

Le test logiciel



Réussite et échec des projets logiciels

- L'objectif du développement logiciel est de créer ou faire évoluer un logiciel le plus rapidement possible et à un coût réduit, sans compromettre sa qualité.
- L'échec d'un projet se mesure traditionnellement selon 3 principaux critères :
 - Dépassement du budget
 - Dépassement du temps
 - Non atteinte des objectifs (en termes de caractéristiques et de fonctionnalités)
- Des statistiques sur le taux d'échec ou de réussite des projets informatiques sont régulièrement publiées par des organismes de recherche et des entreprises de conseil : Standish Group, Project Management Institute, Forrester, Gartner, Harvard Business Review, etc.

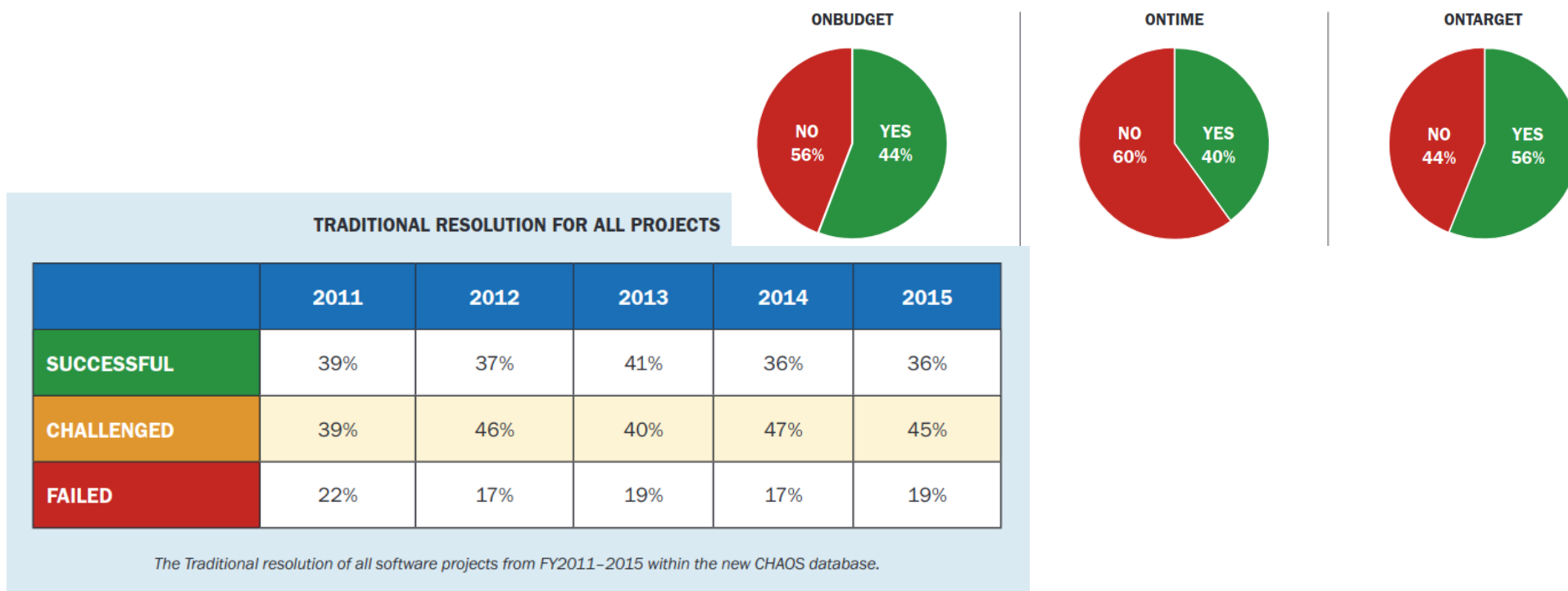
Statistiquement, le taux d'échec des projets logiciels reste élevé.



Réussite et échec des projets logiciels

- Le rapport CHAOS est l'une des sources les plus célèbres pour les statistiques sur les projets de mise en œuvre de systèmes d'information.

Ce rapport est publié annuellement par le [Standish Group](https://www.standishgroup.com/sample_research_files/CHAOSReport2015-Final.pdf), qui est une entreprise de recherche et de conseil basée aux États-Unis et spécialisée dans l'étude des projets informatiques et de leur gestion.



extrait du rapport Chaos 2015 (accessible gratuitement) du Standish Group
https://www.standishgroup.com/sample_research_files/CHAOSReport2015-Final.pdf

Réussite et échec des projets logiciels

- Le rapport CHAOS confirme que plus la complexité, la taille et la durée d'un projet sont importantes, plus le risque d'échec est élevé.

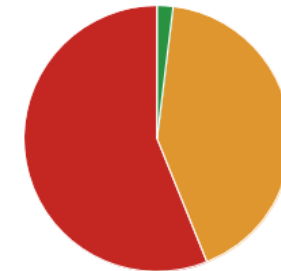
CHAOS RESOLUTION BY PROJECT SIZE

	SUCCESSFUL	CHALLENGED	FAILED
Grand	2%	7%	17%
Large	6%	17%	24%
Medium	9%	26%	31%
Moderate	21%	32%	17%
Small	62%	16%	11%
TOTAL	100%	100%	100%

The resolution of all software projects by size from FY2011–2015 within the new CHAOS database.

LARGE, COMPLEX PROJECTS

Successful 2%
 Challenged 42%
 Failed 56%



The resolution of large and complex software projects from FY2011–2015 within the new CHAOS database.

CHAOS RESOLUTION BY COMPLEXITY

	SUCCESSFUL	CHALLENGED	FAILED
Very Complex	15%	57%	28%
Complex	18%	56%	26%
Average	28%	54%	18%
Easy	35%	49%	16%
Very Easy	38%	47%	15%

The resolution of all software projects by complexity from FY2011–2015 within the new CHAOS database.

extraits du rapport Chaos 2015 du Standish Group



La qualité logicielle et la démarche qualité

Deux concepts clés pour assurer la réussite d'un projet informatique

Qualité logicielle

désigne les caractéristiques du logiciel lui-même qui font que le logiciel répond aux besoins et attentes des utilisateurs, tout en étant fiable, performant, facile à maintenir, sans défaut

- validité : réponse aux exigences
- fiabilité : fonctionnement stable sans erreurs
- robustesse : résistance aux conditions imprévues ou extrêmes
- performance : rapidité et utilisation efficace des ressources
- maintenabilité : facilité des corrections et des évolutions
- sécurité : protection des données et des utilisateurs contre les vulnérabilités
- facilité d'utilisation : retour utilisateur clair, consistance, rapidité d'apprentissage, accessibilité, simplicité d'interface

Démarche Qualité

désigne les caractéristiques du processus de développement en termes d'outils, méthodes et pratiques mis en place pour garantir que la qualité logicielle est atteinte et maintenue tout au long du cycle de vie du logiciel

- normes et standards
- outils d'aide au développement
- amélioration continue
- tests et validation
- gestion des risques
- documentation
- traçabilité

Des critères de qualité (du logiciel ou de la démarche) peuvent être en conflit, par exemple : facilité d'utilisation/sécurité, évolutivité/coût de développement. Dans le développement logiciel, il est important de trouver des compromis en fonction des priorités du projet.



Le contrôle de la qualité

Contrôle

Acte technique permettant de déterminer, avec des moyens appropriés, la conformité d'un produit (y compris service, code source, document). A l'issue d'un contrôle, une décision est prise : soit le produit est déclaré conforme, soit il est déclaré non conforme.

Contrôle a posteriori

Contrôle final effectué lorsque le produit est complètement terminé

Contrôle a priori

Contrôle en cours de production afin d'éliminer les non conformités en amont (approche réactive), de détecter les dérives (approche proactive) et ainsi de participer au pilotage de la production

- La détection précoce des non-conformités (erreurs, défauts ou écarts) permet de réduire les coûts liés aux erreurs.
- Certaines caractéristiques ne sont plus accessibles lorsque le produit est fini, il est alors nécessaire de réaliser le contrôle avant que la caractéristique ne soit masquée.
- Le contrôle en production, en détectant les dérives, permet d'apporter des actions correctives ou préventives pour éviter l'apparition d'une non conformité.



Le contrôle de la qualité

Contrôles a posteriori

- Analyse des retours utilisateurs
- Tests de recette, de validation
- Tests de performance
- Tests de sécurité
- Tests de régression

Contrôles a priori

- Revue des exigences avant de commencer le développement
- Revue de conception
- Revue de code / analyse statique du code
- Prototypage, maquette
- Tests unitaires, tests d'intégration
- Suivi de projet

Dans le contexte du développement logiciel, les **tests** constituent un type particulier de contrôles visant à vérifier le comportement du logiciel.



Le test logiciel

- Les tests sont indissociables de la qualité logicielle.
- **L'objectif du test est de détecter des fautes ou des inadéquations d'un logiciel par rapport à des références établies.**
- **Les tests ne pouvant être exhaustifs, ils ne pourront jamais prouver que le programme est juste** : un test ne peut que démontrer la présence d'erreurs, pas leur absence.
- Les récentes méthodes agiles ont remis les tests automatisés unitaires au centre de l'activité de programmation. Elles préconisent même d'écrire ces tests avant l'unité à tester selon la technique de **développement dirigé par les tests** (acronyme TDD en anglais).
- On distingue deux processus complémentaires pour le test :
 - la **validation** vérifie que le logiciel fonctionne de la façon attendue par les utilisateurs finaux, elle est réalisée à la fin du développement : contrôle a posteriori
 - la **vérification** teste si le logiciel est bien construit, elle se fait tout au long du cycle de développement pour garantir la qualité technique : contrôle a priori



Les types de test logiciel

Critère	Tests
périmètre couvert	unitaire / intégration
objectif	structurel / fonctionnel
techniques	manuel / automatisé + statique / dynamique

- **test unitaire** (contrôle d'une unité de programme telle que méthode, classe, composant, module) ou **test d'intégration** (contrôle du fonctionnement de plusieurs unités de programme ensemble)
- **test structurel** (dirigé par le code de l'unité du programme, obligatoirement conçu après le codage) ou **test fonctionnel** (dirigé par les spécifications de l'unité de programme, possiblement conçu avant le codage)
- **test manuel** (testeur humain : il faut une intervention humaine pour valider le test) ou **automatisé** (vérification sans intervention humaine)
- **test statique** (sans exécution du programme) ou **dynamique** (avec exécution du programme)



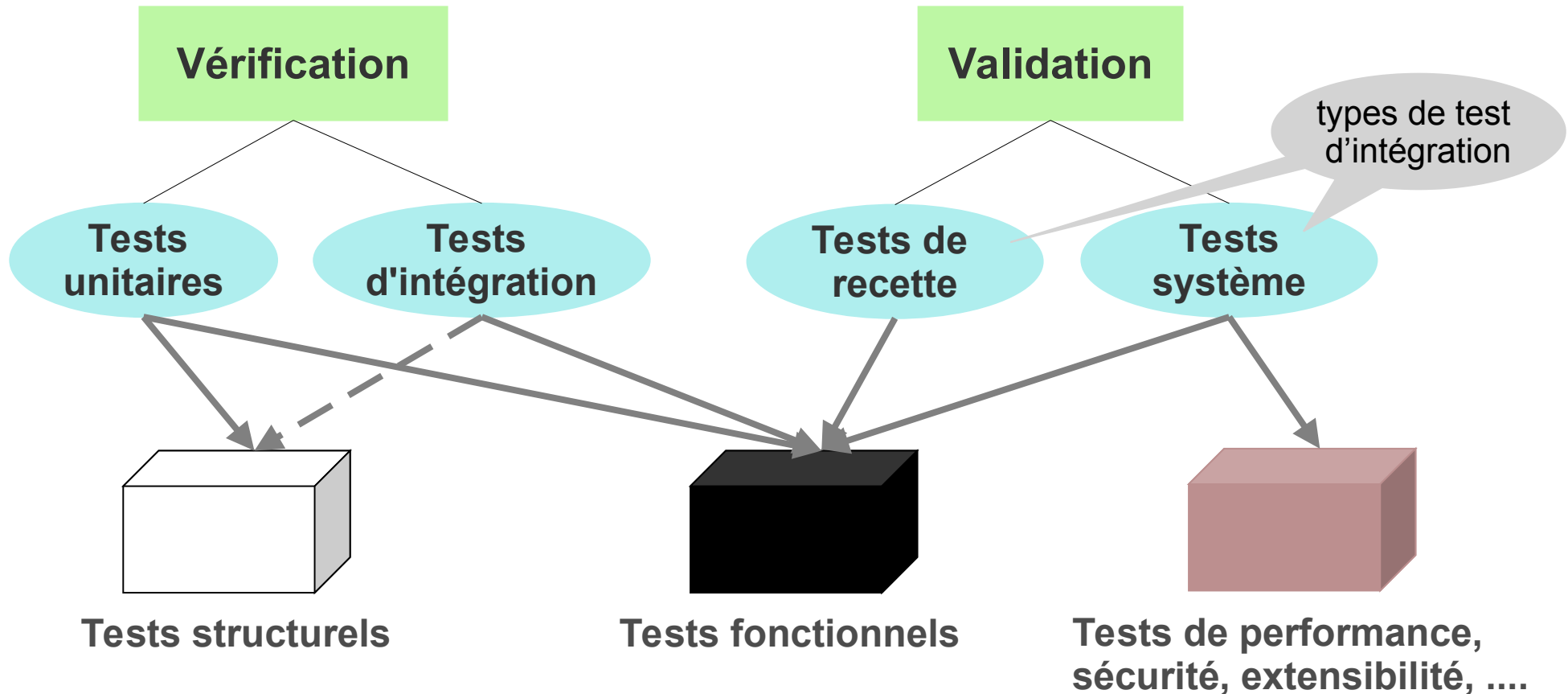
Le test statique du logiciel

- Analyse (de tout ou partie) du texte du logiciel (spécifications, code source, bytecode) sans exécution du logiciel sur des données réelles
 - manuelle : lectures croisées auteurs-lecteurs ou revue en groupe
 - assistée par des outils comme **javadoc**, **CheckStyle**, **PMD**, **SpotBugs**, JavaNCSS, JDepend, Metrics, JarDiff
- Objectifs
 - Respect des normes et des bonnes pratiques de codage
 - Détection précoce d'erreurs : code mort, mauvaises pratiques, code inutilisé
 - Calcul de métriques : complexité cyclomatique, dépendances, cohésion, nombre d'éléments (classes, champs, interfaces, méthodes, commentaires), profondeur d'héritage
- Inconvénients
 - Faux positifs : les outils détectent parfois des problèmes qui n'en sont pas réellement (bugs potentiels)



Le test dynamique du code

Test basé sur l'**exécution effective** (d'une unité) du logiciel à partir d'un sous-ensemble représentatif de ses entrées possibles, appelé **jeu de test**.



Les tests système et de recette contrôlent l'adéquation fonctionnelle du système entier (toutes les unités de programme ensemble) aux spécifications (validation). Les tests système incluent des tests non fonctionnels : performance, charge, etc.



Le test dynamique manuel /automatisé

```
public class MainTest1 {
// doit afficher 5 4 -3 -2 0 2 0 sans Exception
public static void main(String[] args) {
    System.out.println( Calcul.divide(15,3));
    System.out.println( Calcul.divide(-8,2));
    System.out.println( Calcul.divide(-6,-2));
    System.out.println( Calcul.divide(12,-6));
    System.out.println( Calcul.divide(0,4));
    System.out.println( Calcul.divide(14,6));
    System.out.println( Calcul.divide(6,14));
    try { Calcul.divide(6, 0); }
    catch ( IllegalArgumentException e ) {}
}
```

test
manuel

```
public class MainTest2 { // doit s'excuter sans Error
public static void main(String[] args) {
    if (Calcul.divide(15,3) != 5
        || Calcul.divide(-8,2) != -4
        || Calcul.divide(-6,-2) != 3
        || Calcul.divide(12,-6) != -2
        || Calcul.divide(0,4) != 0
        || Calcul.divide(14,6) != 2
        || Calcul.divide(6,14) != 0)
        throw new Error(" BUG divise de Calcul");
    try {
        Calcul.divide(6, 0);
        throw new Error(" BUG divise de Calcul");
    } catch ( IllegalArgumentException e ) {}
}
```

```
class CalculTest {
// doit être exécuté par JUnit sans Error ni Failure
@Test
void testDivise() {
    assertEquals(5, Calcul.divide(15,3));
    assertEquals(-4, Calcul.divide(-8,2));
    assertEquals(3, Calcul.divide(-6,-2));
    assertEquals(-2, Calcul.divide(12,-6));
    assertEquals(0, Calcul.divide(0,4));
    assertEquals(2, Calcul.divide(14,6));
    assertEquals(0, Calcul.divide(6,14));
    assertThrows( IllegalArgumentException.class,
        () -> Calcul.divide(6, 0));
}
```

tests automatisé
avec JUnit

test automatisé
sans JUnit



Le test automatisé

- Le test automatisé est exécuté sans intervention humaine, de manière répétable, pour s'assurer du bon fonctionnement du logiciel pendant le développement ou après des modifications (tests de régression).
- La conception des tests automatisés est coûteuse et peut nécessiter la création de modules fictifs pour activer des composants réalisés ou simuler des composants manquants. Elle peut ainsi devenir un processus de développement à part entière.
- Les environnements de test offrent un mécanisme pour automatiser l'exécution des tests et un cadre qui facilite l'écriture des tests plutôt que de coder directement dans une méthode main. Des outils comme **JUnit** et TestNG sont utilisés pour créer et exécuter des tests unitaires en Java, tandis que **Mockito** sert à créer des objets factices (mocks) afin de tester isolément les unités de code. Selenium et Cypress automatisent les tests d'interfaces utilisateur sur des applications web. Les outils de couverture de code (comme **EclEmma**) permettent de s'assurer que la majorité du code est bien testé.
- En complément des tests, l'instrumentation du code avec des assertions (instructions **assert**) ou des spécifications JML joue un rôle crucial pour le débogage.



Le jeu de test

- **La qualité d'un test repose sur la pertinence et la couverture du jeu de test**
- Un jeu de test est un ensemble fini de cas de test
- La sélection du jeu de test dépend de l'objectif du test
 - **Test structurel** : Le jeu de test vise à exécuter l'ensemble du code. La couverture complète étant difficile à atteindre, on se limite à la couverture des instructions, des conditions ou des chemins d'exécution.
 - **Test fonctionnel** : Le jeu de test doit couvrir les fonctionnalités attendues en représentant tous les domaines de valeurs des entrées, importantes pour le comportement de la fonction.
- Les jeux de tests peuvent être conçus avec :
 - des **données spécifiques**, utiles pour tester avec des scénarios prévus
 - des **données aléatoires**, utiles pour tester avec des scénarios imprévus ou pour réaliser des tests de charge, de stress, de performance. Ces données sont obtenues aléatoirement avec :
 - une distribution uniforme sur le domaine d'entrée, complétée par des données limites ou exceptionnelles,
 - ou bien une distribution proche de celle attendue en conditions réelles d'exploitation



Jeu de test structurel / fonctionnel

Calcul.java ×

```

2
3 public class Calcul {
4     public static int divide(int a, int b) {
5         if (b == 0) {
6             throw new IllegalArgumentException("Cannot divide by zero");
7         }
8         return a / b;
9     }
10 }
11

```

CalculTest.java ×

```

7
8 class CalculTest {
9     @Test
10    void testStructurelMinimalDivise() {
11        assertEquals(5, Calcul.divide(15,3));
12        assertThrows( IllegalArgumentException.class,
13            () -> Calcul.divide(6, 0));
14    }
15    @Test
16    void testFonctionnelComplementaireDivise() {
17        // avec 1 ou 2 operandes negatifs
18        assertEquals(-4, Calcul.divide(-8,2));
19        assertEquals(3, Calcul.divide(-6,-2));
20        assertEquals(-2, Calcul.divide(12,-6));
21        // avec dividende nul
22        assertEquals(0, Calcul.divide(0,4));
23        // avec une partie decimale (en division reelle)
24        assertEquals(2, Calcul.divide(14,6));
25        assertEquals(0, Calcul.divide(6,14));
26    }
27 }

```

Coverage ×

CalculTest (31 janv. 2025 17:49:16)

Element	Coverage	Covered Instruct	Missed Instruc	Total Instructions
Calcul.java	78,6 %	11	3	14
Calcul	78,6 %	11	3	14
divide(int, int)	100,0 %	11	0	11

COUVERTURE
de divide
=100%

jeu de test
constitué de
8 cas de test

CAS FONCTIONNELS

- division de 2 nbs positifs
- division de 2 nbs négatifs
- division avec 1 nb positif et 1 nb négatif
- dividende nul
- diviseur nul
- division avec reste nul
- division avec reste non nul
- quotient nul
- quotient non nul



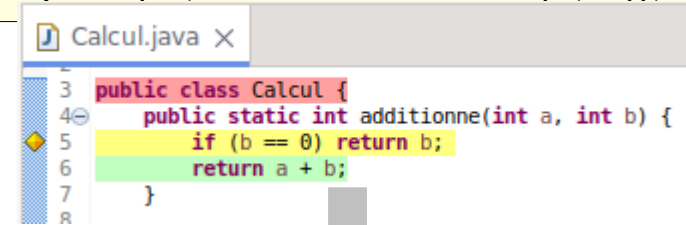
Complémentarité test structurel / fonctionnel

```
public class Calcul {
    public static int additionne(int a, int b) {
        if (b == 0) return b; // bug
        return a + b;
    }
}
```

- La méthode *additionne* est censée calculer la somme de deux entiers. Il y a une erreur dans le code pour $b = 0$.
- Les tests fonctionnels ne peuvent pas détecter une erreur présente dans une instruction qui n'est jamais exécutée. **Les tests fonctionnels doivent donc être complétés par des tests structurels.** L'approche structurelle produira forcément la donnée de test $b = 0$ (à quelconque) et pourra détecter l'erreur commise dans le code.
- Le test fonctionnel exhaustif est généralement impossible à réaliser car l'ensemble des données d'entrée est infini ou de très grande taille. Ici chaque int a 2^{32} valeurs possibles, ce qui fait un nombre total de combinaisons possibles pour les 2 entiers de 2^{64} .

Des tests fonctionnels

```
assertEquals(3, Calcul.additionne(-1,4));
assertEquals(-6, Calcul.additionne(-4,-2));
assertEquals(11, Calcul.additionne(1,10));
assertEquals(-3, Calcul.additionne(0,-3));
```



Test structurel complémentaire
`assertEquals(8, Calcul.additionne(8,0));`



Complémentarité test structurel / fonctionnel

```
public class Calcul {

    public static int divise(int a, int b) {
        return a / b; // bug quand b ==0
    }
}
```

- D'après les spécifications, la méthode *divise* devrait lancer une exception appropriée en cas de division par zéro. Elle est ici incomplète du point de vue des spécifications.
- Les tests structurels ne peuvent pas détecter les omissions ou erreurs de spécification. **Les tests structurels doivent donc être complétés par des tests fonctionnels.** L'approche fonctionnelle prévoiera le test avec la donnée $b = 0$ (cas de la division par zéro) et détectera le défaut.
- Le test structurel exhaustif est lui aussi généralement impossible à réaliser car le parcours du graphe de flot de contrôle conduit à une forte explosion combinatoire.

Tests structurels qui couvrent divise à 100 %
`assertEquals(2, Calcul.divise(8,4));`



```
Calcul.java x
3 public class Calcul {
4     public static int additionne(int a, int b) {
5         return a + b;
6     }
7 }
```

Tests fonctionnels complémentaires
`assertThrows(IllegalArgumentException.class,`
 `() → Calcul.divise(8,0));`
 ...

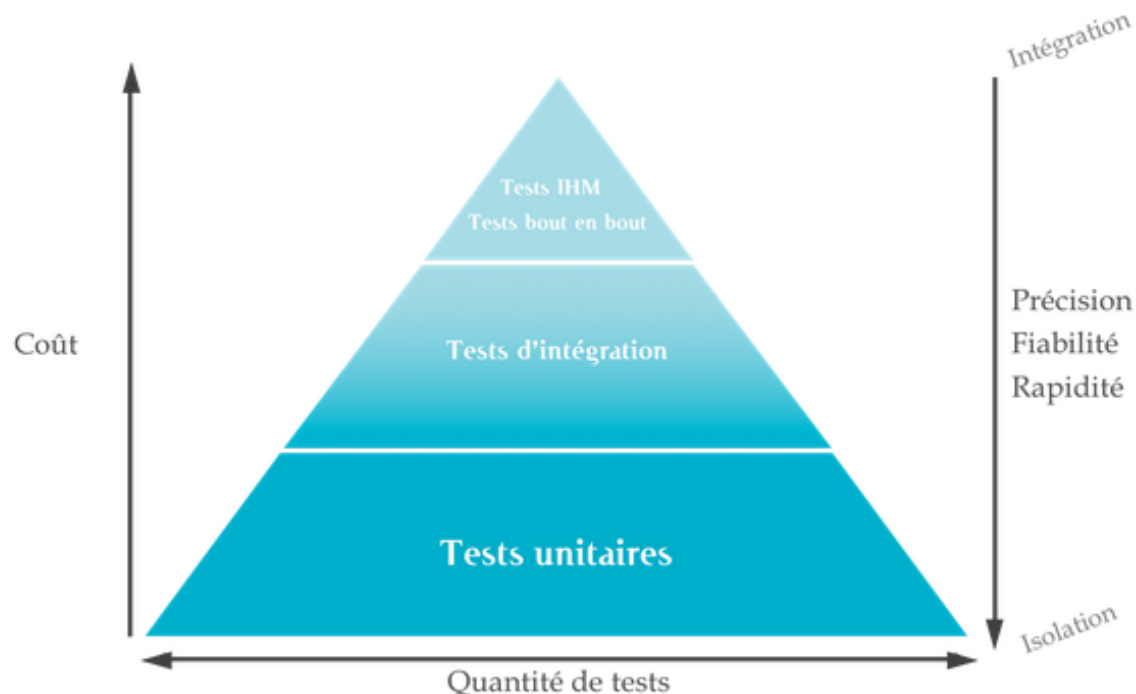


Le développement piloté par les tests

- Principe : On écrit les **tests fonctionnels automatisés** avant d'écrire le code.
- Avantages : amélioration de la qualité et de la pérennité du code
 - Réduction des bugs : impossible de livrer du code non testé, détection des bugs plus tôt dans le processus de développement, gain de temps pendant le débogage, tests de régression facilités
 - Réflexion sur ce que fait le code avant de coder : éviter d'écrire un test faux validant un code faux (puisque le test ne peut se baser que sur la spécification)
 - Architecture testable, meilleure conception : couplage faible entre les objets, pas de dépendances à des implémentations, conception plus facile à faire évoluer.
- Inconvénients
 - Processus long pour maintenir une grande couverture des tests
- Technique
 - Ecrire un test fonctionnel automatisé à la fois qui décrit le comportement attendu pour une nouvelle fonctionnalité. L'exécuter et constater que le test ne passe pas (puisque la fonctionnalité manque).
 - Ecrire le code suffisant pour passer le test. Lancer le test et vérifier qu'il passe. Optimiser le code.
 - Compléter le test fonctionnel par des tests structurels



La stratégie des tests automatisés



<https://blog.octo.com/la-pyramide-des-tests-par-l-a-pratique-2-5>




- **Tests unitaires**

les plus nombreux, simples, isolation, rapides et autonomes. Ils simulent des scénarios très éloignés de l'utilisation de l'application finale. Ils sont stables donc particulièrement rentables.

- **Tests d'intégration**, jusqu'aux **tests (nécessairement fonctionnels) système**

De plus en plus longs et complexes au fur et à mesure que les composants testés grossissent. Ils simulent des scénarios de plus en plus proches de l'utilisation de l'application finale et offrent une meilleure garantie concernant le bon fonctionnement de l'application finale. Comme ils couvrent un spectre plus large de code, ils ont davantage de risques d'être impactés par une modification. Plus lents et moins stables que les tests unitaires, ils sont forcément moins rentables.

La stratégie des tests automatisés

Type de test	Feedback			Coût (création + maintenance)
	Précision	Fiabilité	Rapidité	
Test Unitaire	Très fine (au niveau fonction)	Très fiable (répétable à l'infini)	Très rapide (quelques secondes)	
Test d'Intégration Test Fonctionnel	Moyenne (partie de logiciel intégré)	Fiable (les dépendances peuvent provoquer des échecs)	Assez rapide (quelques minutes)	
Test de bout en bout Test d'IHM	Faible (logiciel pleinement intégré)	Peu fiable (relativement instable)	Lent (plusieurs dizaines de minutes)	

feedback

précision = capacité à déterminer précisément la partie de code qui ne fonctionne pas

fiabilité = le test doit être répétable, ses résultats ne doivent pas varier d'une exécution à l'autre

rapidité = correspond à la rapidité d'exécution des tests

<https://blog.octo.com/la-pyramide-des-tests-par-la-pratique-1-5>

- Les tests d'intégration sont indispensables pour garantir le bon fonctionnement de l'application finale, mais leur coût de développement est important.
- Les tests unitaires garantissent que les unités de code utilisées par les tests d'intégration sont robustes. En permettant de détecter les bugs au plus tôt, les tests unitaires minimisent le coût de correction des bugs et donc l'impact en termes de coût sur le projet.
- Les bonnes pratiques de test visent à coder beaucoup de tests unitaires.
- La lecture des tests doit permettre de comprendre ce que fait le code mais aussi comment il fonctionne et à quel point il est opérationnel. Les tests sont ainsi un outil de communication efficace à destination des développeurs qui devront modifier le code de l'application.