


Génie Logiciel Avancé

Cours 1 — V&V, Testing

Mo Foughali
foughali@irif.fr

Laboratoire IRIF, Université de Paris

2021–2022

URL <https://moodle.u-paris.fr/course/view.php?id=10699>
Copyright' 2021–2022 Mo Foughali
© 2016–2019 Stefano Zacchiroli
© 2014–2015 Mihaela Sighireanu
License Creative Commons Attribution-ShareAlike 4.0 International License
<https://creativecommons.org/licenses/by-sa/4.0/>


V&V : le pourquoi

Systèmes informatiques : domaines d'application de plus en plus critiques

- Avions, automobile, systèmes autonomes, robotique spatiale

Si le système échoue à remplir sa mission, les conséquences peuvent être dramatiques : pertes économiques et/ou humaines considérables

- Vol 501 d'Ariane 5, 1996 (500 millions \$)
- "Toyota Unintended Acceleration", 2007-2013 (89 morts, plus d'un milliard \$)
- Véhicule autonome Uber, 2018 (1 mort, coût économique inconnu)

Souvent, un tel échec survient au niveau du logiciel

- Le logiciel doit être vérifié vis-à-vis de ce qu'on attend du système
 - ▶ Détection et correction de "bugs" avant de déployer le système
 - ▶ Ces bugs peuvent échapper aux tests systématiques (exemple "NASA Remote Agent Experiment")

V&V : le quoi (rappel)

Il s'agit de s'assurer que :

- La conception du logiciel est fiable aux exigences du client (Validation)
- L'implémentation est fiable à la conception (Vérification)

V&V : le quoi (rappel)

Il s'agit de s'assurer que :

- La conception du logiciel est fiable aux exigences du client (Validation)
- L'implémentation est fiable à la conception (Vérification)

On revient à une des grandes problématiques de l'informatique :

V&V : le quoi (rappel)

Il s'agit de s'assurer que :

- La conception du logiciel est fiable aux exigences du client (Validation)
- L'implémentation est fiable à la conception (Vérification)

On revient à une des grandes problématiques de l'informatique :

le logiciel fait-il ce que l'on veut ?

V&V : le quoi (rappel)

Il s'agit de s'assurer que :

- La conception du logiciel est fiable aux exigences du client (Validation)
- L'implémentation est fiable à la conception (Vérification)

On revient à une des grandes problématiques de l'informatique :

le logiciel fait-il ce que l'on veut ?

Cette question est extrêmement compliquée !

Les méthodes formelles

Explication donnée au tableau !

Les méthodes formelles

Exemples

Transition systems – Foughali et al (2020) – exemple vu en amphi

Proof systems, theorem proving – de Gouw et al. (2015) – exemple vu en amphi

V&V : le comment

Les méthodes formelles

Les +

- Méthodes rigoureuses, résultats extrêmement fiables
 - ▶ Même quand non exhaustives
- Algorithmes souvent automatiques

V&V : le comment

Les méthodes formelles

Les +

- Méthodes rigoureuses, résultats extrêmement fiables
 - ▶ Même quand non exhaustives
- Algorithmes souvent automatiques

Les -

- Scalability
- Scalability

- Scalability

V&V : le comment

Testing

Nécessaire : exécution du système réel, découverte d'erreurs à tous les niveaux (exigences, conception, implémentation)

Mais ...

Insuffisant ! (e.g. exhaustivité impossible)

Test de logiciel

Selon IEEE (Standard Glossary of Software Engineering Terminology)

« 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. »

- ▶ Validation dynamique (exécution du système)
- ▶ Comparaison entre système et spécification



Qu'est ce qu'un « bug » ?

(vocabulaire IEEE)

- ▶ **Anomalie** (fonctionnement) : différence entre comportement attendu et comportement observé
- ▶ **Défaut** (interne) : élément ou absence d'élément dans le logiciel entraînant une anomalie
- ▶ **Erreur** (programmation, conception) : comportement du programmeur ou du concepteur conduisant à un défaut

erreur → défaut → anomalie



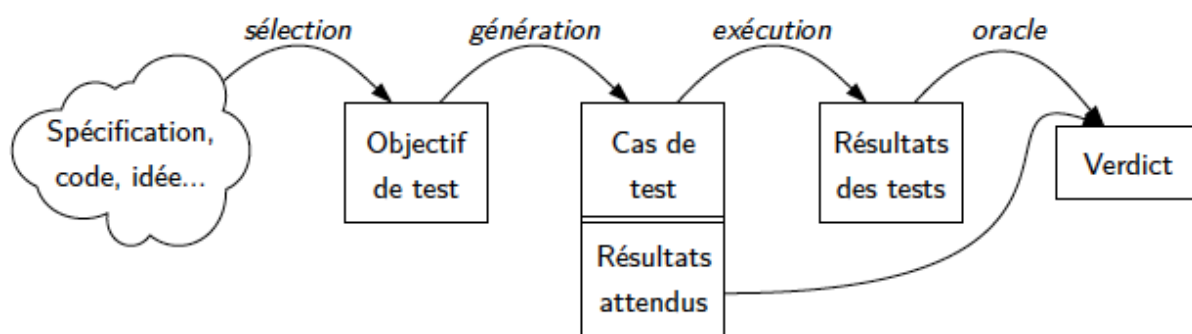
Qu'est ce qu'un test ?

(vocabulaire IEEE)

- ▶ **SUT** (System Under Test) : le système testé.
- ▶ **Objectif de test** : comportement SUT à tester
- ▶ **Données de test** : données à fournir en entrée au système de manière à déclencher un objectif de test
- ▶ **Résultats d'un test** : conséquences ou sorties de l'exécution d'un test
 - ▶ (affichage à l'écran, modification des variables, envoi de messages...)
- ▶ **Cas de test** : données d'entrée **et** résultats attendus associés à un objectif de test



Qu'est ce qu'un test ?



Exemple

- ▶ **Spécification** : Le programme prend en entrée trois entiers, interprétés comme étant les longueurs des côtés d'un triangle. Le programme retourne la propriété du triangle correspondant : scalène, isocèle ou équilatéral.
- ▶ **Exercice** : Écrire un ensemble de tests pour ce programme



Exemple

Cas valides

	Données	Résultat attendu
triangle scalène valide	(10,5,7)	scalène
triangle isocèle valide + permutations	(3,5,5)	isocèle
triangle équilatéral valide	(3,3,3)	équilatéral
triangle plat ($a+b=c$) + permutations	(2,2,4)	scalène

Cas invalides

pas un triangle ($a+b<c$) + permutations	(2,1,5)	triangle invalide
une valeur à 0	(3,0,4)	triangle invalide
toutes les valeurs à 0	(0,0,0)	triangle invalide
une valeur négative	(2,-1,6)	triangle invalide/entrée invalide
une valeur non entière	('a',4,2)	entrée invalide
mauvais nombre d'arguments	(3,5)	entrée invalide



Un autre exemple : tri d'une liste

Objectif de test	Donnée d'entrée	Résultat attendu	Résultat du test
liste vide	[]	[]	[. . .]
liste à 1 élément	[3]	[3]	[. . .]
liste ≥ 2 éléments, déjà triée	[2;6;9;13]	[2;6;9;13]	[. . .]
liste ≥ 2 éléments, non triée	[7;10;3;8;5]	[3;5;7;8;10]	[. . .]

égalité ?



Problème de l'oracle

- ▶ **Oracle** : décision de la réussite de l'exécution d'un test, comparaison entre le résultat attendu et le résultat obtenu
- ▶ **Problème** : décision pouvant être complexe
 - ▶ types de données sans prédicat d'égalité
 - ▶ système non déterminisme : sortie possible mais pas celle attendue
 - ▶ heuristique : approximation du résultat optimal attendu
 - ▶ Exemple : problème du sac à dos
- ▶ **Risques** : Échec d'un programme conforme si définition trop stricte du résultat attendu
 - ▶ => faux positifs (false fails)



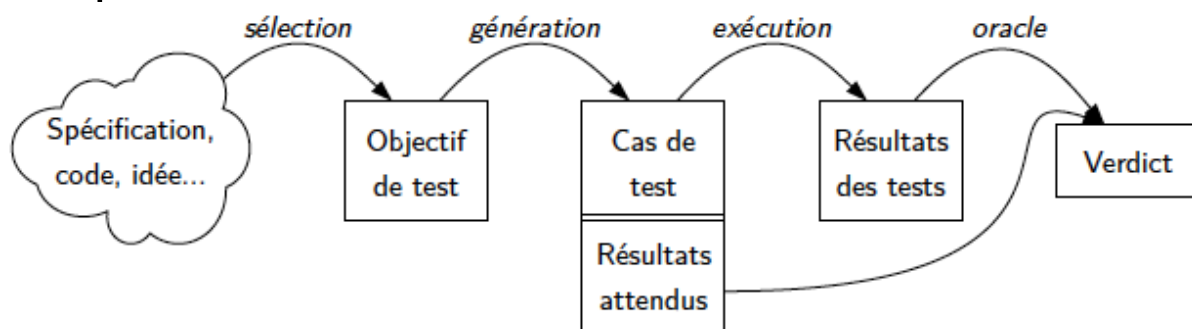
Faux positifs et faux négatifs

- ▶ **Validité des tests** : Les tests n'échouent que sur des programmes incorrects
- ▶ Faux positif (*false-fail*) : *fait échouer un programme correct*
- ▶ **Complétude des tests** : Les tests ne réussissent que sur des programmes corrects
- ▶ Faux négatif (*false-pass*) : *fait passer un programme incorrect*
- ▶ Validité indispensable, complétude impossible en pratique
 - ▶ **Toujours s'assurer que les tests sont valides**



Processus de test

1. Choisir les comportements à tester (**objectifs de test**)
2. Choisir des **données de test** permettant de déclencher ces comportements + décrire le **résultat attendu** pour ces données
3. **Exécuter** les cas de test sur le système + collecter les **résultats**
4. Comparer les résultats obtenus aux résultats attendus pour **établir un verdict**



Exécution d'un test

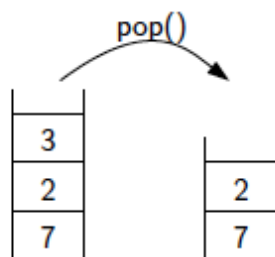
- ▶ **Scénario de test :**
 - ▶ **Préambule** : Suite d'actions amenant le programme dans l'état nécessaire pour exécuter le cas de test
 - ▶ **Corps** : Exécution des fonctions du cas de test
 - ▶ **Identification** (facultatif) : Opérations d'observation rendant l'oracle possible
 - ▶ **Postambule** : Suite d'actions permettant de revenir à un état initial



Exécution de test

Ex : Pop (supprimer le sommet d'une pile)

Cas de test :

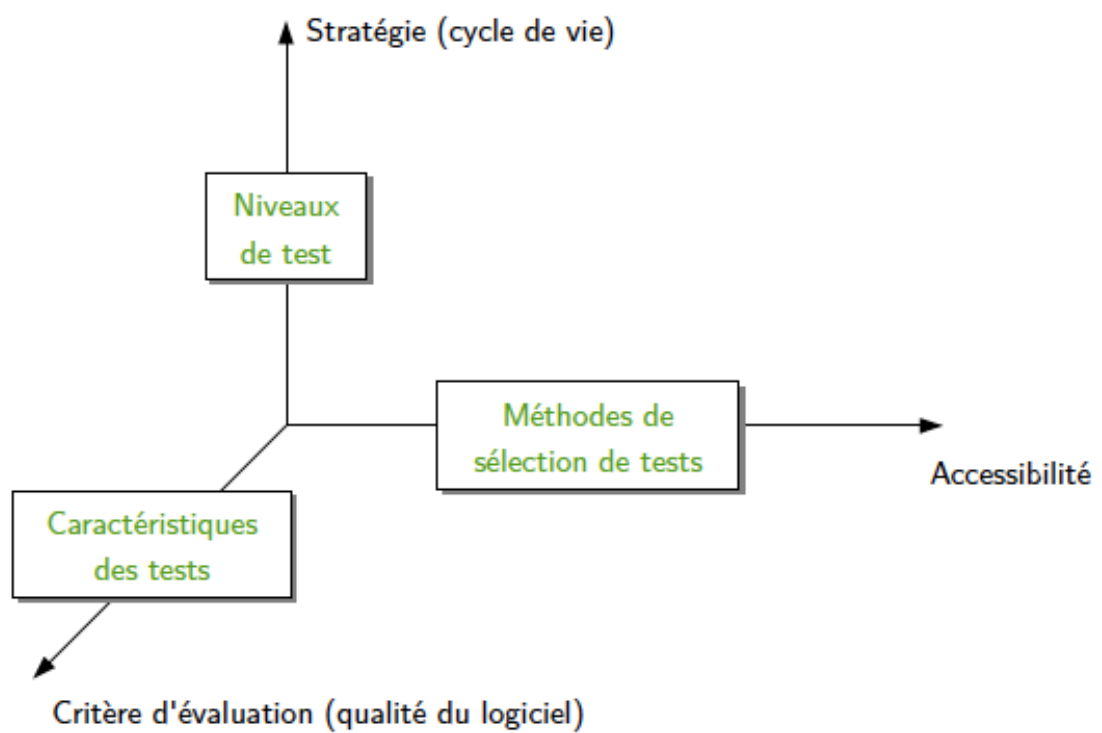


Exécution du test :

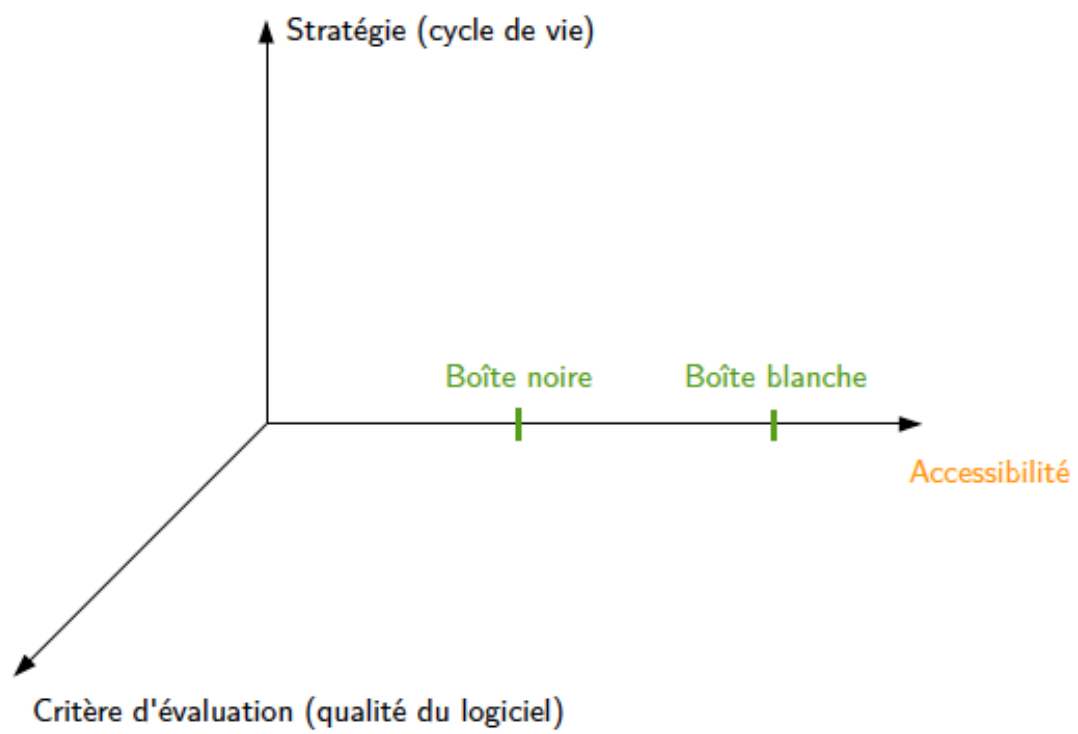
Préambule	push(7)
	push(2)
	push(3)
Corps	pop()
Identification	top() = 2
	pop()
	top() = 7
	pop()
	top() = <i>empty</i>



Types de test

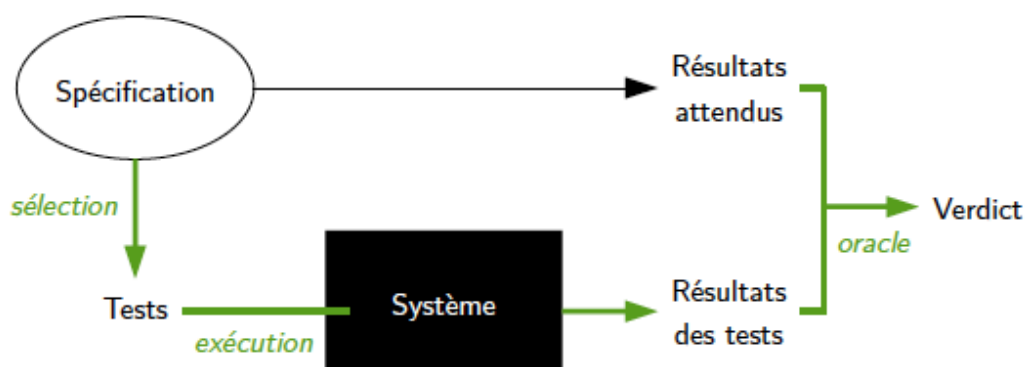


Types de test



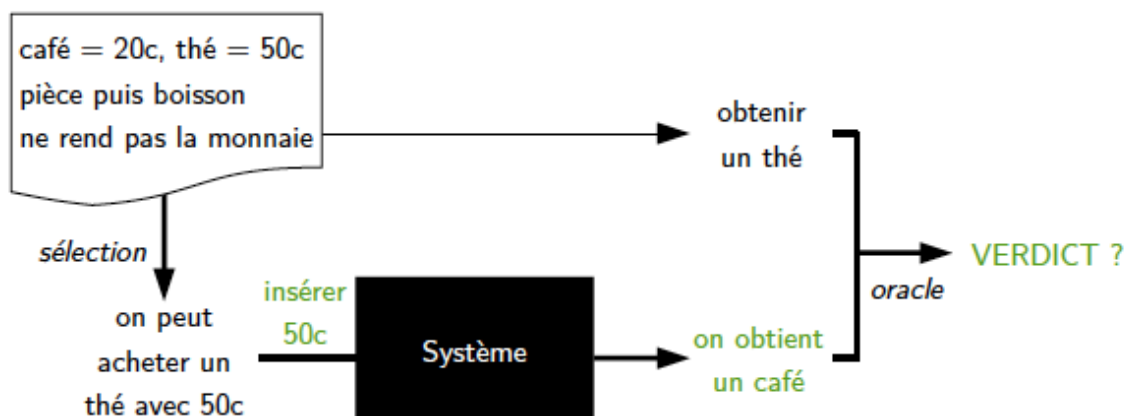
Test en boîte noire

- ▶ Sélection des tests à partir d'une spécification du système (formelle ou informelle), sans connaissance de l'implantation.
 - ▶ Test « fonctionnel »
- ▶ Possibilité de construire les tests pendant la conception, avant le codage



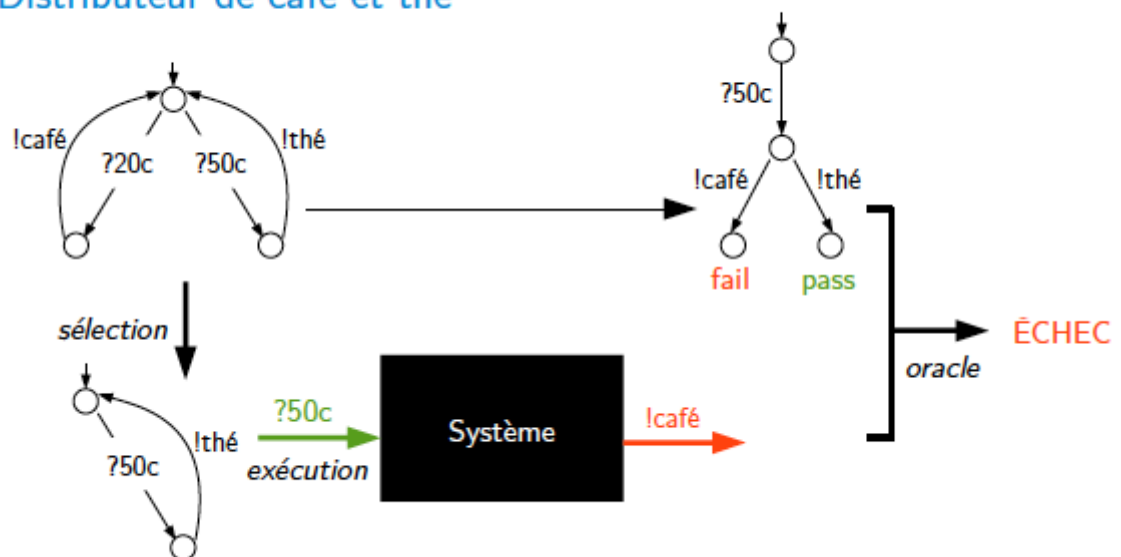
Test en boîte noire

Ex : Distributeur de café et thé



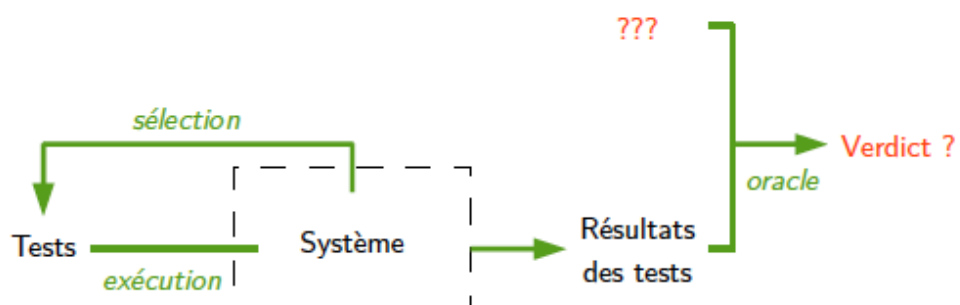
Test en boîte noire

Ex : Distributeur de café et thé



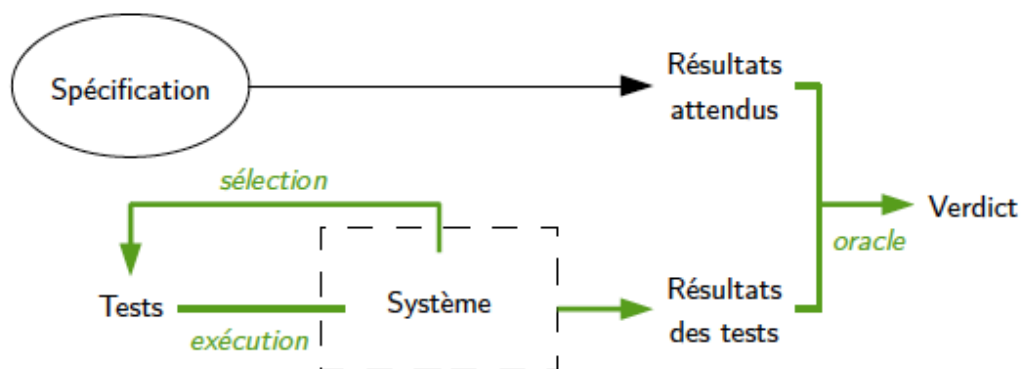
Test en boîte blanche

- ▶ Sélection des tests à partir de l'analyse du code source du système
 - ▶ Test « structurel »
- ▶ Construction des tests uniquement pour du code déjà écrit !

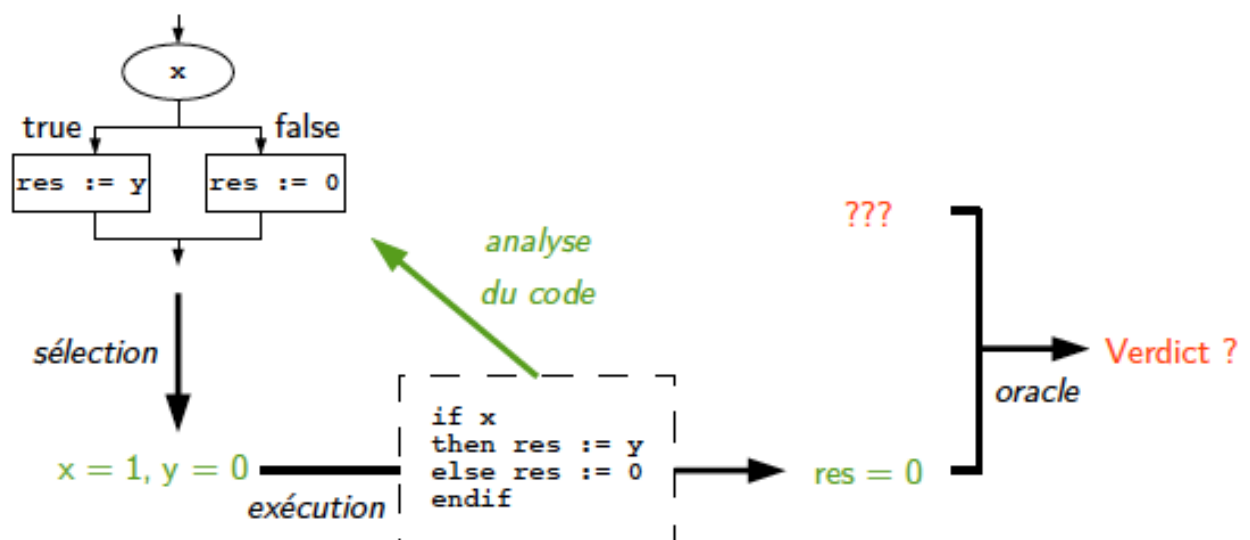


Test en boîte blanche

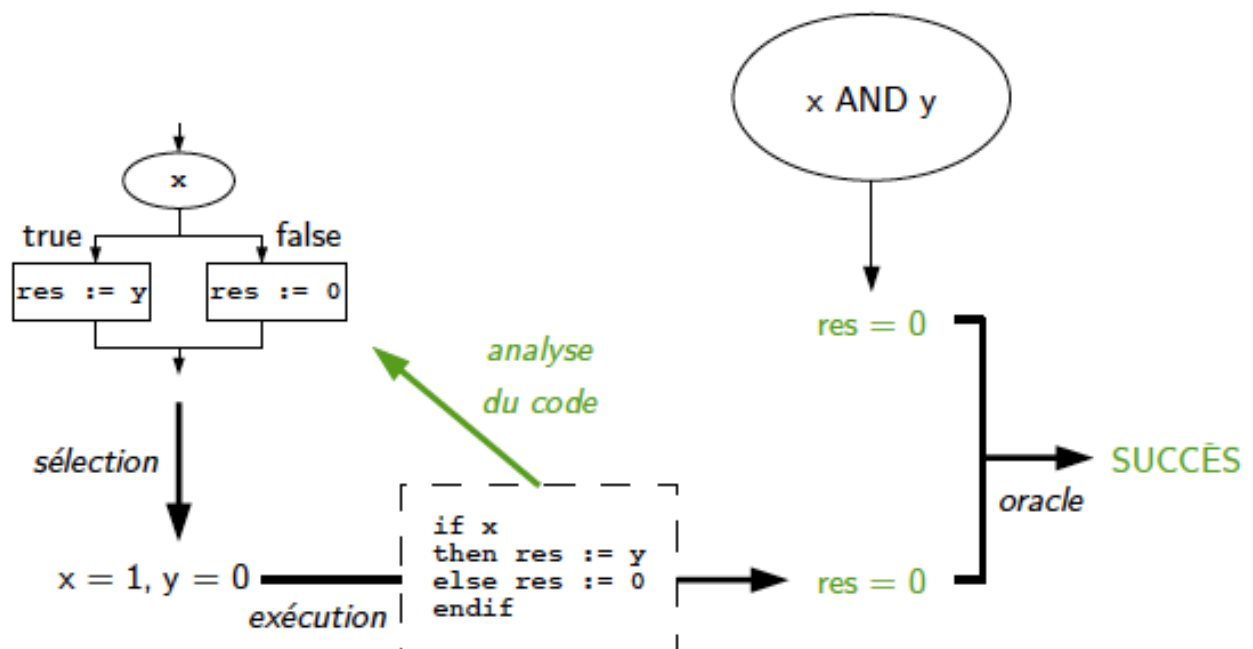
- ▶ Sélection des tests à partir de l'analyse du code source du système
 - ▶ Test « structurel »
- ▶ Construction des tests uniquement pour du code déjà écrit !



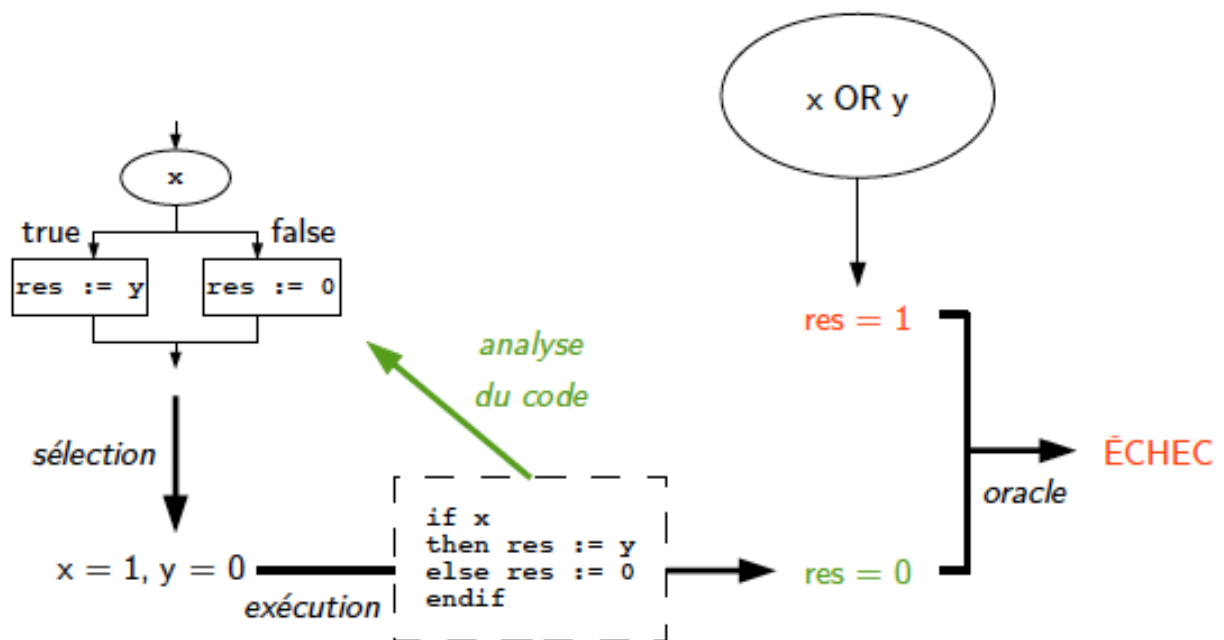
Test en boîte blanche



Test en boîte blanche



Test en boîte blanche



Test en boîte blanche

- ▶ Comment sélectionner des données de test qui détectent le plus d'erreurs ?
- ▶ Réponse : Couvrir les
 - ▶ Instructions du programme
 - ▶ Changement de contrôle du programme
 - ▶ Décisions du programme
 - ▶ Comportements du programme
- ▶ Flot de données (définition -> utilisation de variable)

simple
↓
impossible

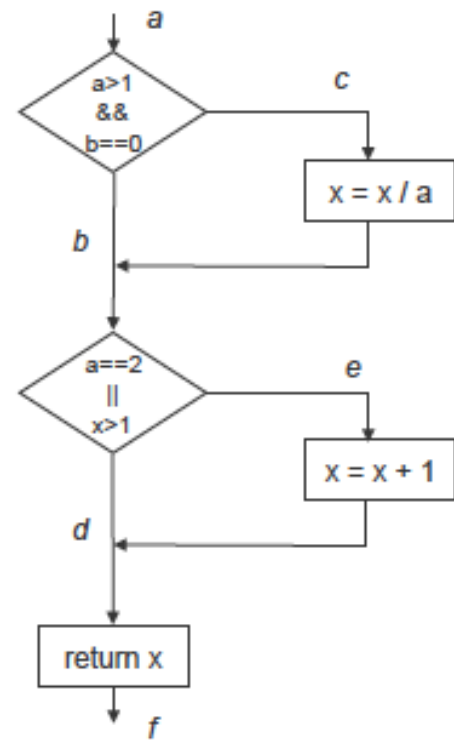


Test en boîte blanche

```
int foo (int a, int b, int x) {  
    if ((a>1) && (b==0))  
        x = x div a;  
    if ((a==2) || (x>1))  
        x = x + 1;  
    return x;  
}
```

C code

Flow control
graph

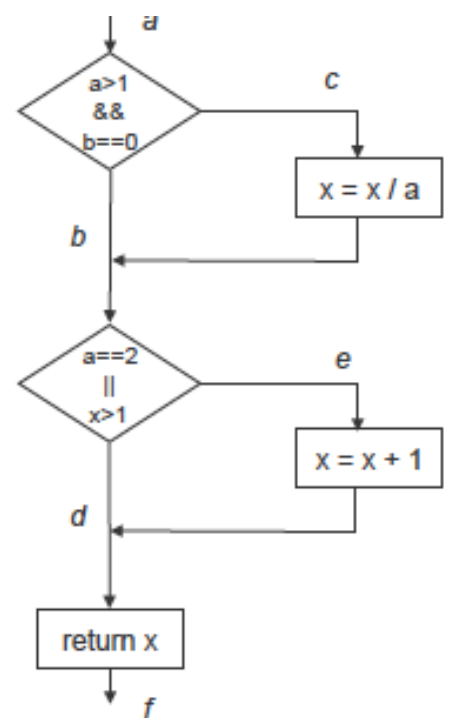


Test en boîte blanche

► Couverture des instructions

- Exemple : acef
- Données de test : $a=2, b=0, x=3$
- Question : Et si le `&&` était un `||` ?
- Question : Et le chemin abdf ?

- Très peu exhaustive, peu utilisé !

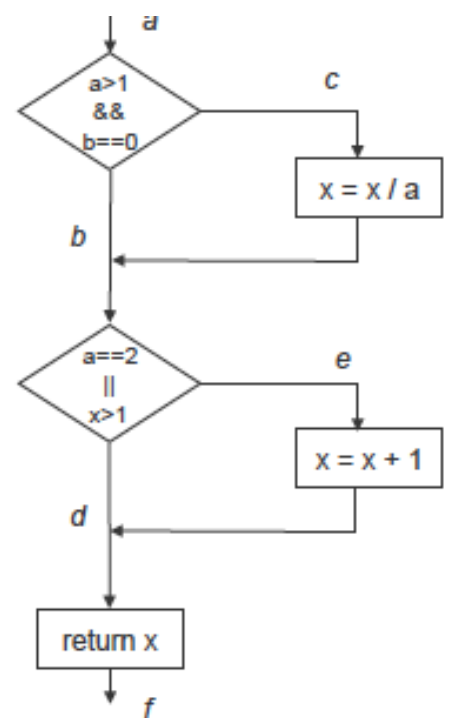


Test en boîte blanche

► Couverture des décisions

= chaque décision est vraie ou fausse au moins une fois

- Exemple : {acef,abdf} ou {acdf,abef}
- Données de test :
 - $a=2, b=0, x=3$
 - $a=2, b=1, x=1$
- Question : Et si $x > 1$ était incorrecte ?



Test en boîte blanche

- ▶ Couverture de plusieurs décisions
= toutes les combinaisons de valeurs pour les décisions sont testées au moins une fois

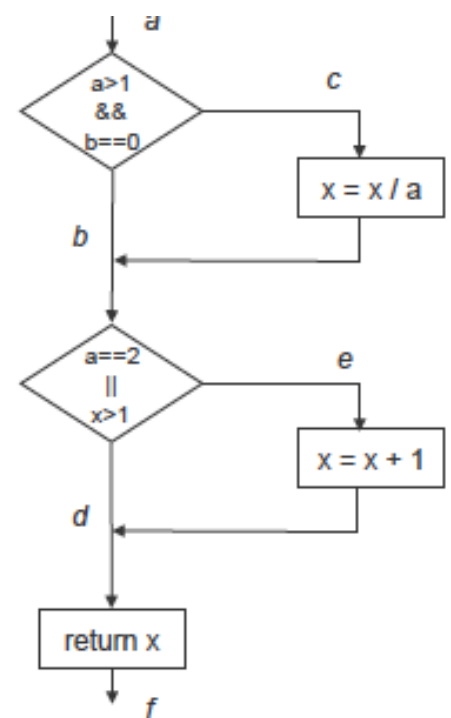
- ▶ Exemple :

1. $a > 1, b = 0$
2. $a = 2, x > 1$
3. $a > 1, b \neq 0$
4. $a = 2, x \leq 1$
5. $a \leq 1, b = 0$
6. $a \neq 2, x > 1$
7. $a \leq 1, b \neq 0$
8. $a \neq 2, x \leq 1$

- ▶ Données de test :

- ▶ $a = 2, b = 0, x = 4$ couvre 1 & 2
- ▶ $a = 2, b = 1, x = 1$ couvre 3 & 4
- ▶ $a = 1, b = 0, x = 2$ couvre 5 & 6
- ▶ $a = 1, b = 1, x = 1$ couvre 7 & 8

- ▶ Toutes les exécutions sont couvertes ?

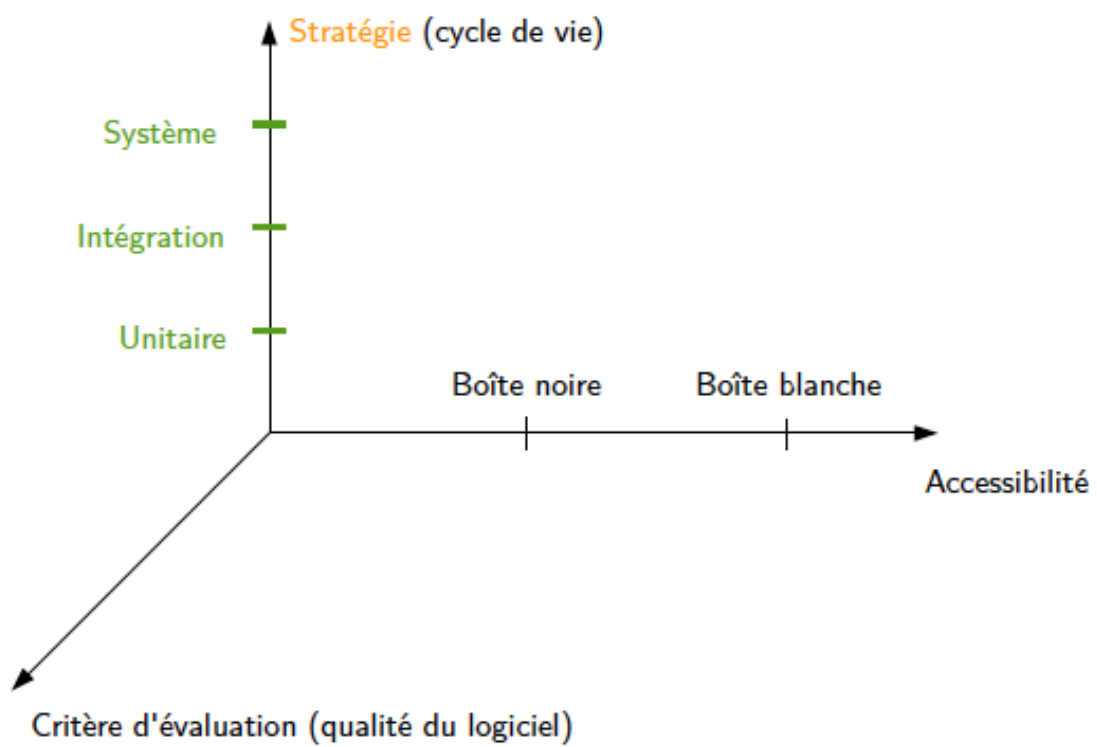


Boîte noire vs. boîte blanche

- ▶ **Complémentarité : détection de fautes différentes**
 - ▶ Boîte noire : détecte les oublis ou les erreurs par rapport à la spécification
 - ▶ Boîte blanche : détecte les erreurs de programmation



Types de test



A hierarchy of tests

Disclaimers:

- there are other hierarchies/taxonomies, on different angles
- **terminology** is not clear cut (as it often happens in SWE)
- the **granularity trend**—from small to big—however matters and is agreed upon

Test hierarchy

- **acceptance**
*Does the **whole system** work?*
(AKA: system tests)
- **integration**
*Does **our code** work against (other) code (we can't change)?*
- **unit**
*Do **our code units** (i.e., classes, objects, modules, etc.) do the right thing and are convenient to work with?*

Acceptance test

Does the whole system work?

Acceptance tests represent **features** that the system should have. Both their lack and their misbehaviour imply that the system is not working as it should. Intuition:

- 1 feature → 1+ acceptance test(s)
- 1 user story → 1+ acceptance test(s) (when using **user stories**)

Exercise (name 2+ acceptance tests for this “user login” story)

After creating a user, the system will know that you are that user when you login with that user's id and password; if you are not authenticated, or if you supply a bad id/password pair, or other error cases, the login page is displayed. If a CMS folder is marked as requiring authentication, access to any page under that folder will result in an authentication check.

<http://c2.com/cgi/wiki?AcceptanceTestExamples>

Preview: we will use acceptance tests to guide feature development

Unit test

Do our code units do the right thing and are convenient to work with?

Before implementing any unit of our software, we have (to have) an idea of **what the code should do**. Unit tests show **convincing evidence** that—in a limited number of cases—it is actually the case.¹

Example (some unit tests for a `List` module)

- calling `List.length` on an empty list returns 0
- calling `List.length` on a singleton list returns 1
- calling `List.last` after `List.append` returns the added element
- calling `List.head` on an empty list throws an exception
- calling `List.length` on the concatenation of two lists returns the sum of the respective `List.lengths`
- ...

1. remember: tests reveal bugs, but don't prove their absence!

Integration test

Does our code work against (other) code (we can't change)?

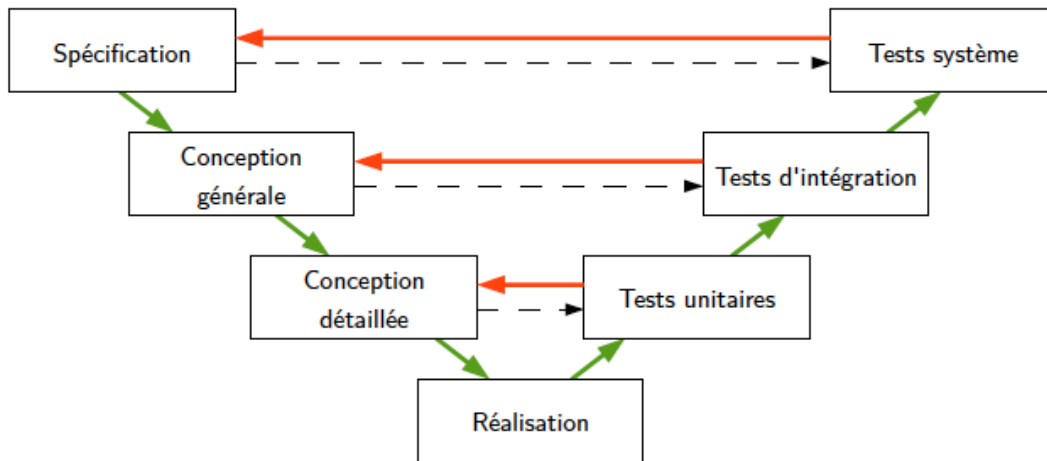
“Code we can't change” =

- 3rd party libraries/framework
 - ▶ be them proprietary or Free/Open Source Software
- code developed by other teams that we don't “own”
 - ▶ (strict code ownership is bad, though)
- code that we do not want/cannot modify in the current phase of development, for whatever reason

Example

- our BankClient should not call the getBalance method on BankingService before calling login and having verified that it didn't throw an exception
- xmlInitParser should be called before any other parsing function of libxml2
- the DocBook markup returned by CMSEditor.save should be parsable by PDFPublisher's constructor

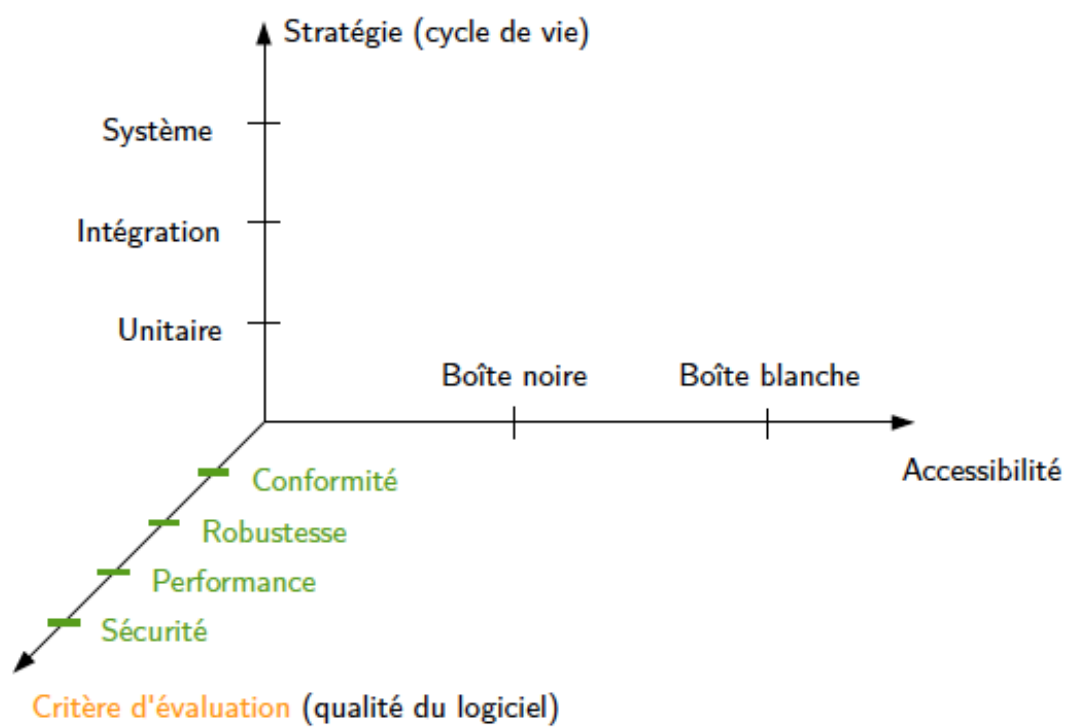
Phases de production d'un logiciel



- ▶ Test unitaire = test des (petites) parties du code, séparément.
- ▶ Test d'intégration = test de la composition de modules.
- ▶ Test du système = de la conformité du produit fini par rapport au cahier des charges, effectué en boîte noire.



Types de test



Test de conformité

- ▶ But : Assurer que le système présente les fonctionnalités attendues par l'utilisateur
- ▶ Méthode : Sélection des tests à partir de la spécification, de façon à contrôler que toutes les fonctionnalités spécifiées sont implantées selon leurs spécifications
- ▶ *Ex : Service de paiement en ligne*
 - ▶ Scénarios avec transaction acceptée/refusée, couverture des différents cas et cas d'erreur prévus



Test de robustesse

- ▶ But : Assurer que le système supporte les utilisations imprévues
- ▶ Méthode : Sélection des tests en dehors des comportements spécifiés (entrées hors domaine, utilisation incorrecte de l'interface, environnement dégradé...)
- ▶ *Ex : Service de paiement en ligne*
 - ▶ Login dépassant la taille du buffer
 - ▶ Coupure réseau pendant la transaction



Test de sécurité

- ▶ But : Assurer que le système ne possède pas de vulnérabilités permettant une attaque de l'extérieur
- ▶ Méthode : Simulation d'attaques pour découvrir les faiblesses du système qui permettraient de porter atteinte à son intégrité
- ▶ *Ex : Service de paiement en ligne*
 - ▶ Essayer d'utiliser les données d'un autre utilisateur
 - ▶ Faire passer la transaction pour terminée sans avoir payé

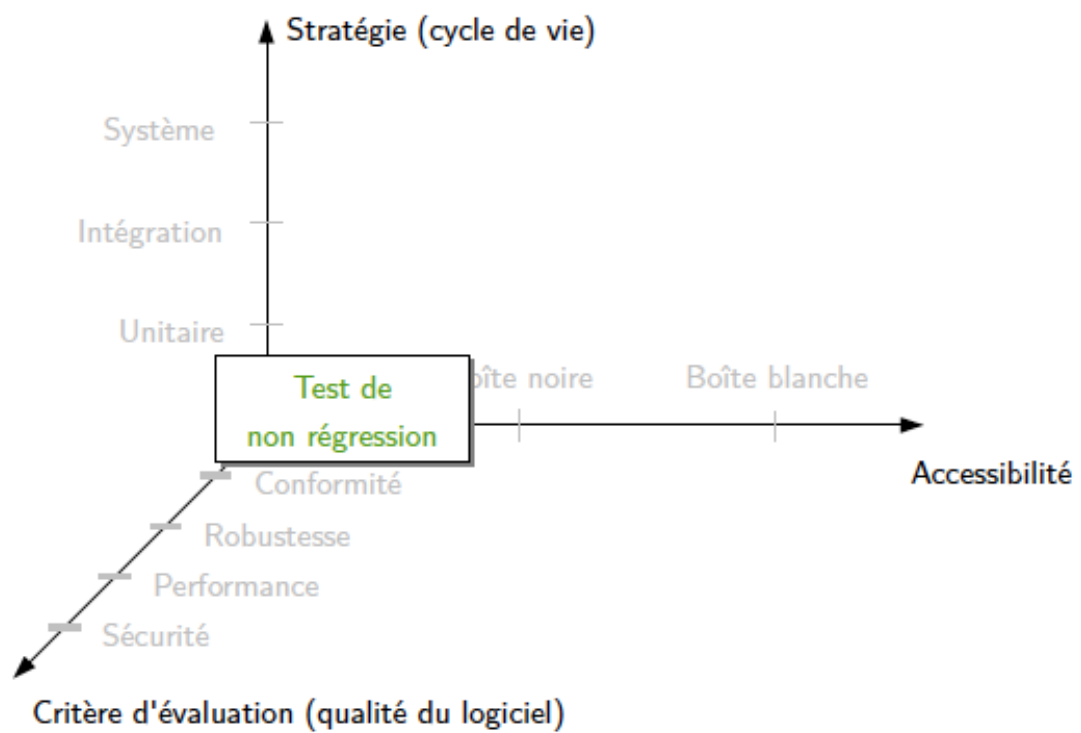


Test de performance

- ▶ But : Assurer que le système garde des temps de réponse satisfaisants à différents niveaux de charge
- ▶ Méthode : Simulation à différents niveaux de charge d'utilisateurs pour mesurer les temps de réponse du système, l'utilisation des ressources...
- ▶ *Ex : Service de paiement en ligne*
 - ▶ Lancer plusieurs centaines puis milliers de transactions en même temps



Types de test



Test de non régression

- ▶ But : Assurer que les corrections et les évolutions du code n'ont pas introduit de nouveaux défauts
- ▶ Méthode : À chaque ajout ou modification de fonctionnalité, rejouer les tests pour cette fonctionnalité, puis pour celles qui en dépendent, puis les tests des niveaux supérieurs
 - ▶ Lourd mais indispensable
 - ▶ Automatisable en grande partie



Automatisation des tests

▶ Outils :

- ▶ Générateur de test : aléatoire ou guidé par propriétés (BN) ou par des critères de couverture (BB).
 - ▶ Microsoft : DART, SAGE
 - ▶ INRIA : TGV
- ▶ Analyseur de couverture : calcule le pourcentage de code couvert durant le test.
 - ▶ Coverlipse, gcov
- ▶ “Record & playback” (Exécutif de test) : enregistre les actions de l'utilisateur pour pouvoir les rejouer à la demande ; utile pour le test des IHM et le test de régression.
- ▶ Gestionnaire de test : maintient des suites de test, leurs résultats et produit des rapports.
 - ▶ Xunit (avec X=C, Java, Python)

