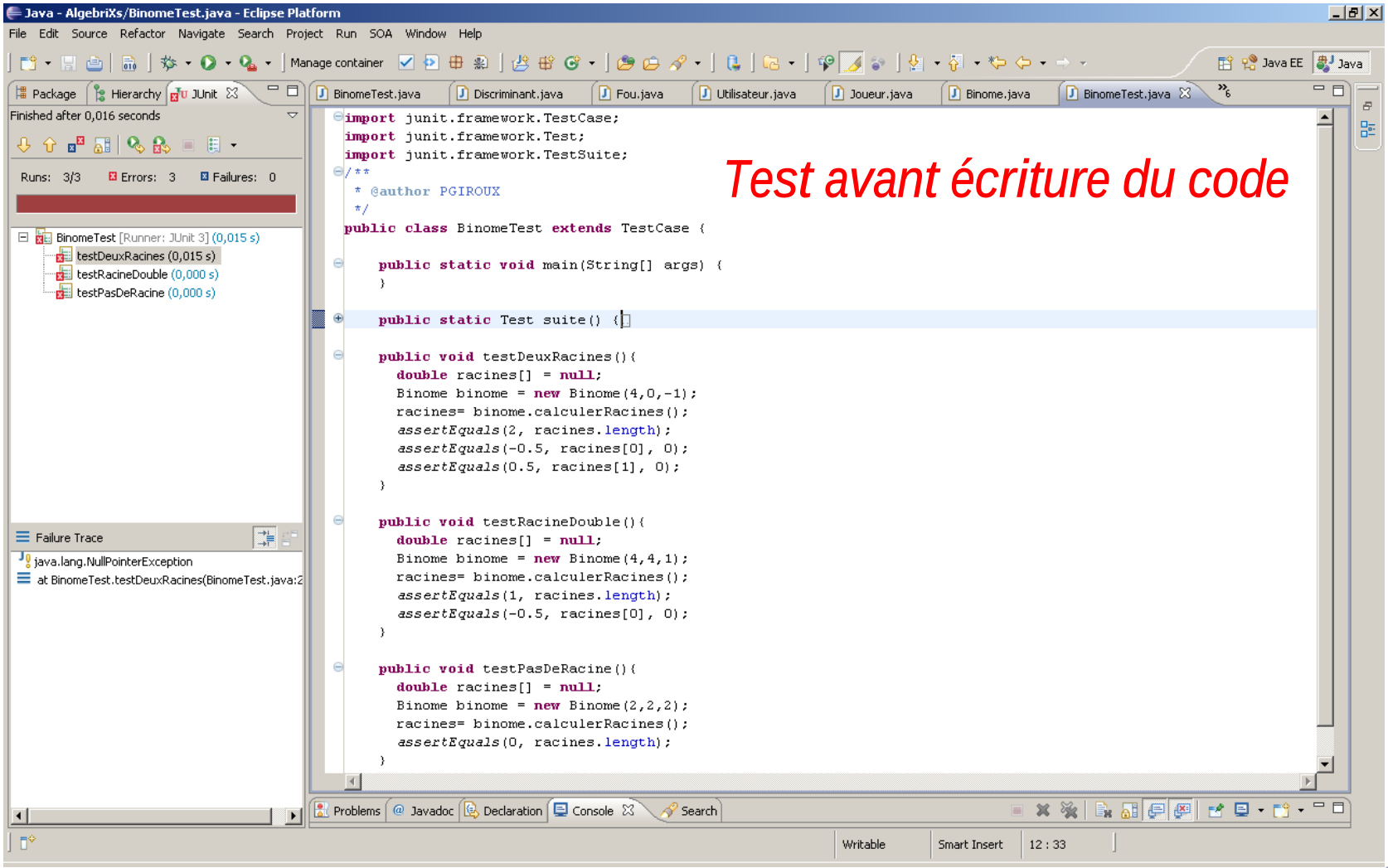
Utilisation du plugin Eclipse



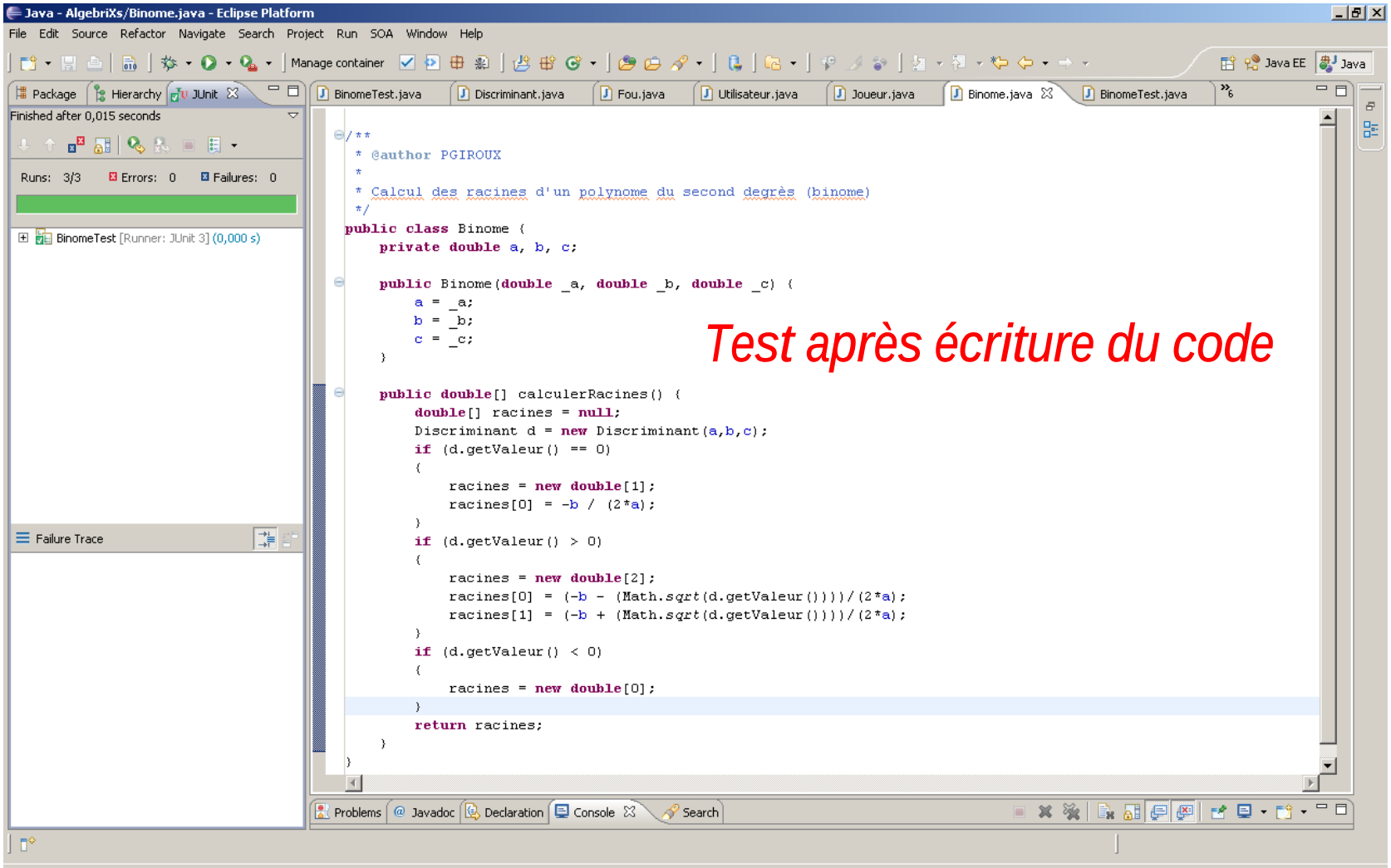
Utilisation du plugin Eclipse



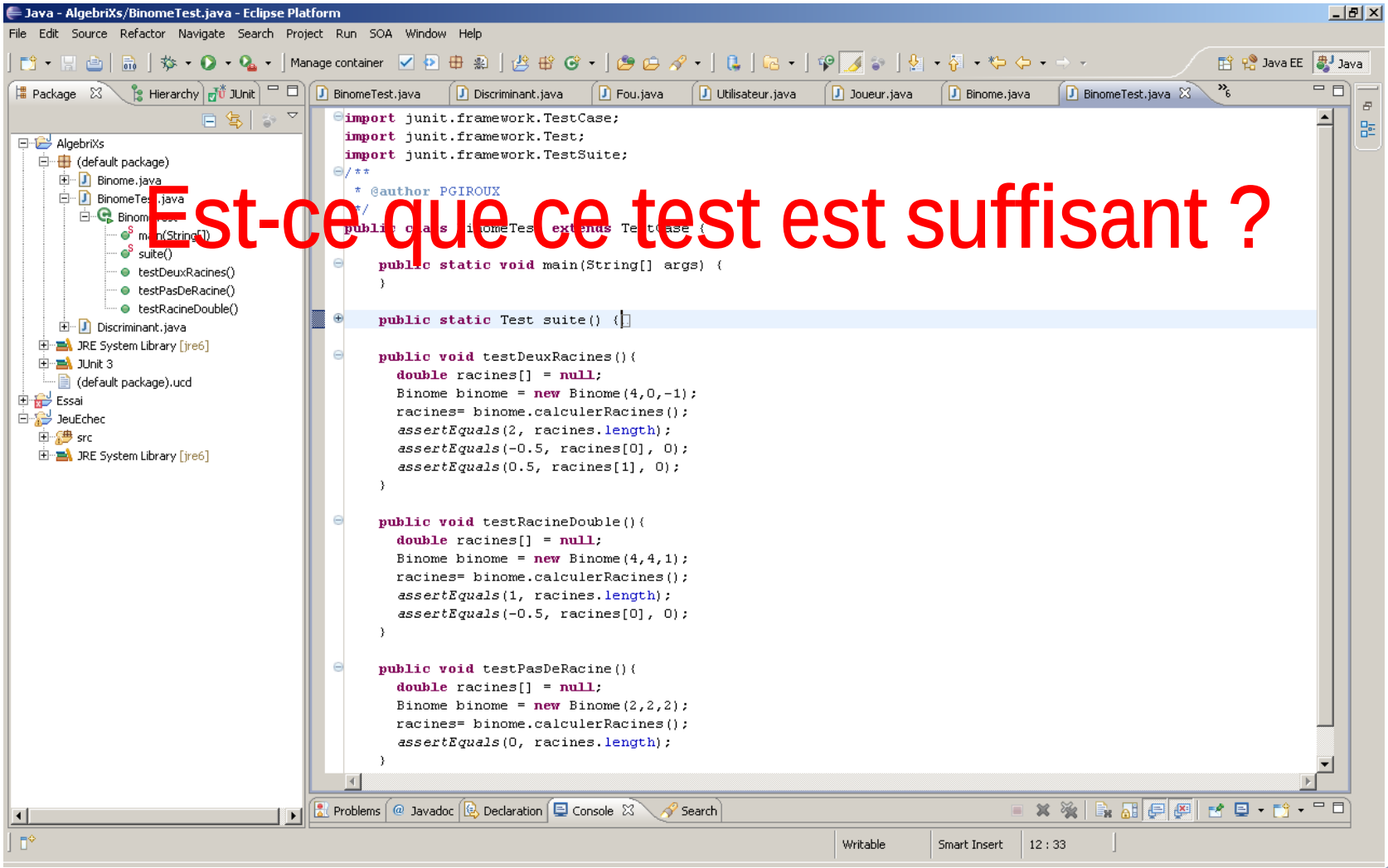
Utilisation du plugin Eclipse



Utilisation du plugin Eclipse



Utilisation du plugin Eclipse



Évaluation des tests

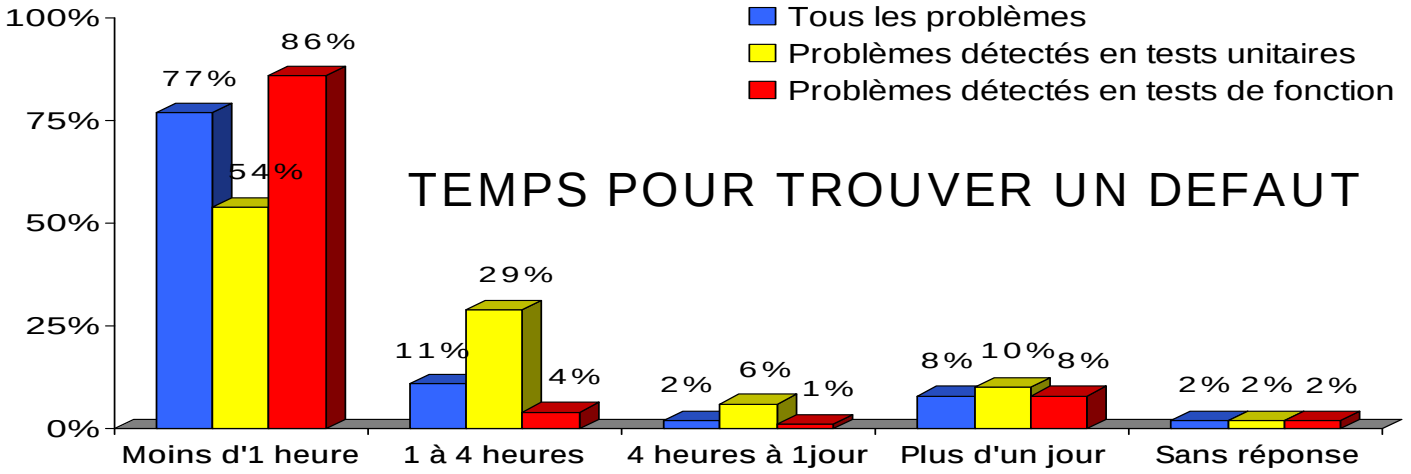
3 OBJECTIFS:

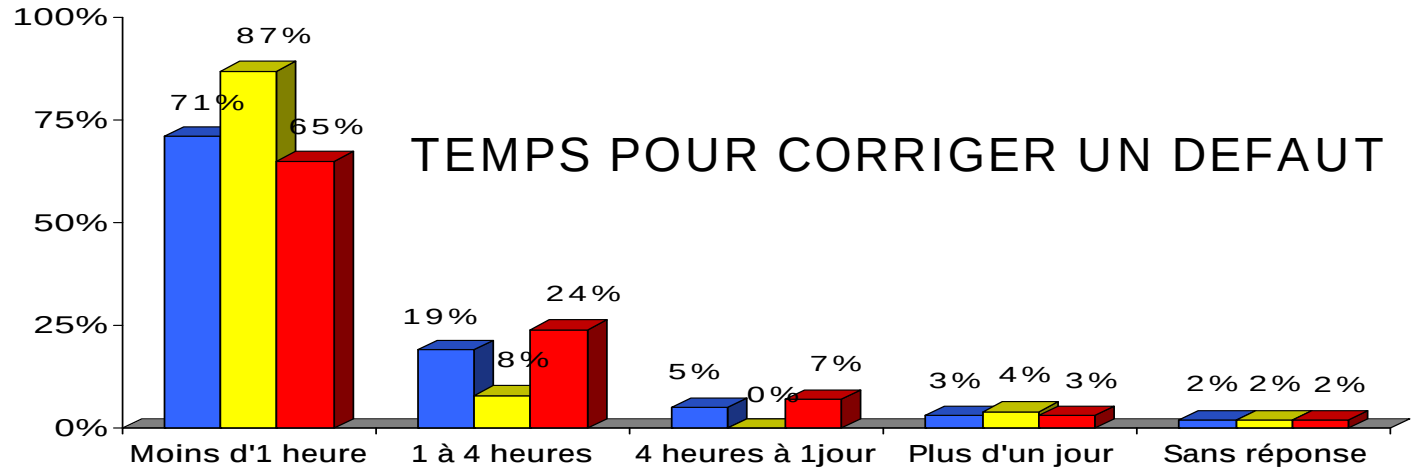
- **Evaluer la qualité du composant**
  - Quel est le nombre de défaut par rapport à la taille du composant?
  - Quelles sont les performances du composant?
  - Quelle est la complexité des corrections par rapport à la taille du composant?
- **Evaluer la complétude du test**
  - Est-ce que tous les tests ont été passés avec succès?
  - Est-ce que les objectifs sont atteints?
  - Est-ce que la couverture spécifiée est réalisée?
- **Evaluer l'effort de test**
  - Quel est le nombre de défauts dus au test?
  - Combien de séances ont été nécessaires?
  - Combien de temps ont duré les tests?

Les 7 règles de localisation

- Considérer que le défaut peut venir du test en non de l'objet testé.
- Travailler sur l'ensemble des anomalies et non pas sur une anomalie particulière.
- Recourir à des techniques de localisation différentes.
- Garder une trace des impasses dans la localisation (ce qui est supposé correct à priori).
- Quand un objet est conforme à sa spécification, mettre en cause la spécification.
- Recourir à des aides externes après 1 journée de recherches infructueuses.
- Savoir que moins de 2% d'anomalies ne reçoivent jamais d'explication.

Trouver le défaut et le corriger





Recommandations

- Tous les tests doivent être exécutés avec succès
- Le code doit être lisible et simple
- Le code ne doit pas être dupliqué
- Le nombre de classes, de méthodes et de lignes de code doit être minimal
- Le code doit être documenté (par des commentaires et par des tests)
- Le code d'instrumentation (destiné au debug) doit être éliminé avant la dernière exécution des tests
- Utiliser des outils de déverminage (analyse de la gestion de la mémoire)

Tous les logiciels sont bogués

Microsoft a publié la liste des anomalies résiduelles de Windows

Mr. Hyan-Lee a fait une erreur: il l'a imprimée …

Attention aux méchants bugs !!!