Génie logiciel

Vérification & Validation

Louis-Edouard LAFONTANT

geodes

Université de Montréal

Vérification & Validation

Nous avons finalement construit notre solution.
Répond-elle vraiment aux besoins du client et des utilisateurs?
Est-on certain de son comportement?

- **Il faut faire des tests pour obtenir cette garantie.**

To Err Is Human

- L’erreur est humaine, **les programmes sont développés par des humains**
- Toujours assumé que mon code contient des erreurs
- L’activité de **V&V est essentielle** au développement de logiciels de **qualité**

Qualité

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

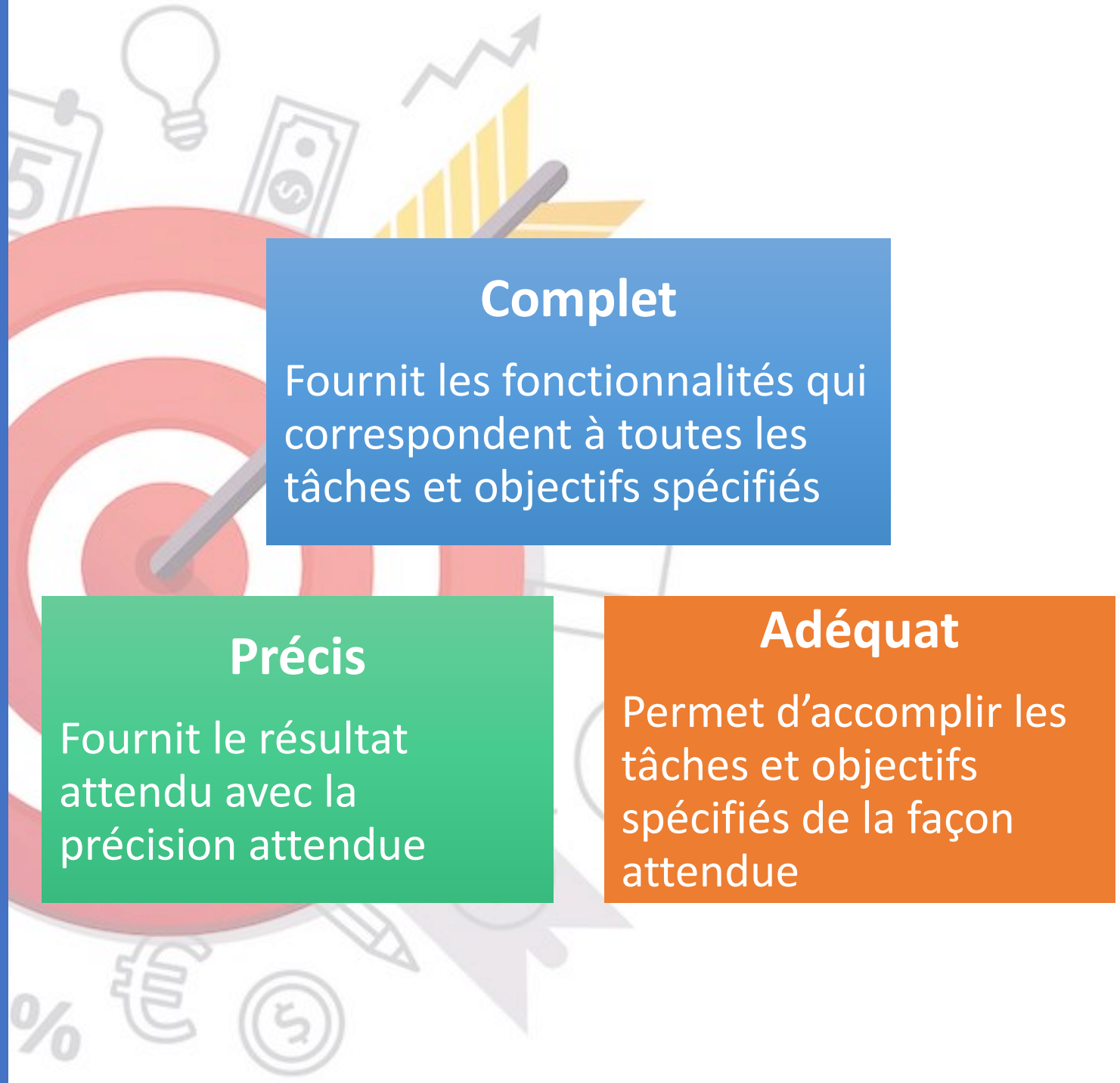
- Pas une question d'« excellence »
- Mesure si le logiciel satisfait les besoins

Qualité du logiciel

Caractéristique mesurable du logiciel

Pertinence fonctionnelle

Fonctionnalités du logiciel correspondent aux besoins exprimés et tacites du client et des utilisateurs.



Complet

Fournit les fonctionnalités qui correspondent à toutes les tâches et objectifs spécifiés

Précis

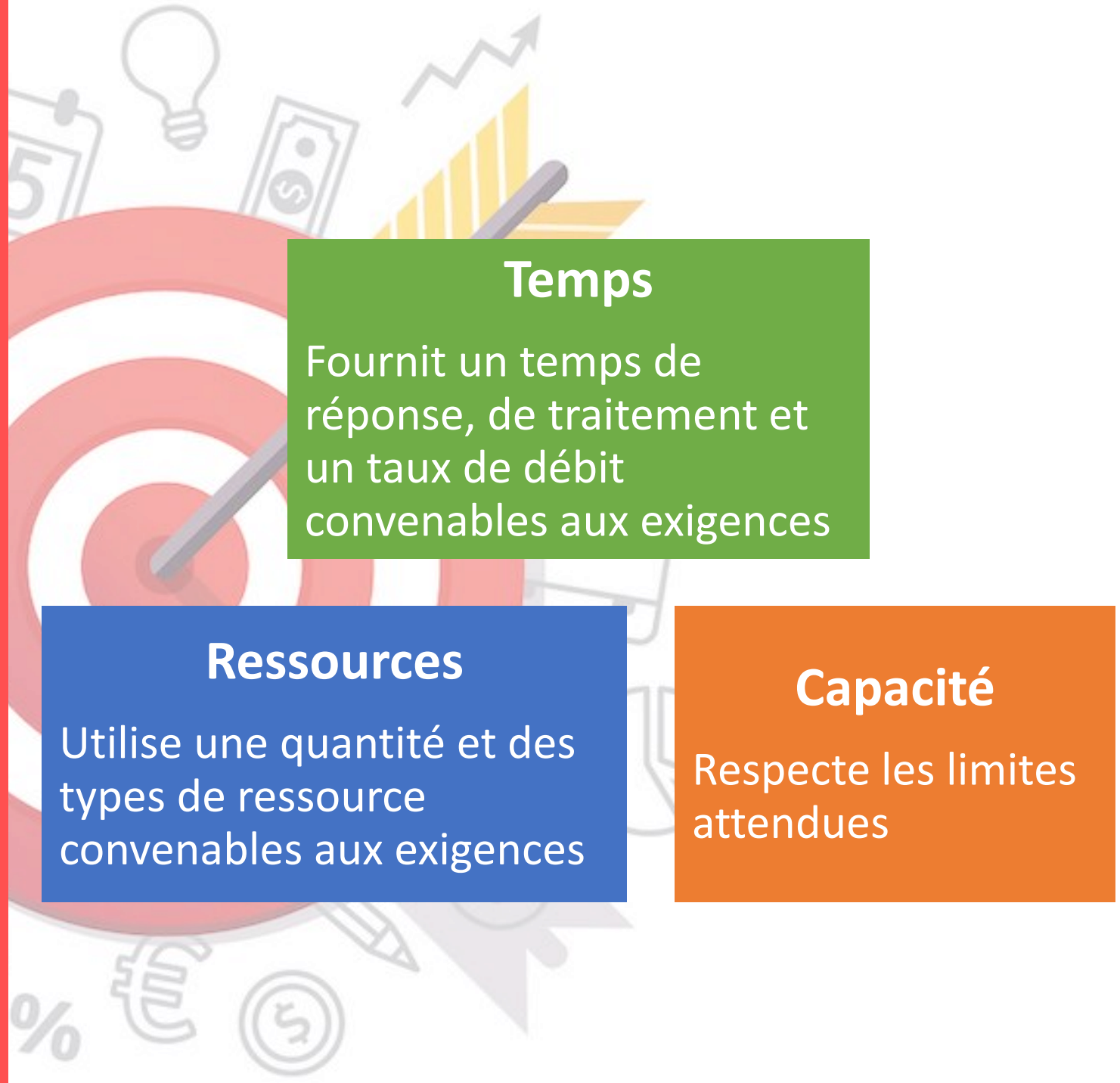
Fournit le résultat attendu avec la précision attendue

Adéquat

Permet d'accomplir les tâches et objectifs spécifiés de la façon attendue

Performance

Logiciel présente des performances relatives aux ressources utilisées.



Temps

Fournit un temps de réponse, de traitement et un taux de débit convenables aux exigences

Ressources

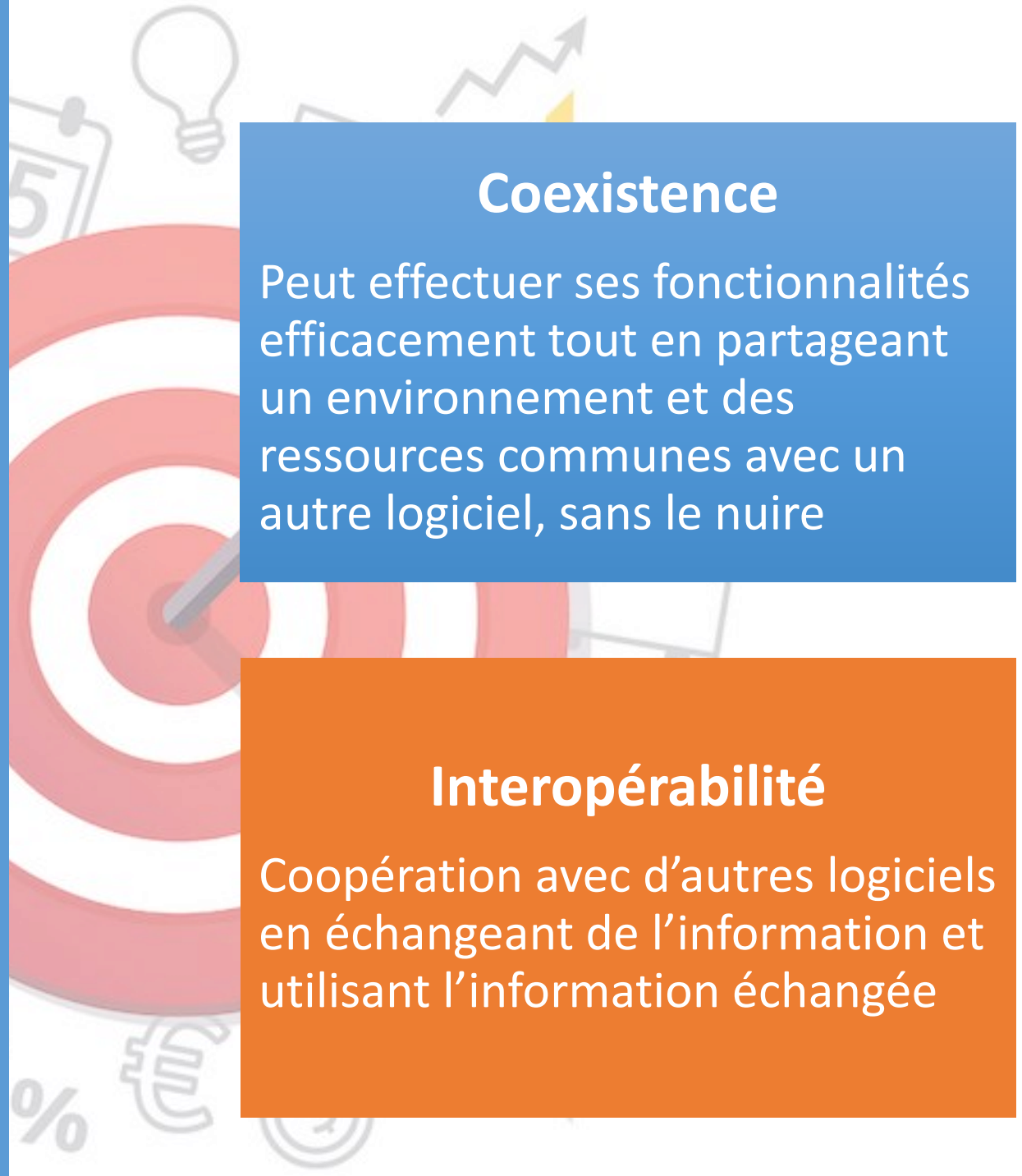
Utilise une quantité et des types de ressource convenables aux exigences

Capacité

Respecte les limites attendues

Compatibilité

Deux modules 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 de manière efficace et efficiente par ces utilisateurs (humain), en procurant une expérience satisfaisante.

Apprentissage

Facilité de prendre en main du logiciel et d'accomplir des taches de base

Efficience

Rapidité avec laquelle on accomplit une tache après l'apprentissage

Mémorisation

Permet à l'utilisateur de retrouver ses repères et mécanismes après une période d'inutilisation.

Erreur

Empêche l'utilisateur de commettre des erreurs et lui permet de récupérer facilement en cas d'erreur

Esthétique

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

Robustesse

Fonctionne tel que prévu malgré la présence de fautes matériel ou logiciel

Récupération (potentiel)

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 utilisateurs aient les accès requis conformément à leur type/rôle 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 donnée

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'adapter aux changements dans l'environnement et dans les exigences.

Modularité

Composé de composants distincts afin de minimiser l'impact d'un changement sur un composant

Réutilisabilité

Composant peut être utilisé dans plusieurs systèmes et contextes, ou pour construire de nouveaux composants

Analyse (potentiel)

Capacité d'évaluer l'impact d'un changement, de diagnostiquer les défauts, ou identifier les parties à modifier

Changement (potentiel)

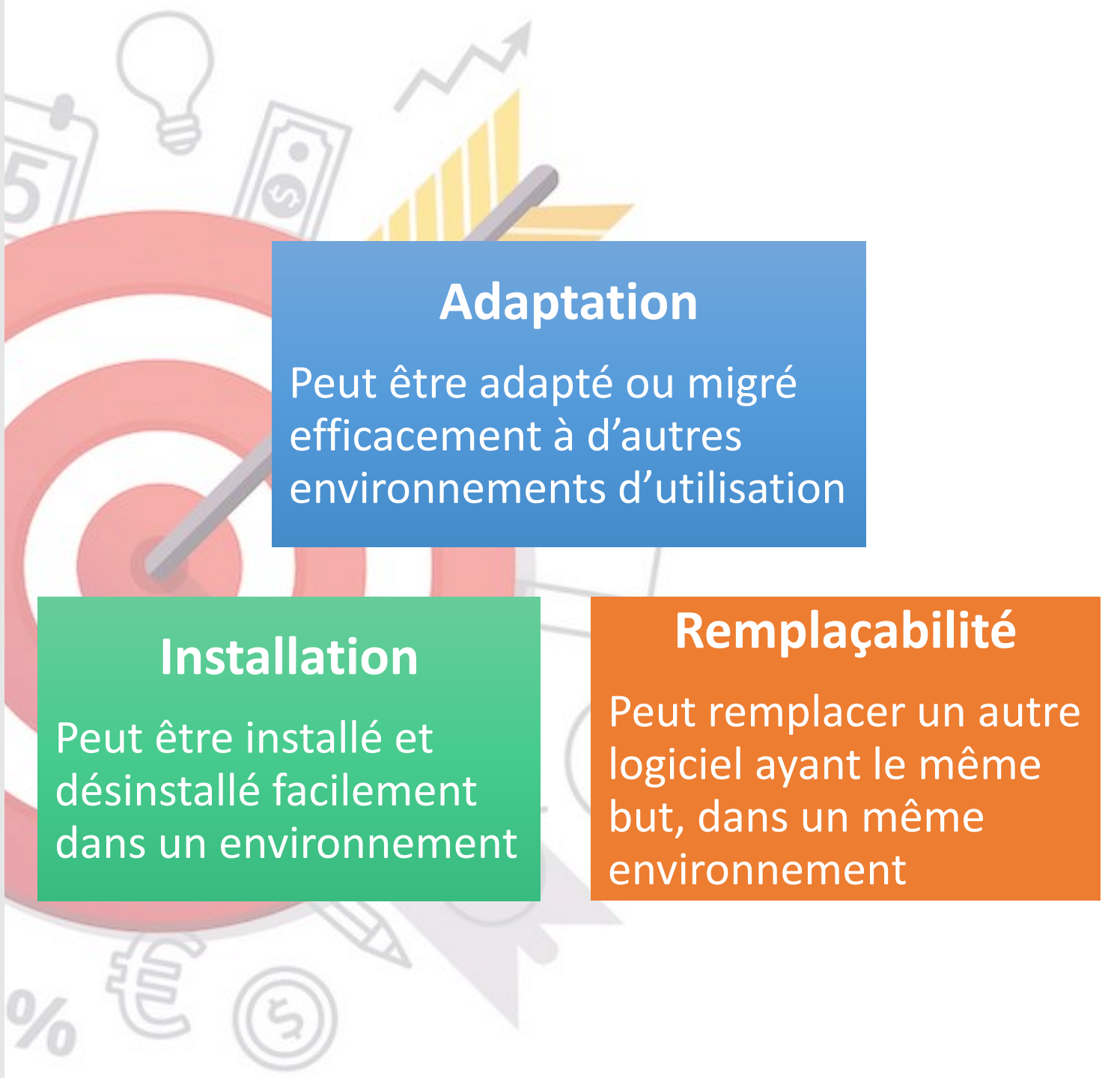
Peut être modifié facilement sans introduire des défauts ou dégrader la qualité

Test (potentiel)

Permet d'établir des critères pour tester un composant et tester pour déterminer si ces critères sont satisfaits

Transférabilité

Logiciel peut être transféré d'un environnement (logiciel ou matériel) à un autre.



Adaptation

Peut être adapté ou migré efficacement à d'autres environnements d'utilisation

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

Vérification et validation

Assurance qualité logiciel (AQ)

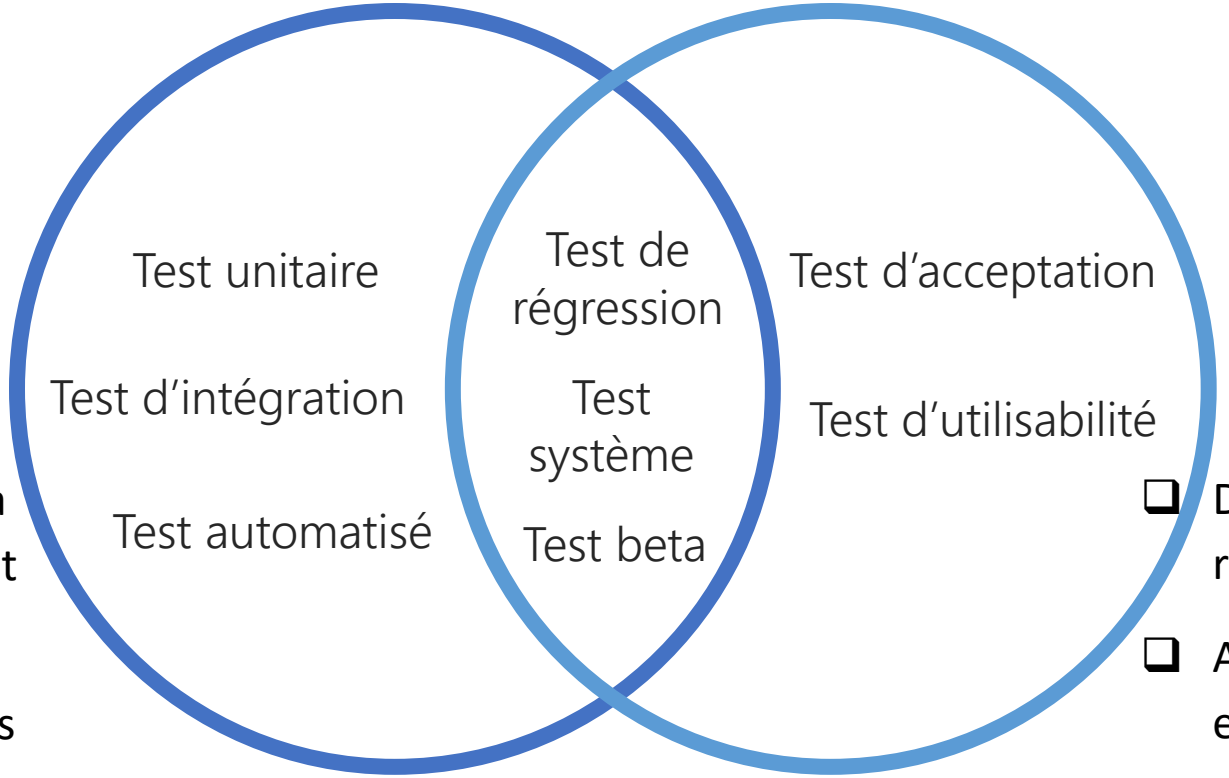
- Les membres de l'équipe AQ doivent s'assurer que les développeurs fournissent un travail de haute qualité
  - À la fin de **chaque workflow**
  - AQ doit être appliquée au processus de développement aussi
- Équipes AQ et de développement sont horizontales
  - **Aucun n'a de pouvoir sur l'autre**
- 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

Développement de logiciel fiable

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

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

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

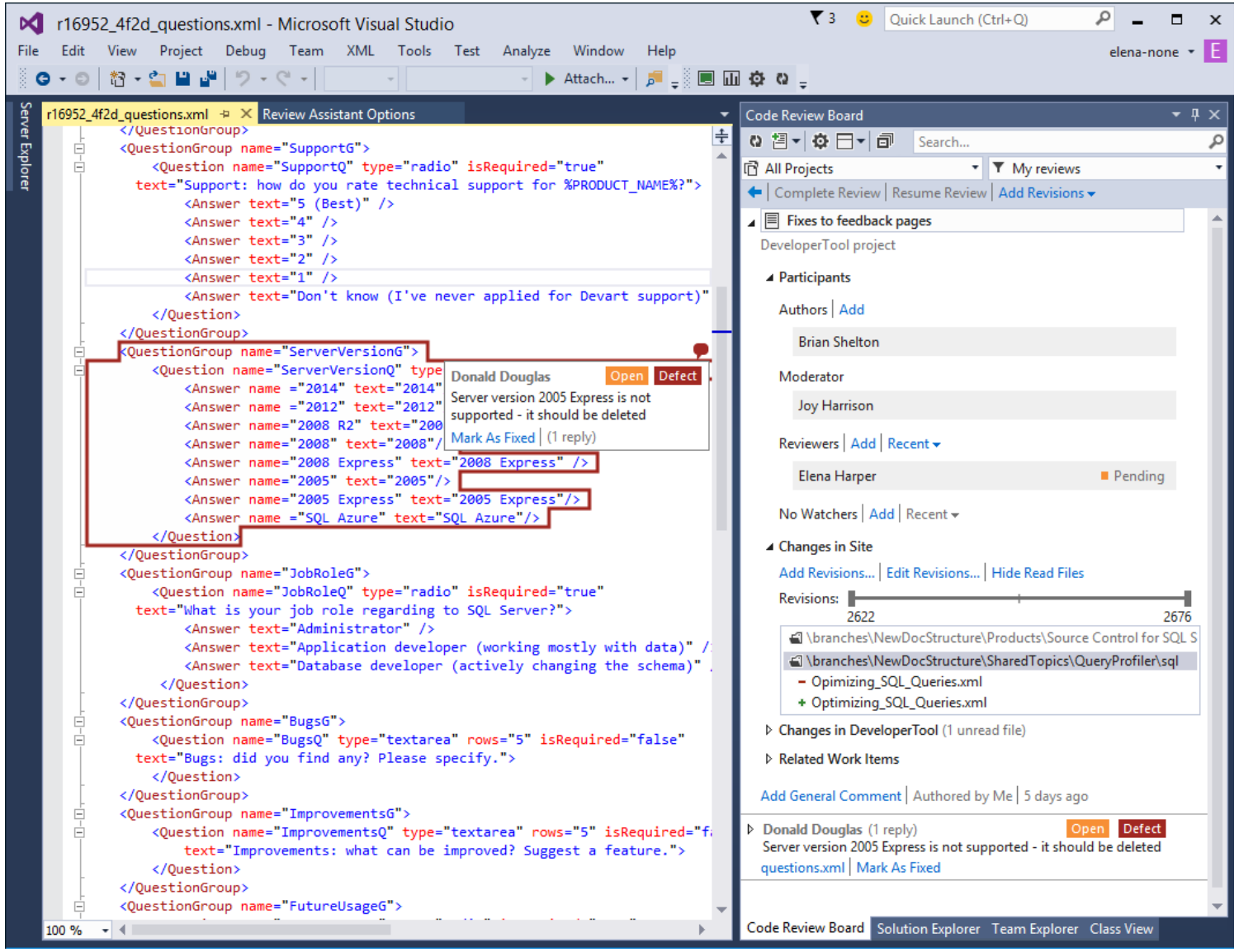
Statistiques d’inspection

- Pour chaque workflow, on **compare le taux d’erreur avec l’incrément précédent**
- Réagir s’il y a un grand nombre de fautes dans un artéfact

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)

Revue de code (*code review*)



Tester

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

« Attention aux bogues dans ce code; je n'ai seulement prouvé qu'il était correct, mais je ne l'ai jamais essayé. » *Donald Knuth*

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

Les spécifications doivent être correctes

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

Specification en entrée:
  p: tableau de n entiers, n > 0

Spécification en sortie:
  q: tableau de n entiers, t.q. q[0] ≤ q[1] ≤ … ≤ q[n-1]

La fonction **`trickSort`** (pas adéquat) 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 :

Specification en entrée:

```
p: tableau de n entiers, n > 0
```

Spécification en sortie:

```
q: tableau de n entiers, t.q. q[0] ≤ q[1] ≤ … ≤ q[n-1]
                                ∀ i, p[i] inclus dans q
```

➢ Les tests supposent que la spécification est correcte

=> Toujours **valider les spécifications** avant de tester

Méthodes de vérification

Méthodes formelles

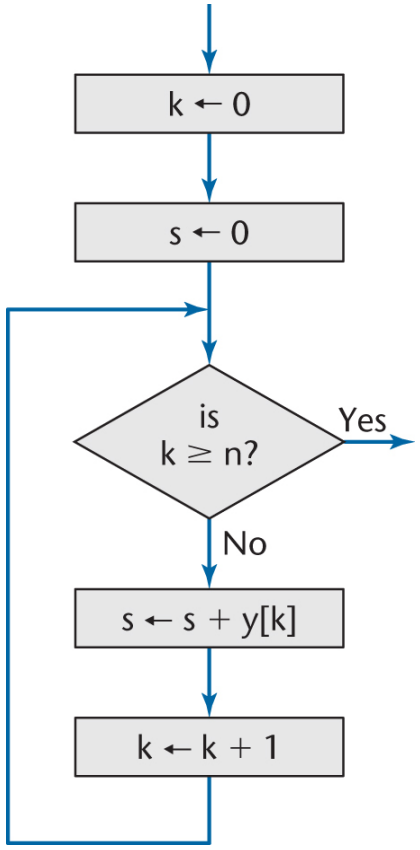
Tests (fonctionnel, structurel)

Méthodes de vérification formelles

- Que calcule le code suivant ?
- Comment **prouver** qu'il est **correct** ?

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

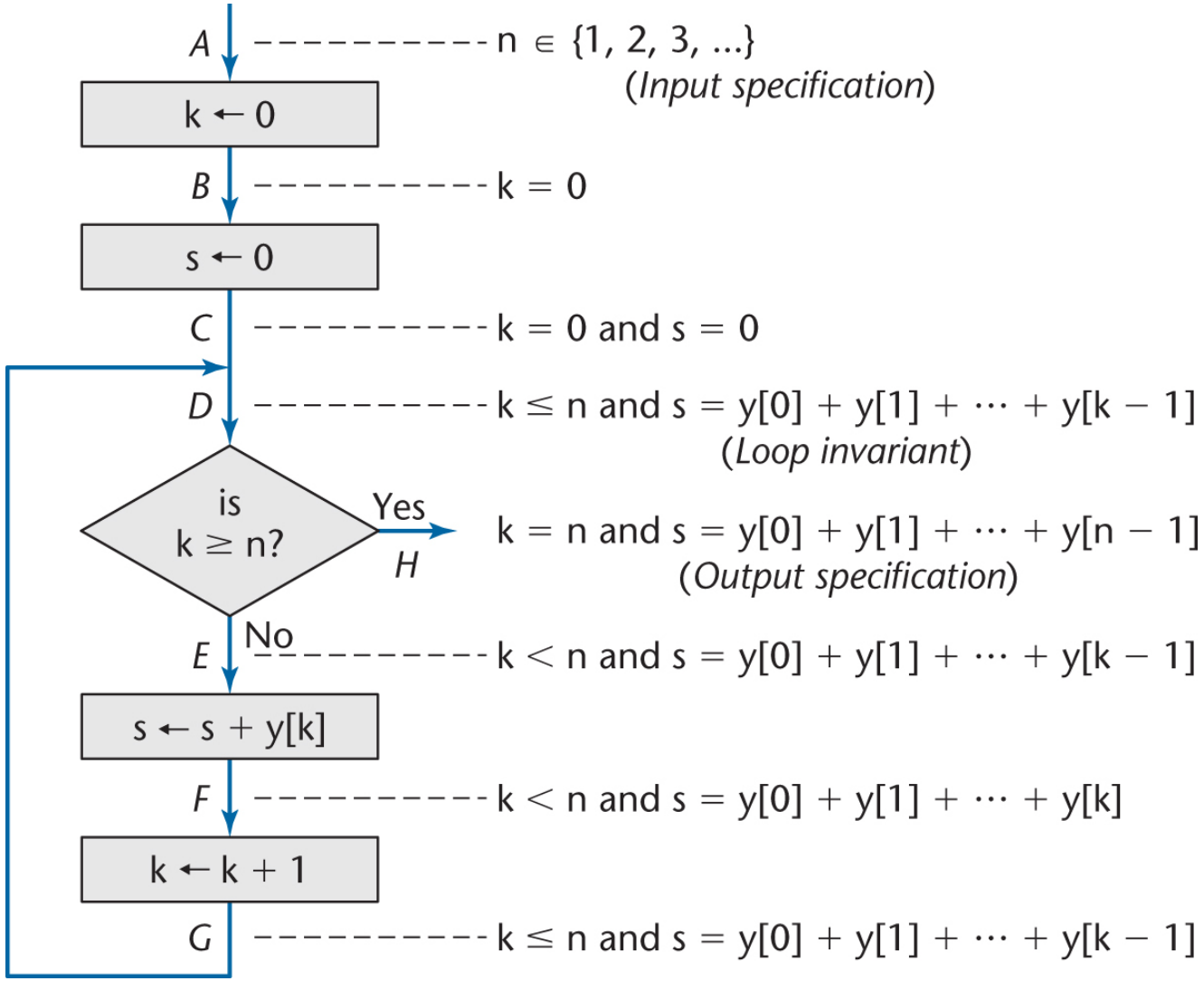
Graphe du flux de contrôle




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

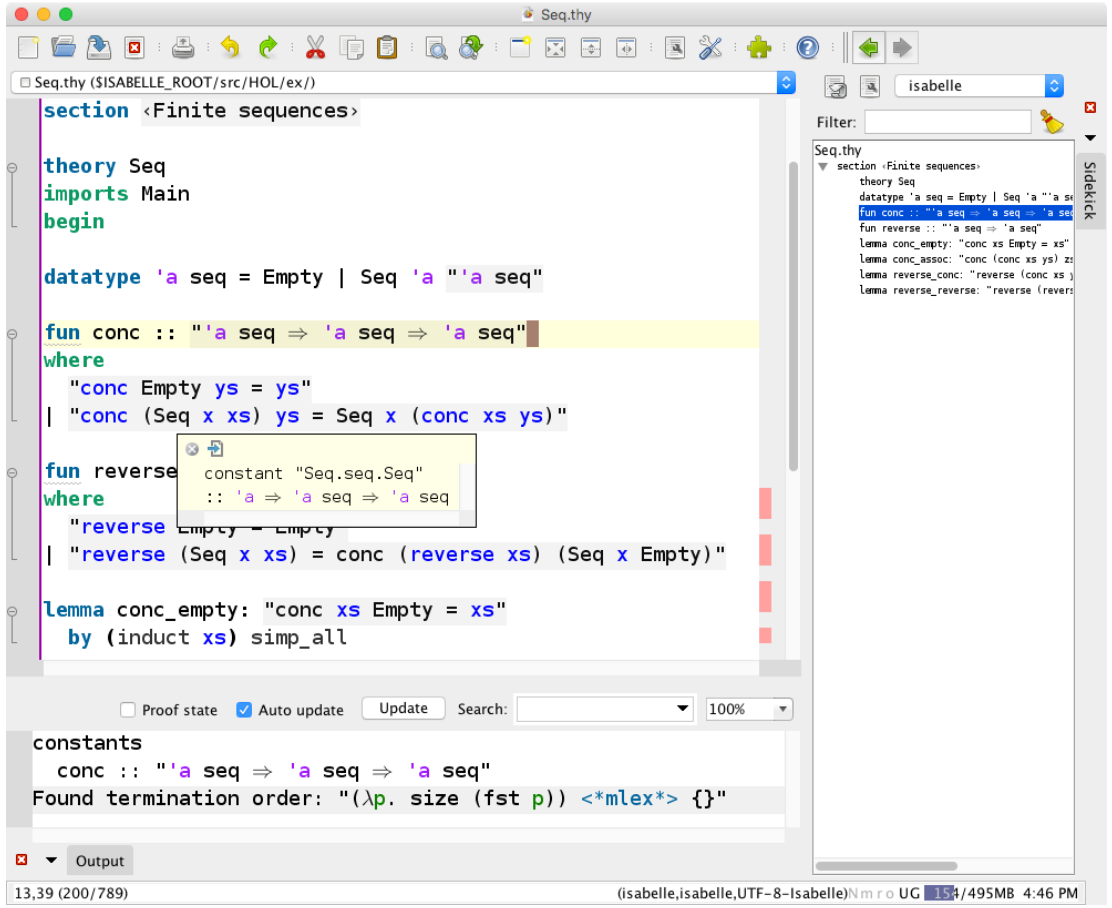
Exécution symbolique du programme

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

Test

On essaie de faire échouer. Si on n’y arrive pas, on passe.

Qu’est-ce qui guide nos tests?

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

Code

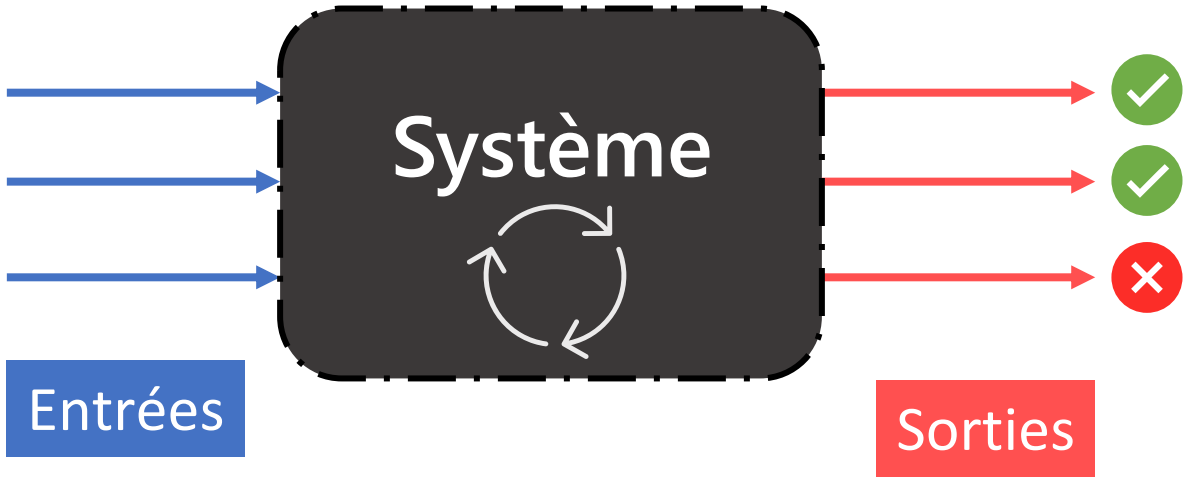
- Ignore les spécifications: utilise le code pour choisir les cas de test
- Test à la **boîte blanche**, dirigé par la logique, par la structure du code

Boite noire et boite blanche

Boite noire

Test fonctionnel

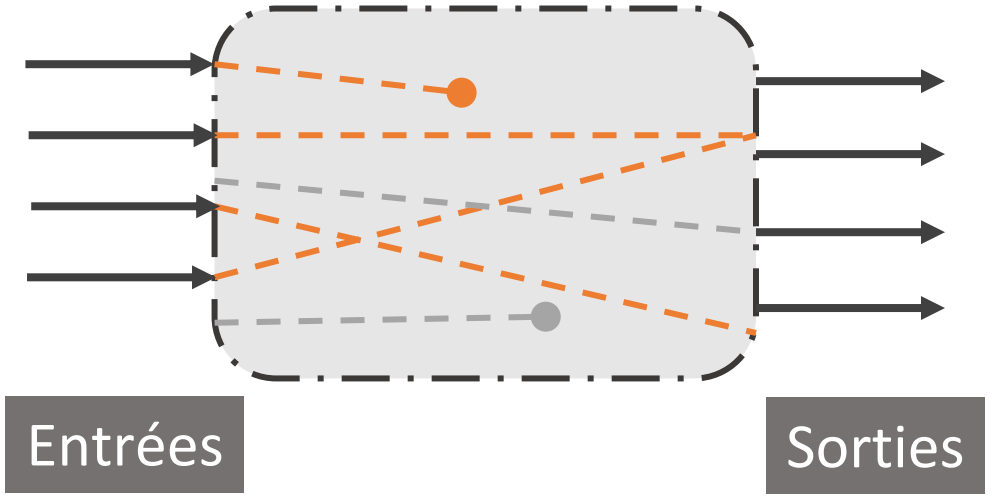
- Vérifie si le **comportement externe** du logiciel est **conforme aux exigences**



Boite blanche

Test structurel

- Vérifie si l'**implémentation** du logiciel est **correcte**



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

Analyse

- **Comparer** les résultats aux sorties attendues
- S’assurer que **l’oracle** (mécanisme permettant de décider si un test a échoué) 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

Exemple

```
public class UniversitiesControllerTests
{
  [Fact]
  public async Task GetUniversity_ReturnsHttpNotFound()
  {
    // ARRANGE
    int universityId = 12;
    // Mock le repository servant à récupérer une université dans la base de données
    var m_universityRepo = new Mock<IUniversityRepository>();
    m_universityRepo.Setup(repo => repo.GetByIdAsync(universityId))
                    .Returns(Task.FromResult((UniversityDTO)null));

    var controller = new UniversityController(m_universityRepo.Object);

    // ACT
    var result = await controller.GetUniversity(universityId);

    // ARRANGE
    var notFoundResult = Assert.IsType<NotFoundObjectResult>(result);
    var returnValue = Assert.IsType<ErrorResultDTO>(notFoundResult.Value);
    Assert.Equal("University not found", returnValue.Summary);
  }

  // autres méthodes et reste de la classe
}
```

Boite blanche

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

Exemple

```java
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;
}
```

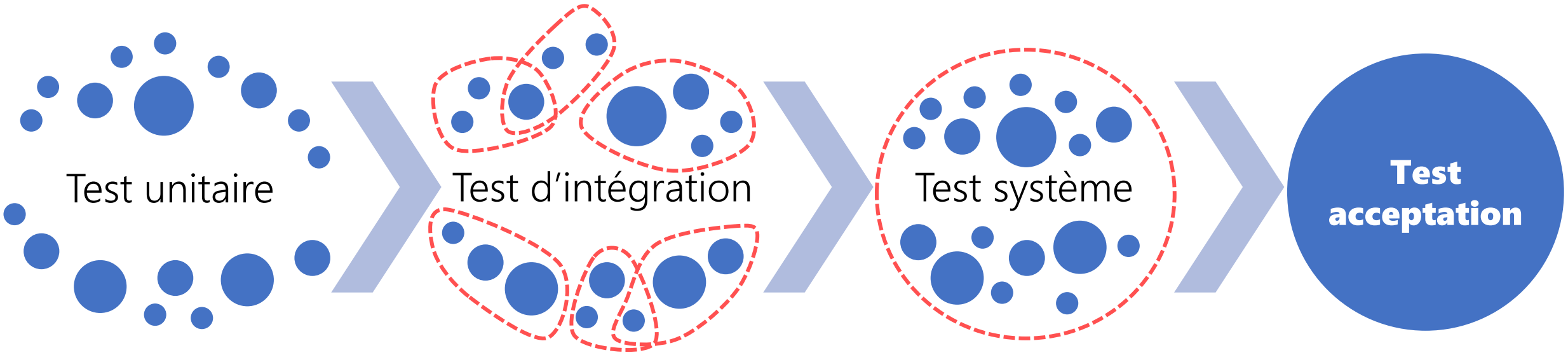
```java
private static double[] computeRoots2(double a, double b, double c) {
    List<Double> possibleRoots = new ArrayList<Double>();
    double tolerance = Double.MIN_NORMAL;
    for (double x = -3; x < Double.MAX_VALUE; x += tolerance)
    {
        double sol = a * Math.pow(x, 2) + (b * x) + c;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

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**
  - 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.
  - faux module peut retourner des **valeurs prédéterminées** lorsqu'invoqué

Types de test



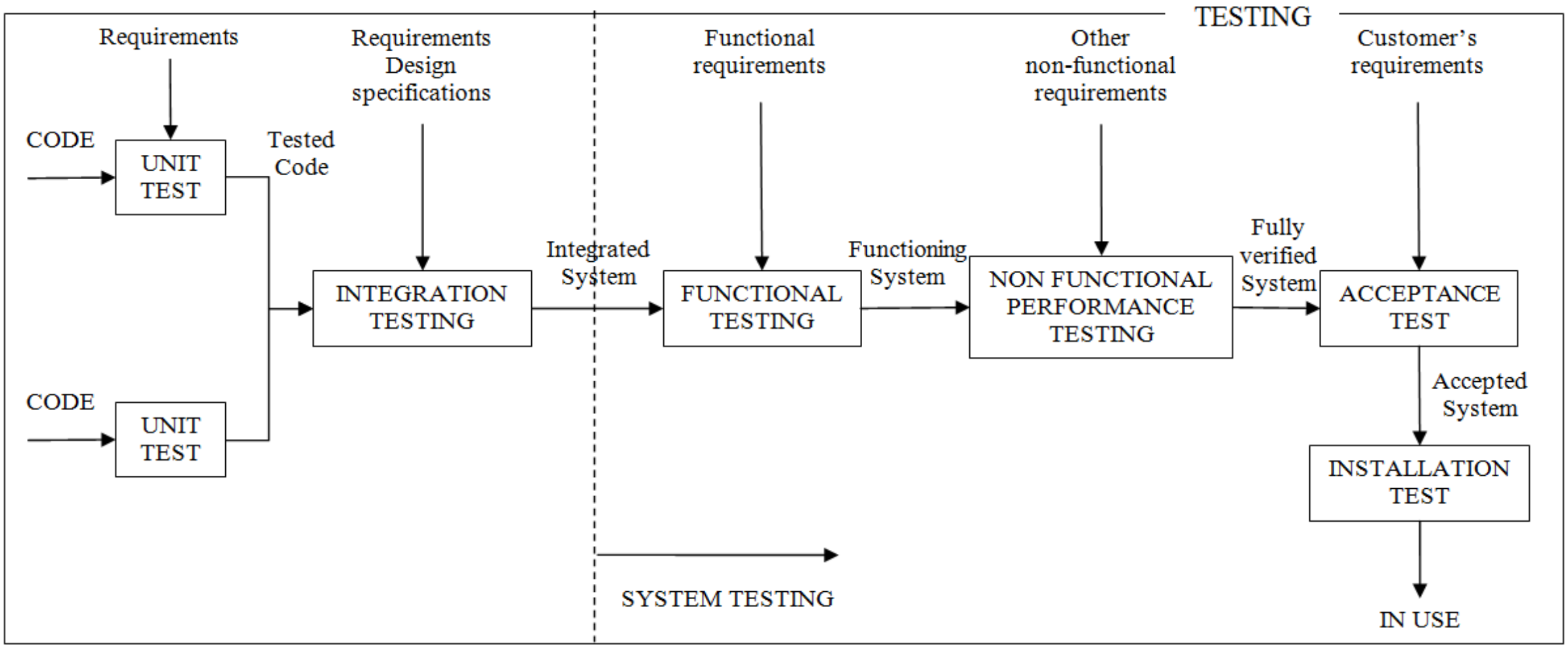
Chaque module individuellement

Groupe de modules interdépendants

Système complet

Système final (par client)

Fil conducteur

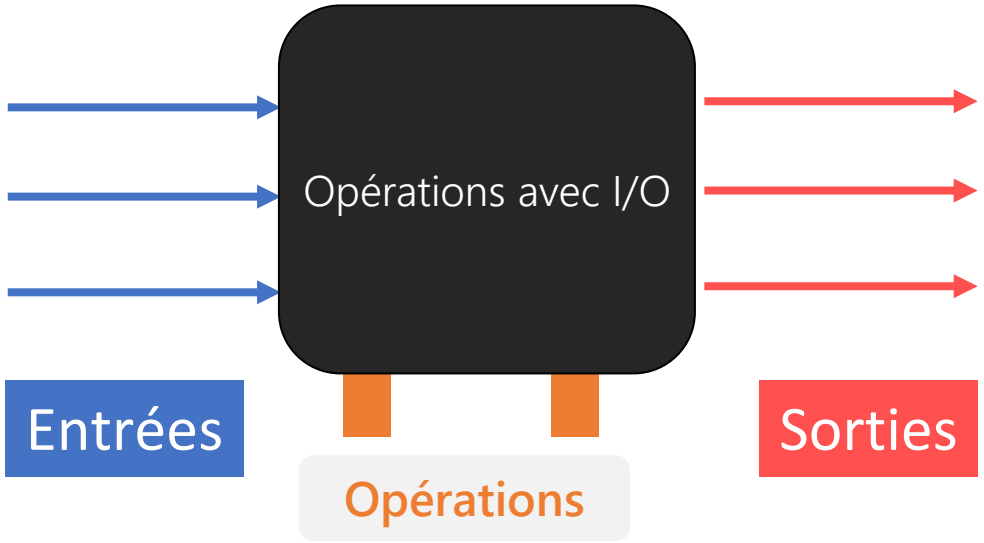


Test unitaire

Vérifie chaque module **individuellement**

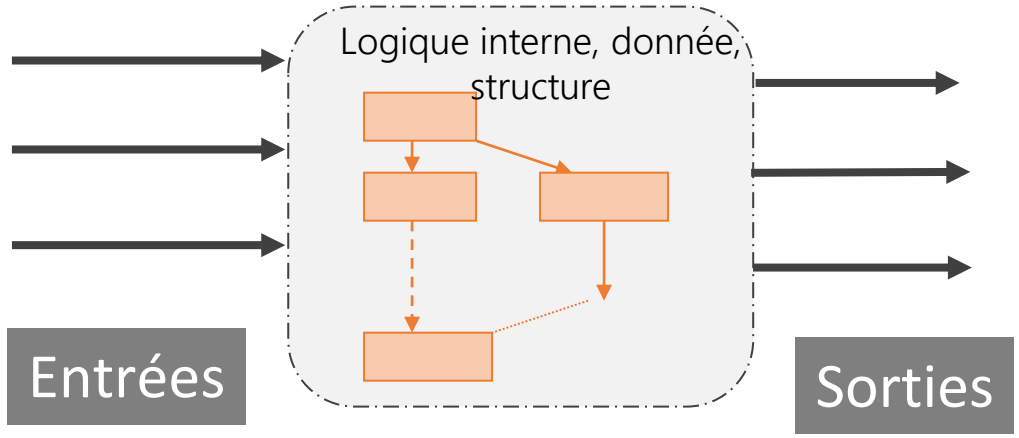
Boite noire

- ❑ Test fonctionnel
- ❑ Dépend des entrées et sorties



Boite blanche

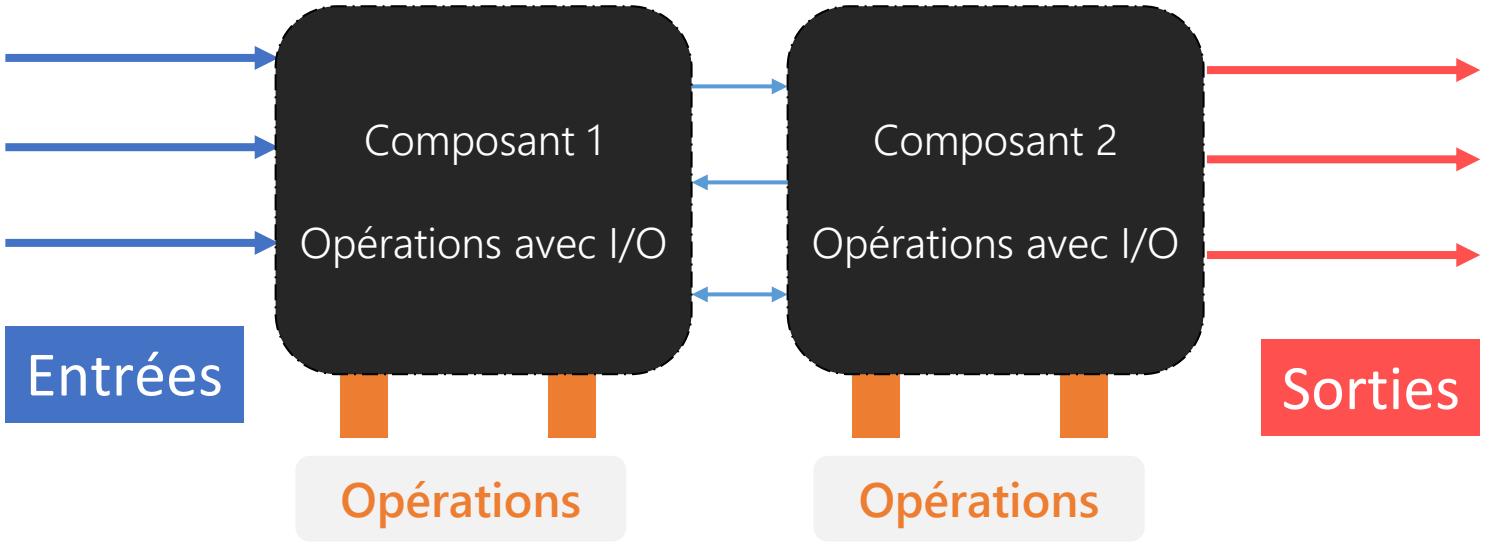
- ❑ Test structurel
- ❑ Dépend du code



Test 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 (ex: cliques, touches, mouvements)



Test fonctionnels

- L’équipe AQ doit faire une approximation des tests d’acceptation
- Assure que le code est **conforme aux exigences**
- 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 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

- 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**

Test alpha & beta

Alpha (sur application)

- Effectués par des développeurs
- Environnement contrôlé
- Test à la boite noire et boite blanche
- Quand la première ronde de correction des bogues est complétée, le produit va en test Beta

Beta (sur produit)

- Effectués par des utilisateurs
- Conditions normales d'utilisation
- Test à la boite blanche

Bonne pratique

Qui devrait tester?

- Le développeur effectue des tests **informels** sur son code.
- L'équipe AQ effectue des tests systématiques
- Le développeur 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 (documentation)

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…

Quand s'arrêter de tester?

× 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