Verification et Validation

Malédiction du logiciel

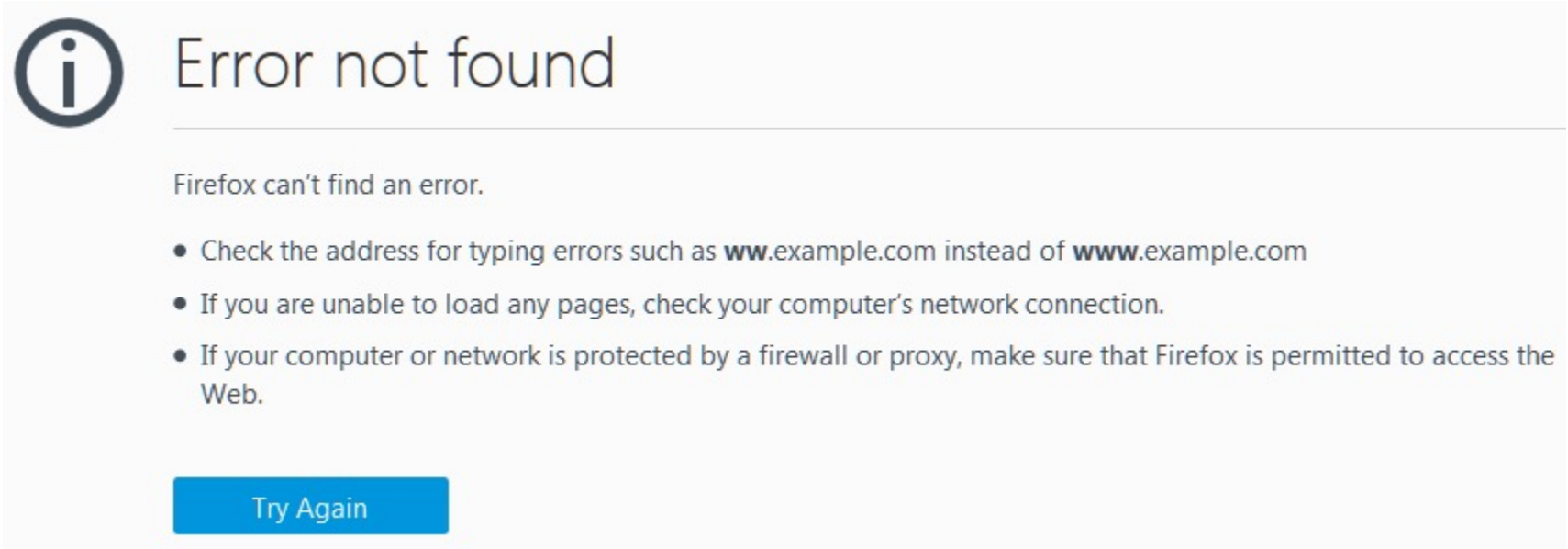
- L'erreur est humaine, les programmes sont développés par des humains, tous les programmes contiennent des erreurs
  - On peut assumer qu'un programme contient des erreurs
- Les erreurs sont difficiles à identifier :
  - Grande complexité des logiciels
  - Invisibilité du système développé
  - Pas de principe de continuité
- L'activité de V&V est essentielle au développement de logiciels de qualité
- Certaines organisations dépensent près de 50% du budget sur le test, mais le logiciel livré est rarement fiable

Qualité du logiciel

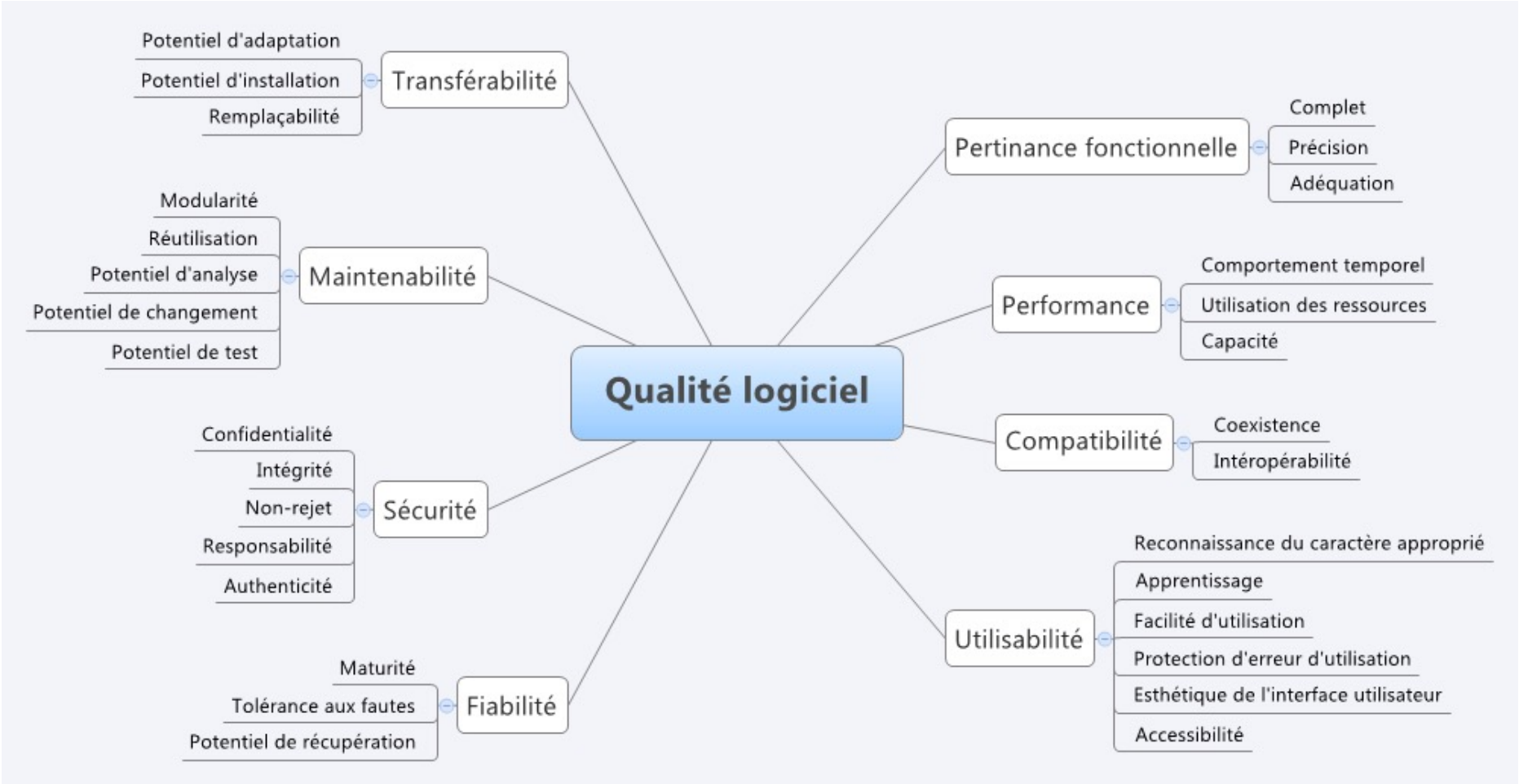
- Pas « l'excellence »
- Mesure si le logiciel satisfait les besoins
  - Fonctionnels et non-fonctionnels
- Chaque professionnel du logiciel est responsable de s'assurer que son travail est correct
  - Qualité prise en compte dès le début

Qualité, définie négativement

La qualité est l'absence de lacune, carence et autres défauts.



ISO 25010: Modèle de qualité du logiciel



http://iso25000.com/index.php/en/iso-25000-standards/iso-25010

Pertinence Fonctionnelle

- Logiciel fournit des fonctionnalités en adéquation avec les besoins exprimés et tacites quand il est utilisé sous conditions spécifiées
- **Complet**
  - Fournit les fonctionnalités qui correspondent à toutes les tâches et objectifs de l'utilisateurs spécifiés
- **Précision**
  - Fournit le résultat attendu avec la précision attendue
- **Adéquation**
  - Permet d'accomplir les tâches et objectifs spécifiés

Performance

- Logiciel présente des performances relatives aux ressources utilisées
- **Comportement temporel**
  - Fournit un temps de réponse, de traitement et un taux de débit convenables aux exigences
- **Utilisation des ressources**
  - Utilise une quantité et des types de ressource convenables aux exigences
- **Capacité**
  - Respecte les limites attendues

Compatibilité

- Deux composants peuvent échanger des informations et/ou effectuer leurs tâches, tout en partageant le même environnement matériel ou logiciel
- **Coexistence**
  - Peut effectuer ses fonctionnalités efficacement tout en partageant un environnement et des ressources communes avec un autre logiciel, sans le nuire
- **Interopérabilité**
  - Coopération avec d'autres logiciels en échangeant de l'information et utilisant l'information échangée

Utilisabilité

- Logiciel peut être utilisé par l'utilisateur d'une manière efficace et satisfaisante
- **Reconnaissance du caractère approprié**
  - Permet à l'utilisateur de reconnaitre s'il répond à ses besoins
- **Apprentissage**
  - Permet à l'utilisateur d'apprendre à l'utiliser efficacement, sans risque et d'une manière satisfaisante
- **Facilité d'utilisation**
  - Permet de l'utiliser et de le contrôler facilement
- **Protection d'erreur d'utilisation**
  - Empêche l'utilisateur de commettre des erreurs
- **Esthétique de l'interface utilisateur**
  - Attire et satisfait l'interaction pour l'utilisateur
- **Accessibilité**
  - Peut être utilisé par des personnes ayant des caractéristiques et capacités très variées

Fiabilité

- Logiciel performe des fonctionnalités attendues dans des conditions attendues sur une durée attendue
- **Maturité**
  - Satisfait les besoins de fiabilité dans lors de son utilisation normale
- **Disponibilité**
  - Est opérationnel et accessible lorsqu'on en a besoin
- **Tolérance aux fautes**
  - Fonctionne tel que prévu malgré la présence de fautes matériel ou logiciel
- **Potentiel de récupération**
  - En cas de panne ou d'échec, récupère les données affectées et rétablit l'état attendu du système

Sécurité

- Logiciel protège l'information pour que les personnes ou autres systèmes aient les accès requis conformément à leur type et niveau d'autorisation
- **Confidentialité**
  - Garantie que les données ne sont accessibles qu'à ceux qui y sont autorisés
- **Intégrité**
  - Empêche l'accès et la modification non-autorisée des programmes ou données
- **Non-rejet**
  - Peut prouver que des actions ou évènements ont eu lieu, afin de ne pas les rejeter plus tard
- **Responsabilité**
  - Peut retracer de façon unique les actions d'une entité jusqu'à celle-ci
- **Authenticité**
  - Peut prouver que l'identité d'un sujet ou d'une ressource est bien celle déclarée

Maintenabilité

- Logiciel peut être modifié efficacement pour l'améliorer, le corriger ou l'adapter aux changements dans l'environnement et dans les exigences
- **Modularité**
  - Composé de composants distincts de sorte que changer un composant a un impact minimal sur les autres
- **Réutilisation**
  - Composant peut être utilisé dans plus qu'un système, ou pour construire de nouveaux composants
- **Potentiel d'analyse**
  - Évaluer l'impact d'un changement dans une ou plusieurs parties, diagnostiquer les défauts, ou identifier les parties à modifier
- **Potentiel de changement**
  - Peut être modifié facilement sans introduire des défauts ou dégrader la qualité
- **Potentiel de test**
  - Permet d'établir des critères pour tester un composant et tester pour déterminer si ces critères sont satisfaits

Transferabilité

- Logiciel peut être transféré d'un matériel, logiciel ou environnement d'utilisation à un autre
- **Potentiel d'adaptation**
  - Peut être adapté ou migré efficacement à d'autres matériel, logiciel ou environnements d'utilisation
- **Potentiel d'installation**
  - Peut être installé et désinstallé facilement dans un environnement
- **Remplaçabilité**
  - Peut remplacer un autre logiciel ayant le même but, dans un même environnement

Assurance qualité

Assurance qualité logiciel (AQ)

- Les membres de l'équipe AQ doivent s'assurer que les développeurs fournissent un travail de haute qualité
  - À la fin de chaque workflow
  - Quand le produit est complété
  - AQ doit être appliquée au processus de développement aussi
    - *Capability Maturity Model Integration (CMMI)*
      - Ch. 3.13
- Équipes AQ et de développement sont horizontales
  - Aucun n'a de pouvoir sur l'autre
- Le travail de l'équipe AQ est de **vérifier** et de **valider** le logiciel et son processus de développement

Vérification et validation (V&V)

- **Vérification**: est-ce que le logiciel **a été fait correctement** ?
  - Détermine si le workflow a été complété correctement
  - Assure que le logiciel est conçu pour livrer toutes les fonctionnalités exigées
- **Validation**: est-ce que le logiciel **fait la bonne chose** ?
  - Détermine si le logiciel répond aux besoins
  - Assure que la fonctionnalité exigée est le comportement attendu du logiciel
  - Se fait typiquement une fois les vérifications complétées

Développement de logiciel fiable

Impossible de prévenir tous les problèmes !

- **Prévention de fautes**
  - Détecter les fautes sans exécution, avant la livraison
    - Vérification formelle, inspection, développement rigoureux
- **Prévision des fautes**
  - Mesurer la probabilité de l'occurrence de faute
    - Métriques de qualité, méthodes statistiques
- **Détection et correction de fautes**
  - Trouver une faute et la corriger
    - V&V: débogage, test
- **Tolérance aux fautes**
  - Récupération après des fautes qui arrivent lors de l'exécution
    - Gestion d'exception, blocks de récupération, programmation en N versions (ex: Ada)

Inspections

Vérification sans exécution

- Principe fondamental :
  - On ne révise pas son propre travail
  - Synergie entre l'équipe de développeurs et celle de l'AQ
- Tester le logiciel sans exécuter de cas de test
  - **Révision du logiciel**: lire attentivement
  - Vérifier **tous les artéfacts** produits par chaque workflow à chaque incrément
- Technique : **inspection**

Inspection

- L'équipe d'inspection
  - Modérateur
  - Membre de l'équipe qui effectue le workflow courant
  - Membre de l'équipe qui effectue le prochain workflow
  - Membre de l'équipe AQ
- Étapes formelles :
  - **Aperçu** de l'artefact à réviser
  - **Préparation** de la liste des types de fautes trouvées
    - Statistiques de fautes
    - Se concentre là où on a identifié le plus de fautes
  - **Inspection** méticuleuse qui parcourt tout l'artefact
  - Le responsable de l'artéfact **corrige** les fautes
  - **Suivi** pour s'assurer que toutes les fautes ont été traitées

Statistiques d'inspection

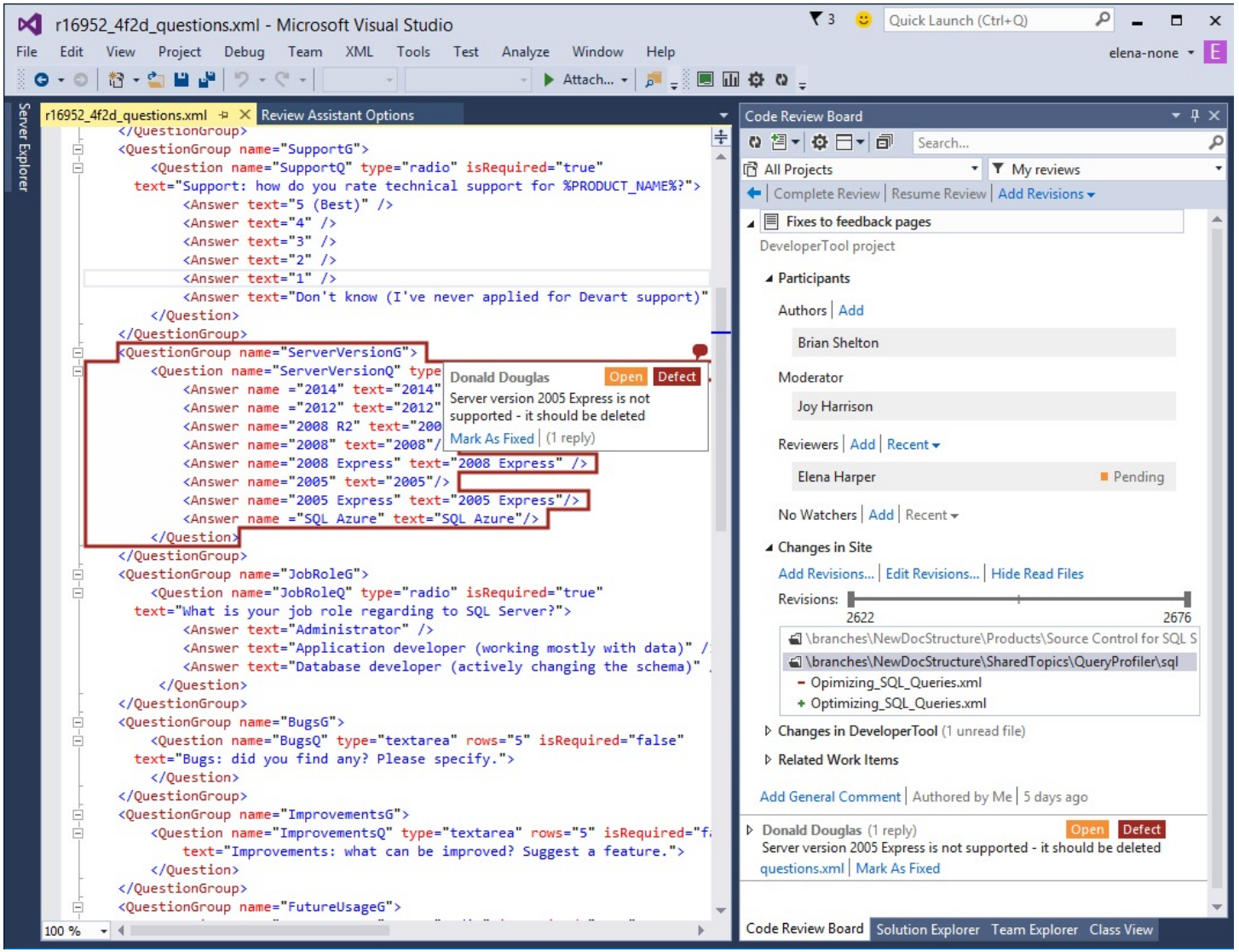
- Pour chaque workflow, on **compare le taux d'erreur avec l'incrément précédent**
- Réagir s'il y a un nombre disproportionné de fautes dans un artéfact
  - Reconcevoir dès le début devient une bonne alternative
- Métriques d'inspection
  - Taux d'inspection (diagramme/page révisé par heure)
  - Densité de fautes (fautes par KLOC inspectées)
  - Taux de détection de fautes (fautes détectées par heure)
  - Efficacité de détection de fautes (nombre de fautes majeures détectées par heure)

QUESTION

Qu'est-ce qu'une augmentation de 50% du taux de détection de fautes signifie ?

- Le niveau de qualité est à la baisse ?
- L'inspection est plus efficace ?

Revue de code



Critique des inspections

- Inspections peuvent s'avérer très efficaces
  - Fautes détectées tôt dans le processus
- Inspections moins efficaces si le processus est inadéquat
  - Logiciel à grande échelle doit être composé de plus petits sous-systèmes indépendants
  - Documentation du workflow précédent doit être complète et disponible
- Revue de code permet de détecter plus rapidement et plus rigoureusement les fautes
  - Réduction des coûts de maintenance jusqu'à 95%
  - Mais demande le temps nécessaire à allouer à la revue

Tester

Citations populaires

Utiliser un logiciel c'est l'exécuter dans le but d'atteindre l'objectif qui lui est destiné

« *Tester* un logiciel, c'est l'exécuter dans le but de le faire **échouer**. »

Glenford J. Myers'79

« Tester ne peut que montrer la **présence d'erreurs**, pas leur absence. »

Edsger W. Dijkstra

Qu'est-ce qu'un test ?

- Processus de trouver les différences entre
  - Le comportement attendu et
  - Le comportement observé

- Technique de détection de fautes qui tente de **faire échouer le logiciel** ou l'amener à un **état erroné**, de façon **planifiée**

- Un test est réussi s'il identifie des fautes ou démontre que le logiciel ne présente pas les fautes qu'il cherche à relever

Vérification basé sur l'exécution

- Processus d'**inférer** certaines propriétés du **comportement** du logiciel en se basant, en partie, sur le résultat obtenu de son **exécution** dans un **environnement connu** avec des **entrées choisies**

- Inférence
  - On a un rapport de fautes, le code source et, souvent, rien d'autre
- Environnement connu
  - Mais on ne connait *jamais* vraiment *tous* les paramètres de notre environnement
- Entrées choisies
  - Des fois, on ne peut pas fournir les entrées voulues
    - La **simulation** est alors nécessaire

QUESTION

Quels critères faut-il tester dans un logiciel ?

- S'il est correct
- S'il est utile aux utilisateurs ciblés
- S'il est fiable
- S'il est robuste
- Si sa performance est adéquate

- Tous les **facteurs de qualité** du logiciel s'appliquent

Les spécifications doivent être correctes

- Est-ce que la spécification suivante est correcte pour le tri ?

Input specification:	p : array of n integers, $n > 0$.
Output specification:	q : array of n integers such that $q[0] \leq q[1] \leq \cdots \leq q[n-1]$

- La fonction `trickSort` satisfait cette spécification :

```
void trickSort (int p[ ], int q[ ])
{
    int i;
    for (i = 0; i < n; i++)
        q[i] = 0;
}
```

Les spécifications doivent être correctes

- La spécification de tri corrigée :

Input specification:	p : array of n integers, $n > 0$.
Output specification:	q : array of n integers such that $q[0] \leq q[1] \leq \cdots \leq q[n-1]$ The elements of array q are a permutation of the elements of array p, which are unchanged.

- Les tests supposent que la spécification est correcte
  - Toujours **valider les spécifications** avant de tester

Méthodes formelles

Méthodes de vérification formelles

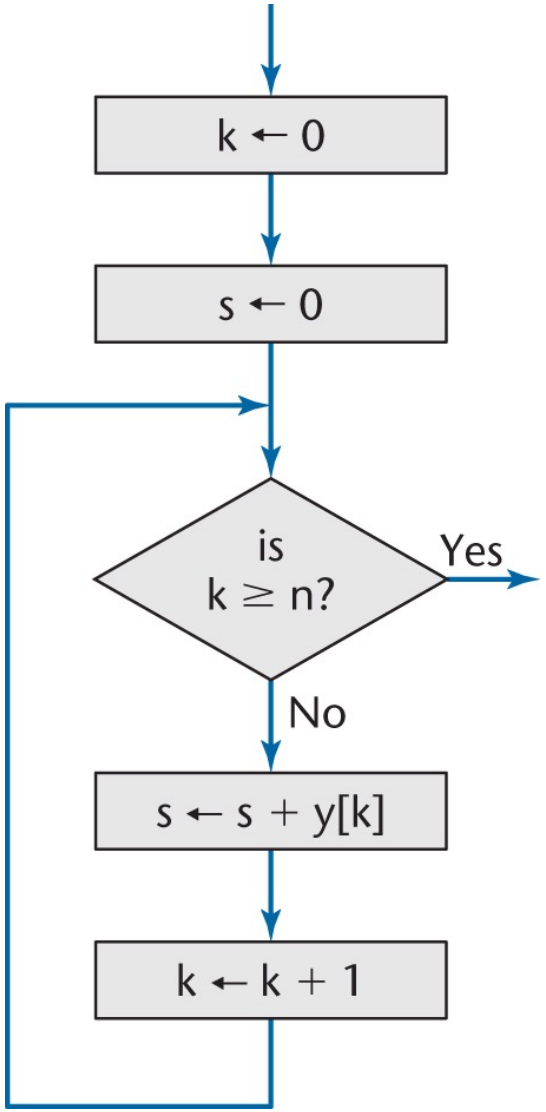
- Que calcule le code suivant ?

```
int k, s;
int y[n];
k = 0;
s = 0;
while (k < n)
{
    s = s + y[k];
    k = k + 1;
}
```

Comment prouver qu'il est correct ?

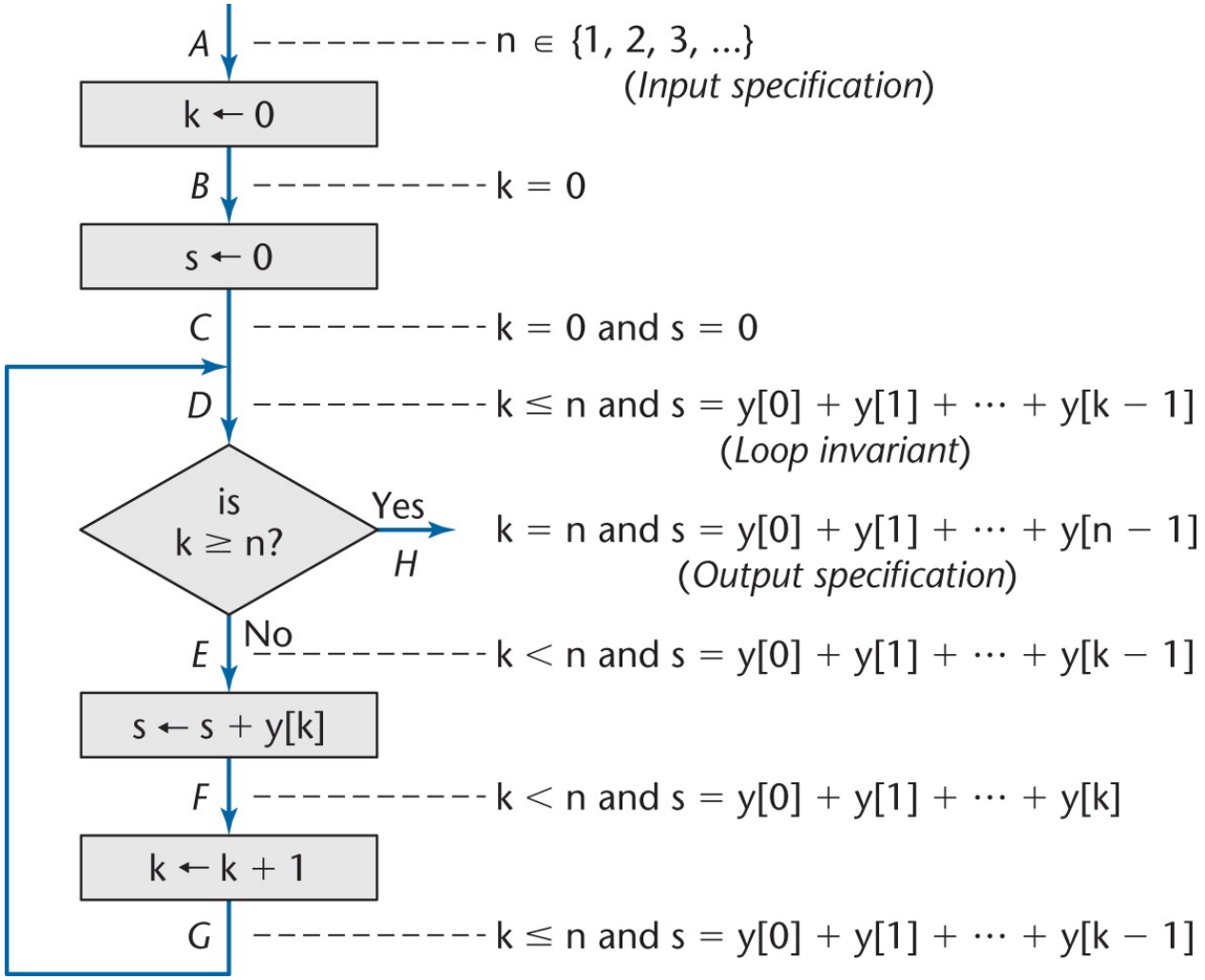
Graphe du flux de contrôle

```
int k, s;
int y[n];
k = 0;
s = 0;
while (k < n)
{
    s = s + y[k];
    k = k + 1;
}
```



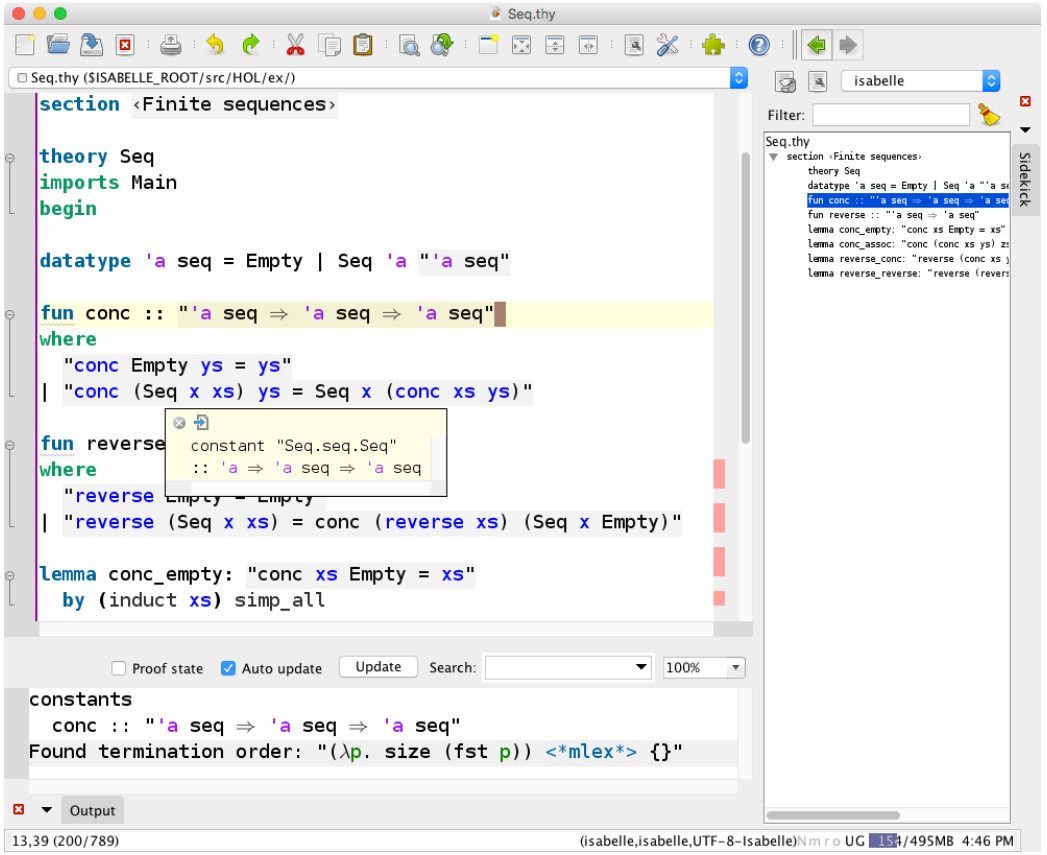
Exécution symbolique du programme

Preuve par induction sur n



Démonstration automatique de théorèmes

- Outils d'assistance aux preuves
  - Isabelle, Rodin
- Certification de code
- Appliqué dans la conception de circuits intégrés
  - Pentium, HP, AMD
- Programmation logique
  - 1er ordre ou supérieur
  - Model checking
  - Réécriture de termes



Technique de vérification peu répandue

- Démonstration d'exactitude sous-utilisée en génie logiciel
- Ingénieurs logiciel n'ont pas assez de connaissances mathématiques pour rédiger les démonstrations
  - Bien que, la plupart des informaticiens ont la capacité de l'apprendre
- Démontrer coûte trop cher pour être pratique
  - Viabilité économique est déterminé par une analyse coût-avantage
- Démontrer est difficile
  - Pourtant plusieurs logiciels non-triviaux ont été démontrés
    - Compilateurs
  - Outils d'assistance de preuves peuvent être très utiles

Tests structurels

Tester par rapport à quoi ?

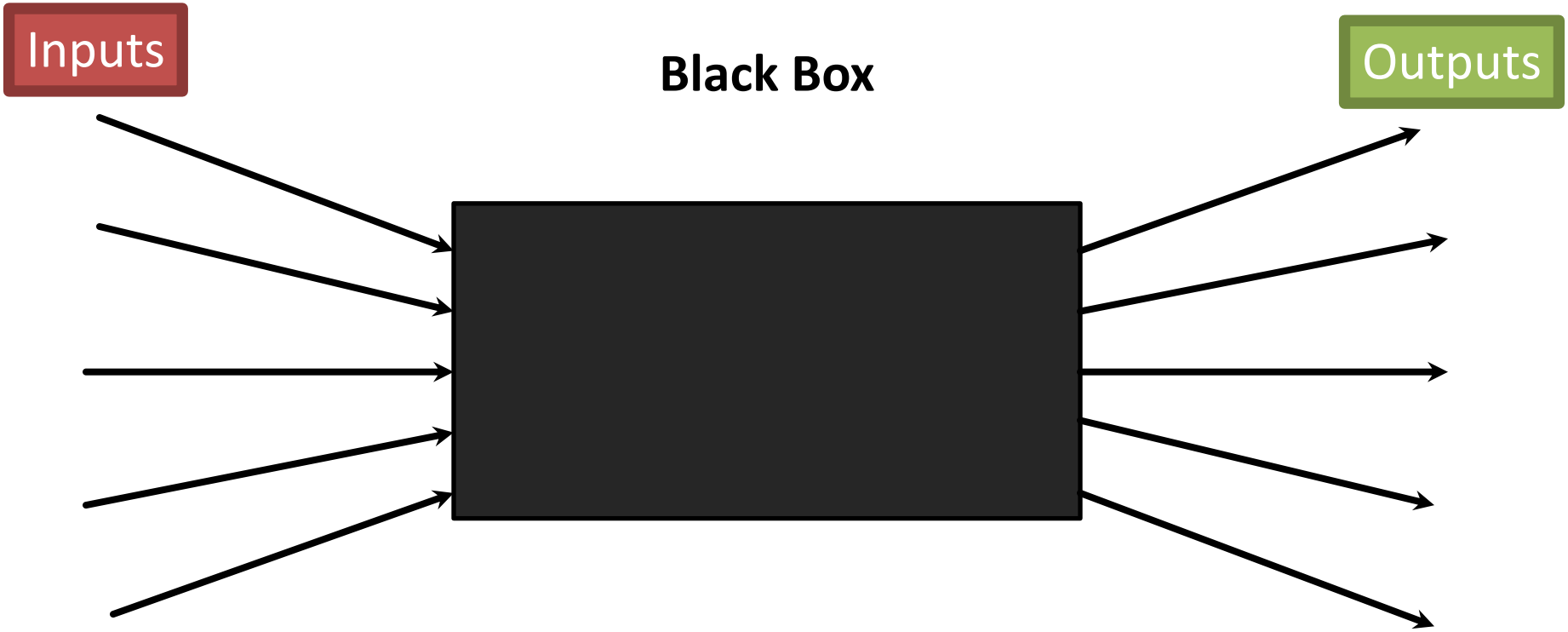
- Tester **par rapport aux spécifications**
  - Ignore le code: utilise les spécifications pour choisir les cas de test
  - Test à la boîte noire, dirigé par les donnés, par le I/O, par les fonctionnalités

- Tester **par rapport au code**
  - Ignore les spécifications: utilise le code pour choisir les cas de test
  - Test à la boîte blanche, dirigé par la logique, par la structure du code

Test à la boîte noire

Vérifie si le comportement externe du logiciel est conforme aux exigences

- **Test fonctionnel**



Processus d'un test à la boîte noire

- Planification
  - Identifier les **fonctions externes** à tester
  - Définir les **entrées** et **sorties** de chaque fonction
  - Établir les objectifs de qualité
    - Critère de terminaison et suffisance des **cas de test**
- Exécution
  - **Exécuter** chaque cas de test sur chaque fonction
  - Observer le comportement
  - Enregistrer les problèmes rencontrés
  - Noter l'information d'exécution qui servira à corriger les défauts
- Analyse
  - **Comparer** les résultats aux sorties attendues
  - S'assurer que **l'oracle** est adéquat
  - Corriger les défauts découverts

Sélection des cas de test

- Comment déterminer les entrées et sorties d'une fonction?
- Impossible de tester exhaustivement tous les cas possibles

- L'art du test est un problème d'optimisation
  - Sélectionner un **petit ensemble** de cas de test gérable, afin de
  - **Maximiser** les chances de détecter une faute, tout en
  - **Minimisant** les chances de gaspiller un cas de test

- Chaque cas de test doit détecter une faute unique qui ne serait pas détectée par un autre

QUESTION

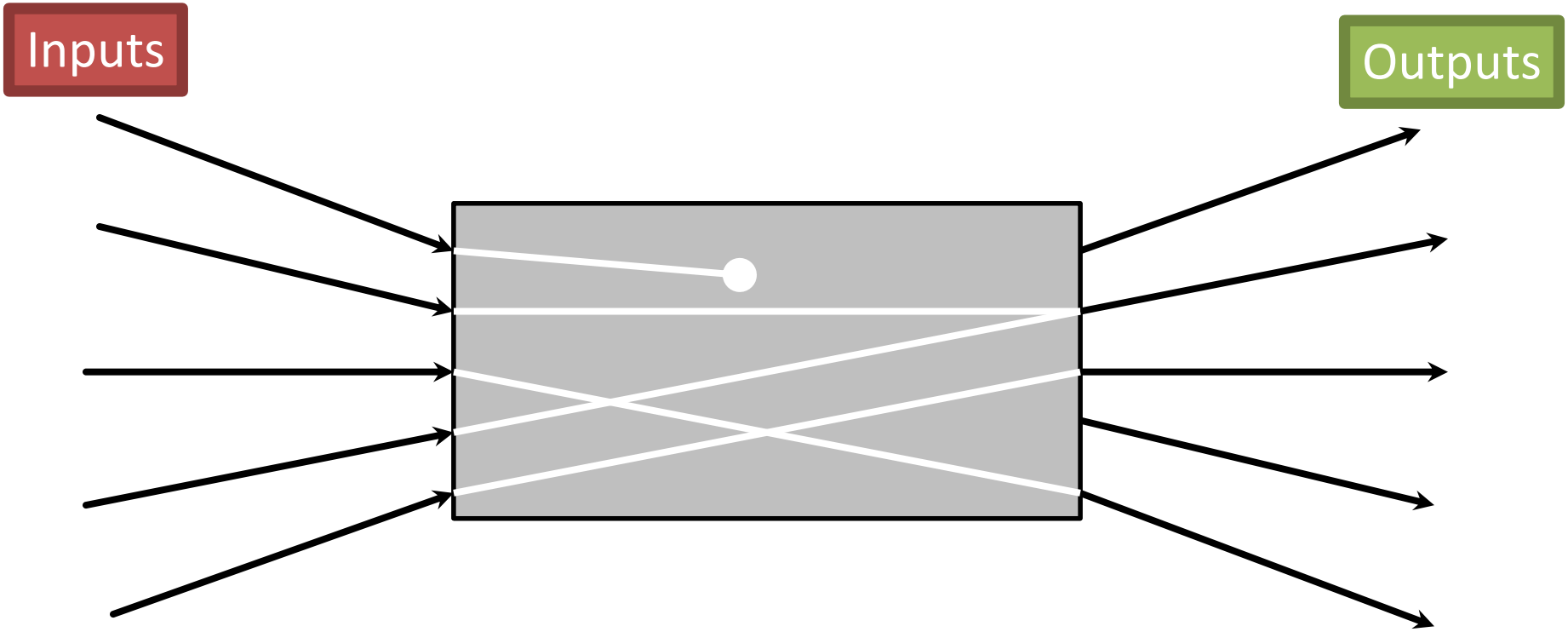
Un programme lie 3 entiers naturels. Ils représentent les longueurs des côtés d'un triangle. Le programme affiche trois messages possibles: le triangle est scalène, isocèle ou équilatéral. Identifiez 5 cas de tests pour ce problème.

- Scalène: $A \neq B \neq C$
  - $A = 2, B = 3, C = 4$
- Équilatéral: $A = B = C \neq 0$
  - $A = 2, B = 2, C = 2$
- Isocèle: $A = B$ ou $A = C$ ou $B = C$
  - $A = 2, B = 2, C = 3$
  - $A = 2, B = 3, C = 2$
  - $A = 3, B = 2, C = 2$
- Invalide: $A + B < C$ ou $A + C < B$ ou $B + C < A$ ou $A = B = C = 0$

Test à la boîte blanche

Vérifie si l'implémentation du logiciel est correcte

- **Test structurel**



Tester tous les chemins possibles du code

- Tester chaque **expression**
  - Exécuter un ensemble de cas de test tel que chaque expression soit exécutée au moins une fois
- Tester chaque **branche** du code
  - Exécuter un ensemble de cas de test tel que chaque branche soit exécutée au moins une fois
    - Conditions, boucles, polymorphisme, multithread
- Tester chaque déclaration et utilisation de **variable**
  - Pour chaque variable, tester les valeurs qu'elle peut avoir pour chaque expression dans laquelle elle est utilisée
- On peut détecter des **chemins inatteignables** (code mort)
  - Indice de présence de faute

```
if (k < 2)
{
  if (k > 3)
       ↑
    x = x * k;
}
```

Exemple

```
private static double[] computeRoots(double a, double b, double c) {
    double[] roots;
    double delta = Math.pow(b, 2) - (4 * a * c);
    if (delta > 0)
    {
        roots = new double[]
        {
            (-b + Math.sqrt(delta)) / (2 * a),
            (-b - Math.sqrt(delta)) / (2 * a)
        };
    }

    else if (delta == 0)
    {
        roots = new double[]
        {
            -b / (2 * a)
        };
    }

    else
    {
        roots = new double[0];
    }

    return roots;
}
```

Exemple

```
private static double[] computeRoots2(double a, double b, double c) {
    List<Double> possibleRoots = new ArrayList<Double>();
    double tolerance = Double.MIN_NORMAL;
    for (double x = -3; x < Double.MAX_VALUE; x += tolerance)
    {
        double sol = a * Math.pow(x, 2) + (b * x) + c;
        if(Math.abs(sol) <= tolerance)
        {
            possibleRoots.add(x);
        }
    }

    double[] roots = new double[possibleRoots.size()];
    for (int i = 0; i < roots.length; i++)
    {
        roots[i] = possibleRoots.get(i);
    }
    return roots;
}
```

Cas de test pour test structurels

- Programme de 50-100 LOC
- 1 boucle (1-20 itérations), 4 conditions imbriquées

```
do
   if (…) then
       if (…) then
          if (…) then … else …
       else …
          if (…) then … else …
   else …
while (i < 20)
```

- Les chemins d'exécution possibles : $\sum_{i=1}^{20} 5^i$
- 3 174 ans pour tester tous les chemin à raison d'un chemin par milliseconde !

Tests exhaustifs ?

- En général, tester un programme de façon exhaustive est **impossible**

- Il est possible d'utiliser des **tests aléatoires**, mais leur efficacité est faible pour tester le comportement attendu

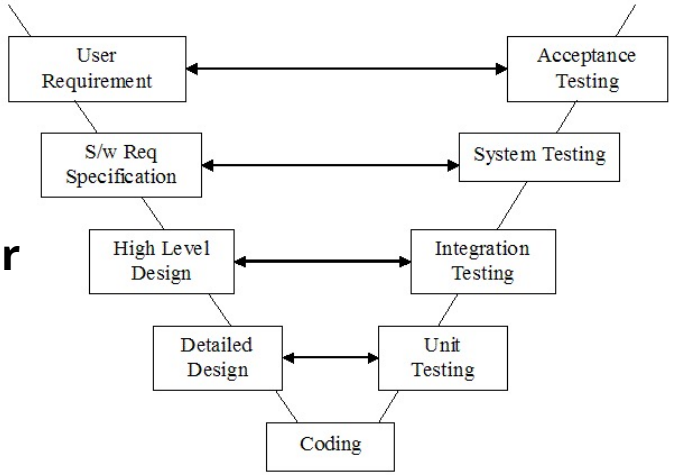
- Une meilleure approche :
  1. Déterminer un ensemble de tests fonctionnels
  2. Qui seront complémentés des tests structurels

Tester en présence de dépendances

- La plupart des modules requièrent d'autres modules pour leur bon fonctionnement
- Idéalement, chaque module devrait être testé en **isolation**
  - Débogage facilité
  - Tests plus robustes en présence de **changements** dans le système
  - Permet une meilleure organisation des tests
- Comment isoler le comportement d'un module particulier?
  - Par la création d'un « faux » **module synthétique** qui joue le rôle de la dépendance pour les tests effectués
  - Par exemple, le faux module peut retourner des **valeurs prédéterminées** lorsqu'invoqué

Types de tests

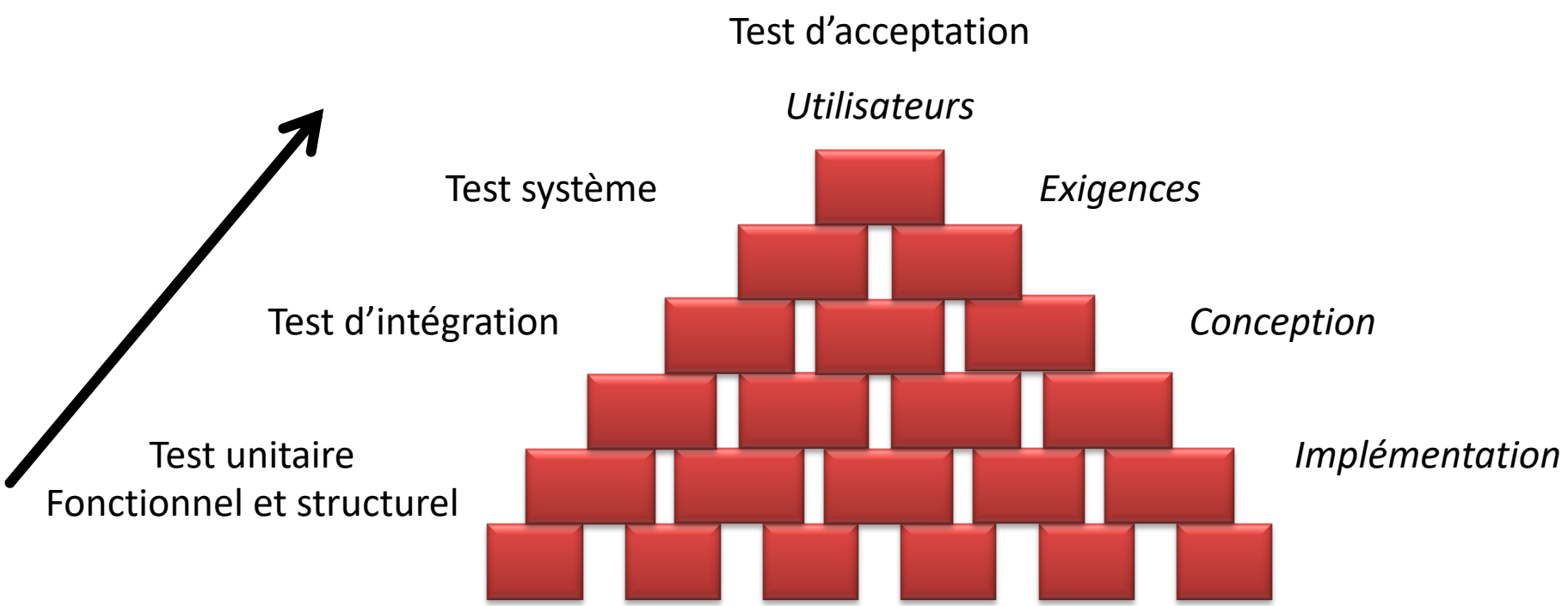
Workflow de test



Test à chaque phase

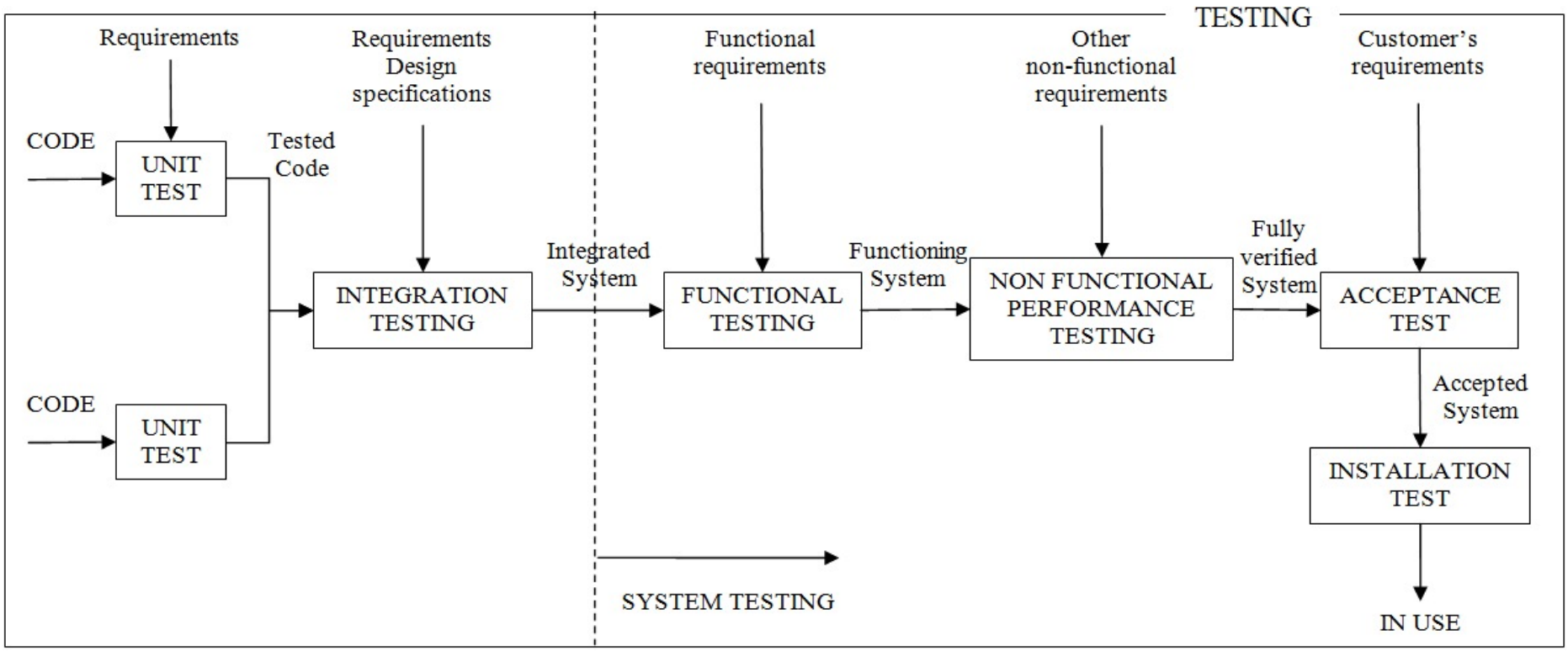
- Exigences
  - Tous les artéfacts du logiciel produits doivent **provenir** des exigences
  - Client révise les exigences
- Analyse
  - Spécifications **révisées conjointement** avec un expert du client et l'équipe d'analyse
- Conception
  - **Révisé** par les développeurs et l'équipe d'assurance qualité
- Implémentation
  - **Test unitaire**: chaque composant implémenté doit être testé dès que complété
  - **Test d'intégration**: après chaque itération, combiner les composants et tester
  - **Test système**: tester le système au complet
  - **Test d'acceptation**: par le client à chaque livraison

Types of de tests



Tests **Cibles de la V&V**

Fil conducteur

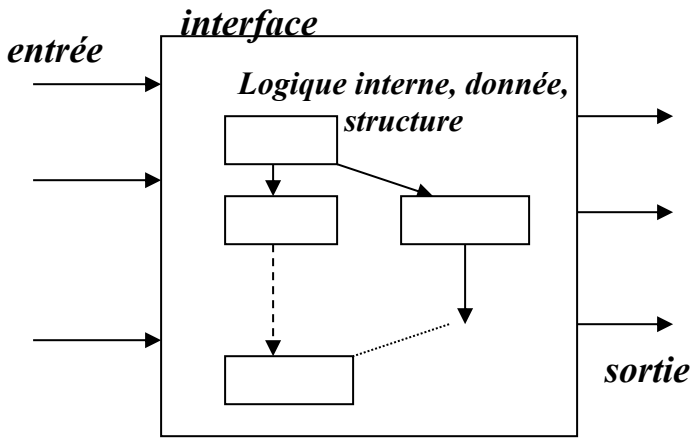


Tests unitaires

Vérifie chaque module individuellement

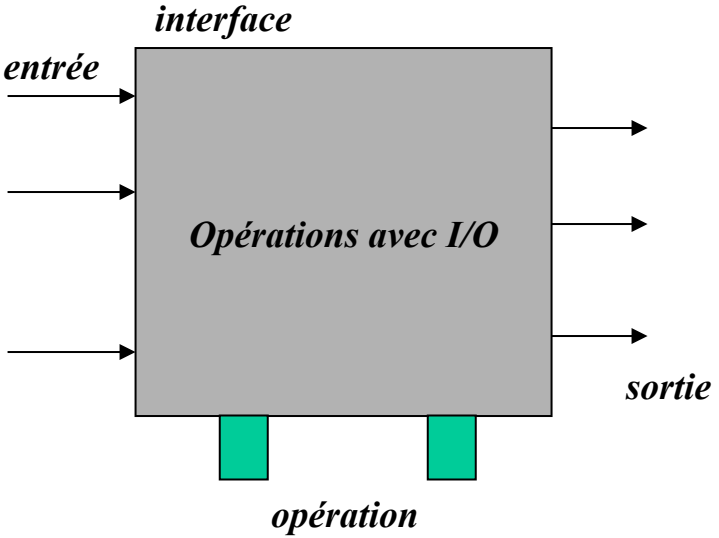
Boîte blanche

- Test structurel
- Dépend du code



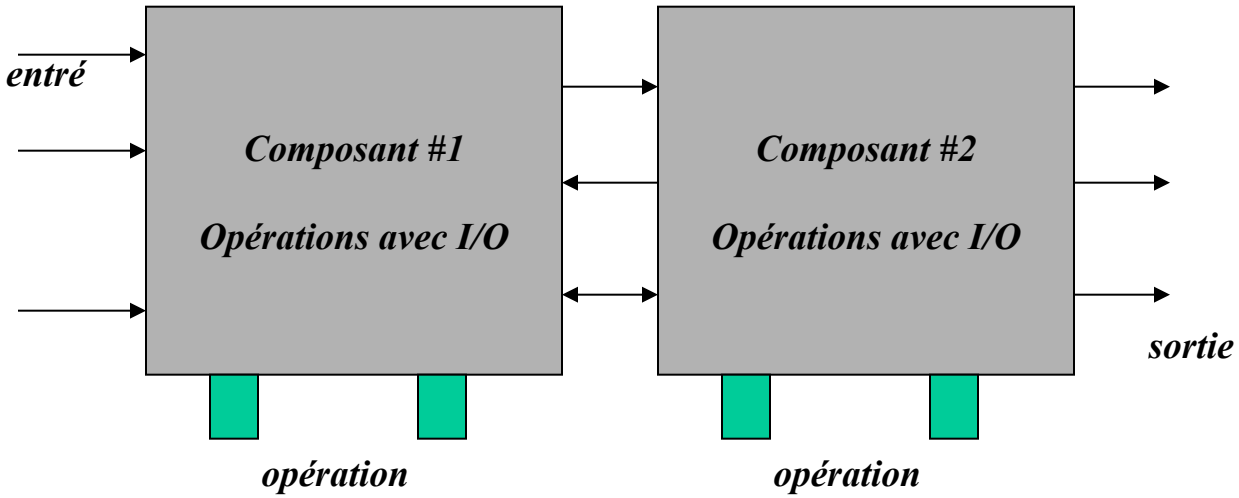
Boîte noire

- Test fonctionnel
- Dépend des entrées et sorties



Tests d'intégration

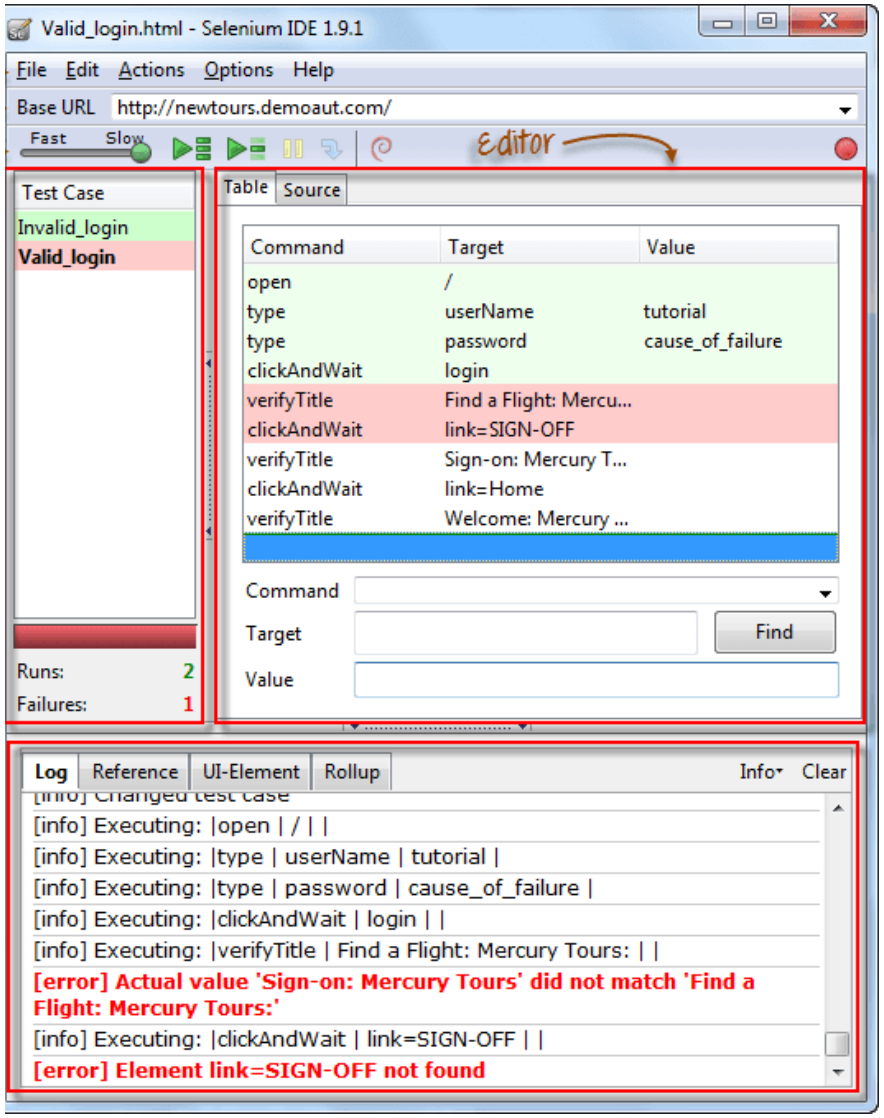
- Vérifie les interactions entre plusieurs modules
- Utilisé lors de l'ajout de nouveaux modules au groupe de modules (testés) déjà existants
- Attention particulière lors des test du GUI
  - Tester les événements générés
    - Cliques, touches, mouvements



Tests fonctionnels

- L'équipe AQ doit faire une approximation des tests d'acceptation
- Assure que le code est **conforme aux exigences**
  - Vérifie que toutes les fonctionnalités sont présentes
- Établit que le logiciel est **complet**, **précis** et **adéquat**
- Fonctionnalité souvent réalisée par une combinaison de **méthodes** d'après les **diagrammes de séquence**
- Tester du point de vue de l'utilisateur
  - Tester **chaque CU**
  - **Test à la fumée** pour tester les fonctionnalité du produit complet
  - **Test big-bang** vérifie que le système **en entier** est conforme aux exigences fonctionnelles

Tests fonctionnels automatisés



Test fonctionnel vs. structurel

Fonctionnel

- Vérifie l'adéquation input-output d'un algorithme
- Requière **moins de ressources**
- Ne permet pas d'identifier les cas où plusieurs erreurs s'annulent pour produire le bon résultat
- Ne permet pas d'assurer la couverture du code parles tests
- Ne permet pas d'évaluer la qualité du code

Structurel

- Vérifie l'implémentation d'un algorithme de façon précise
- Permet d'évaluer la quantité de code non-testé
- Permet d'évaluer la **qualité du code** et sa conformité aux standards
- Assure la couverture du code
- Utilise plus de ressources

Tests non-fonctionnels

- Vérifie que le système **en entier** est conforme aux exigences non-fonctionnelles
- **Test de performance**
  - Test de stress, de volume, de sécurité, de fiabilité
- Toutes les **contraintes** doivent être vérifiées
- Toute la **documentation** doit être vérifiée
  - Exactitude, conformité aux standards, cohérent avec la version courante du logiciel

Test d'acceptation = Validation

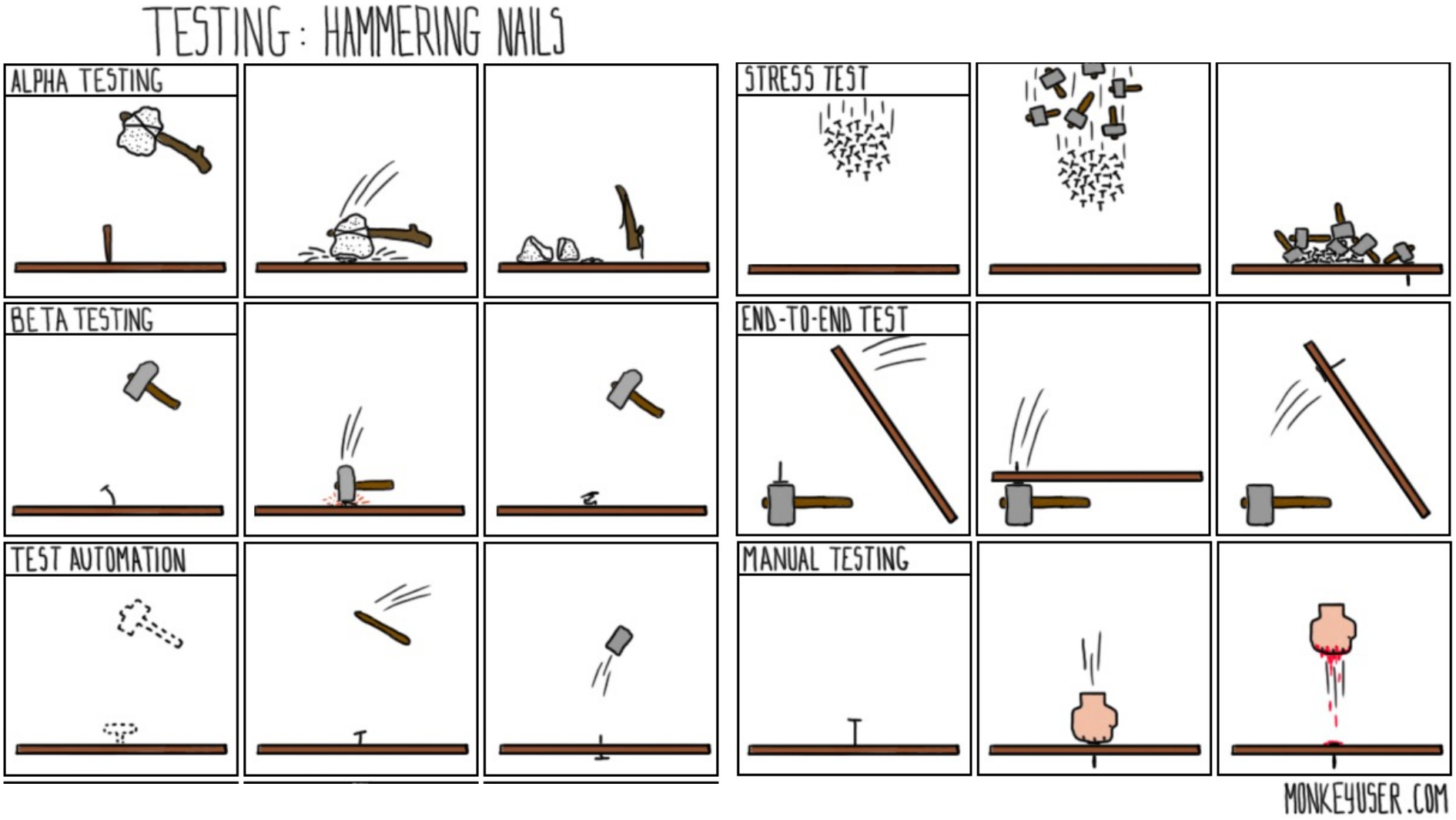
- Client détermine si le logiciel satisfait ses besoins
  - Exactitude
  - Robustesse
  - Performance
  - Documentation
- Différence entre les tests systèmes et d'acceptation est sur le jeu de données
  - Système: sur des **données fictives**, dans un **environnement contrôlé**
  - Acceptation: sur des **données réelles** dans l'**environnement d'utilisation réel**

Tests α et β

- Test Alpha
  - D'abord tester le nouveau logiciel dans un **environnement contrôlé**
  - Quand la première ronde de correction des bogues est complétée, le produit va en test Beta
- Test Beta
  - Les tests du logiciel révisé sont effectués par des utilisateurs dans des **conditions normales d'utilisation**

- Pensez aux proportions quand Gmail est testé

Types de tests



QUESTION

Devrait-on programmer les tests de son propre code ?

- Programmer est constructif ; tester est destructif
  - La même personne ne peut donc pas faire les deux
- Risque de partialité
  - Si j'ai programmé la méthode, je connais toutes les hypothèses explicites (ex: types) et tacites (intervalle des valeurs)
  - Si j'ai écrit les cas de test, je connais les cas que je vérifie à l'avance
- Je peux quand-même exécuter les tests déjà présents quand je programme!

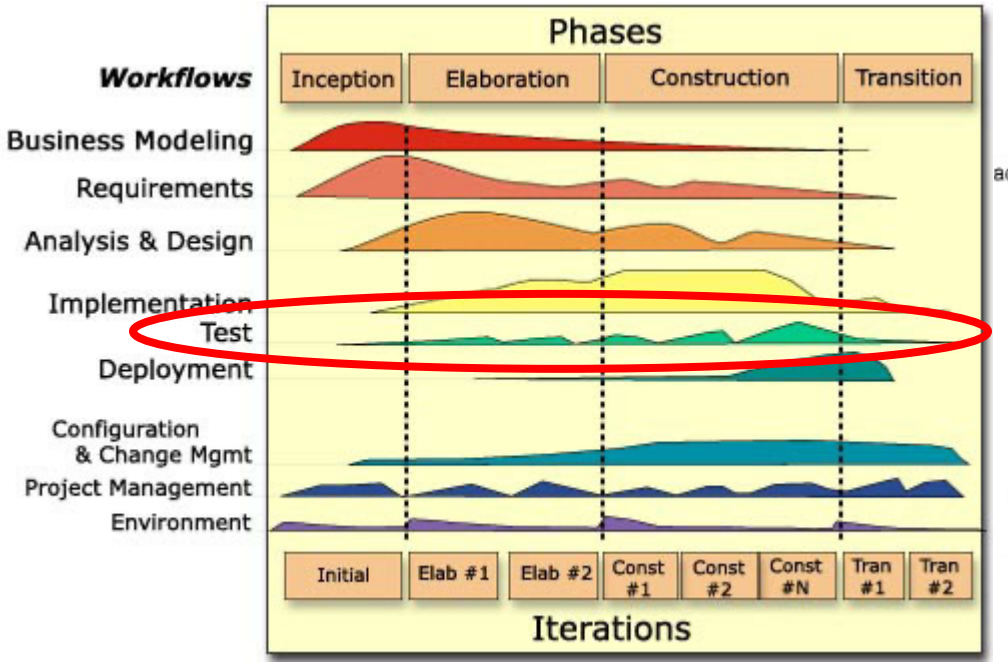
Qui devrait tester alors ?

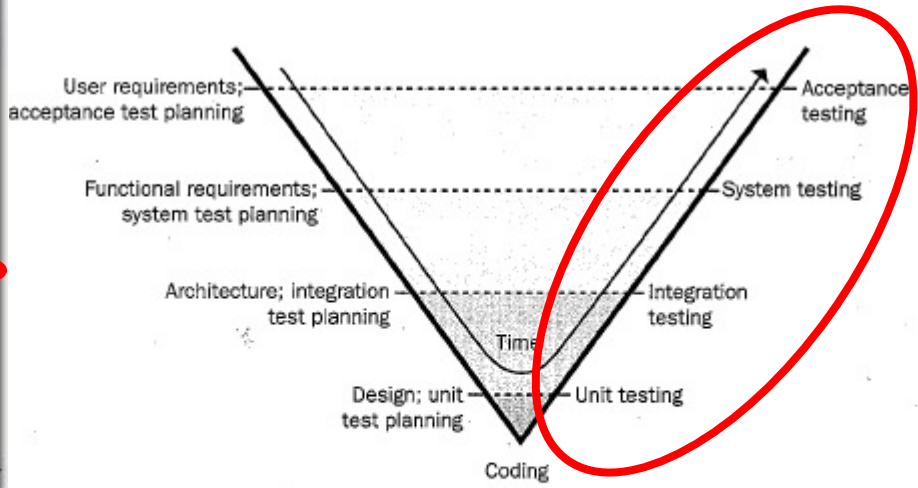
- Le programmeur effectue des tests informels de son code
- L'équipe AQ effectue des tests systématiques
- Le programmeur corrige les modules qui ont échoué

- Tous les tests doivent
  - Être planifiés à l'avance
  - Prévoir ce qui est attendu
  - Être conservés par la suite

Quand tester ?

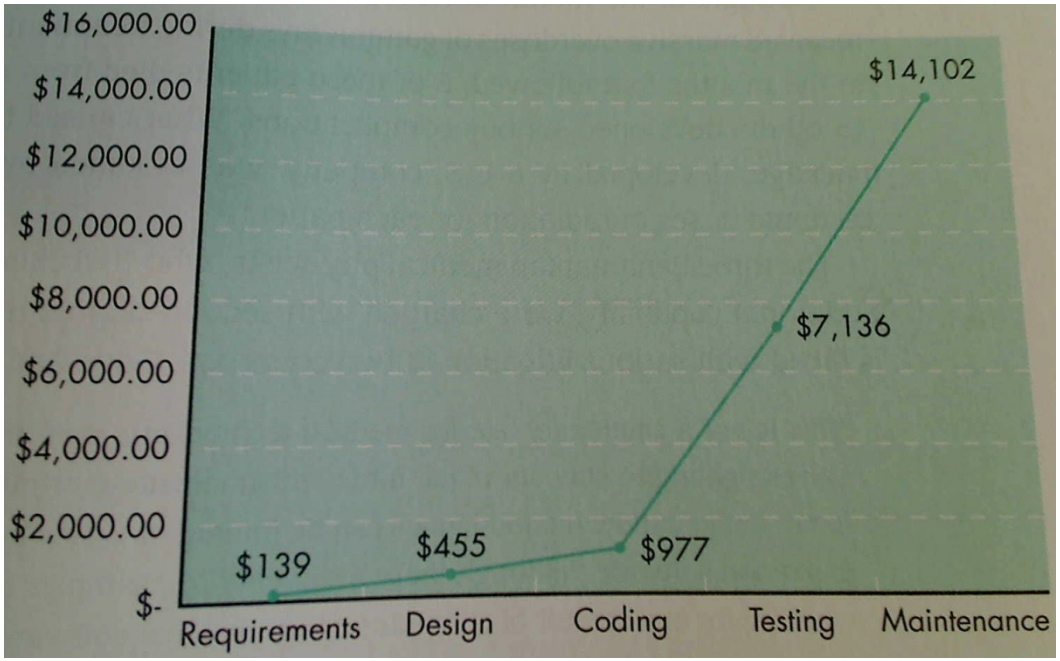
- Les tests peuvent être utilisés durant toutes les étapes du développement
  - Avant de débuter l'implémentation
  - Durant le développement
  - Après que le logiciel soit implémenté





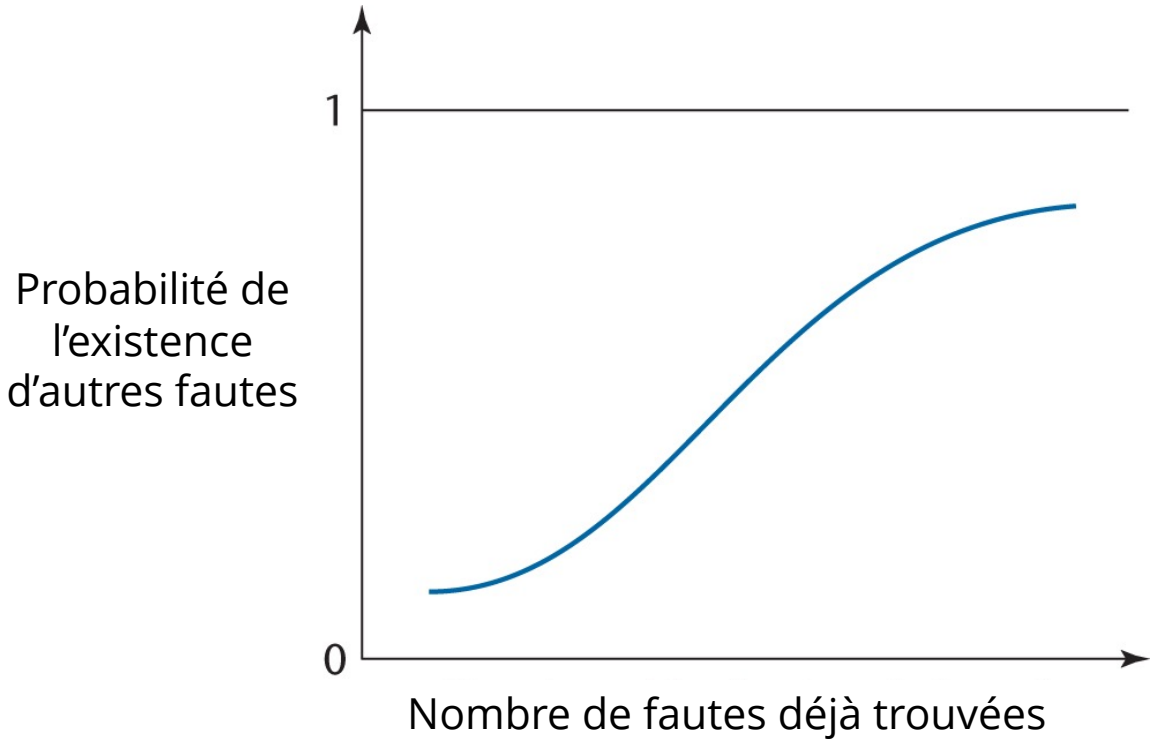
Coûts

- Les tests représentent une partie importante du développement
  - Jusqu'à 33% du budget
  - Jusqu'à 27% du temps de développement
- Coût relatif à la correction d'erreurs



Quand recoder plutôt que corriger ?

- Quand un artefact a trop de fautes
  - Moins dispendieux de reconcevoir et recoder
- Plus on trouve de fautes, plus il est probable qu'il y aura plus de fautes



Combien de fautes doit-on trouver ?

- Pour chaque artéfact, le gestionnaire doit **prédéterminer le nombre maximal de fautes** trouvées pendant les tests
- Si ce nombre est atteint
  - Jeter l'artefact
  - Reconcevoir le composant
  - Ré-implémenter le code
- Le nombre maximal autorisé de fautes trouvées **après la livraison** est **ZÉRO**
  - Idéalement…

QUESTION

Quand s'arrêter de tester ?

- Avant le déploiement ?
  - Besoin de vérifier que l'installation est correcte
- Après le déploiement ?
  - Demandes de changements peuvent arriver : recommencer le cycle de développement
- Donc, théoriquement, les tests ne s'arrêtent seulement quand le logiciel est **retiré**
- Idéalement, s'arrêter quand
  - Tous les tests planifiés passent avec succès: toutes les fautes sont corrigées

En pratique, s'arrêter quand…

- X% des cas de tests sont réussis
  - Mauvaise idée
- Il y a moins de X défauts restants
  - Mauvaise idée
- Il n'y a plus de budget disponible pour les tests
  - Pas le choix
- L'échéancier de la livraison est arrivé
  - Pas le choix
- Il ne reste plus de défauts bloqueurs, critiques ou majeurs
  - Acceptable