Introduction au test de logiciels

Delphine Longuet

delphine.longuet@lri.fr

Pourquoi vérifier et valider ?

Pour éviter les bugs

- Sonde Mariner 1, 1962 : problème de spécification
  `http://fr.wikipedia.org/wiki/Mariner_1`
- Ariane V vol 501, 1996 : problème de conversion de nombres

  Coût : 370 millions de dollars
  `http://fr.wikipedia.org/wiki/Vol_501_d'Ariane_5`
- Therac-25, 1985-87 : problème de synchronisation

  Coût : plusieurs vies
  `http://fr.wikipedia.org/wiki/Therac-25`
- Panne d'électricité aux États-Unis et au Canada, 2003

  Impact sur 55 millions d'habitants. Coût : 6 milliards de dollars

Pourquoi vérifier et valider ?

Pour éviter les bugs

- Pour l'utilisateur : coût économique, humain, environnemental
- Pour le fournisseur : coût de la correction des bugs

en phase d'implantation	coût 1
en phase d'intégration (bug de conception)	coût 10
en phase de recette (bug de spécification)	coût 100
en phase d'exploitation	coût $>$ 1000

Pourquoi vérifier et valider ?

Pour assurer la qualité

- Capacité fonctionnelle : réponse aux besoins des utilisateurs
- Facilité d'utilisation : prise en main et robustesse
- Fiabilité : tolérance aux pannes
- Performance : temps de réponse, débit, fluidité...
- Maintenabilité : facilité à corriger ou transformer le logiciel
- Portabilité : aptitude à fonctionner dans un environnement différent de celui prévu

Pourquoi des méthodes pour vérifier et valider ?

Pour réduire le coût

- Vérification et validation :
  - environ 30% du développement d'un logiciel standard
  - plus de 50% du développement d'un logiciel critique

- Phase de test souvent plus longue que les phases de spécification, conception et implantation réunies

Méthodes de vérification et validation

Vérification : assurer que le système fonctionne correctement

« Are we building the product right ? »

Preuve formelle de propriétés sur un modèle du système

↳ model-checking, preuve

Validation : assurer que le système fonctionne selon les attentes de l'utilisateur

« Are we building the right product ? »

Assurance d'un certain niveau de confiance dans le système

↳ test

Méthodes de vérification

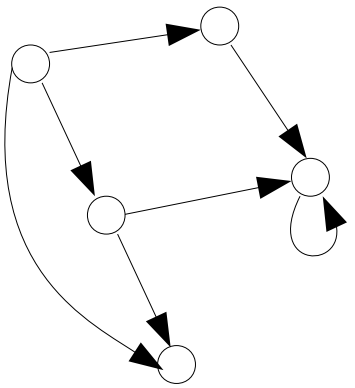
Vérification : Le système vérifie-t-il la propriété ?

Preuve

```
insert(x : integer, l : list) : list
     if l == []
     then [x]
     else if x <= hd(l)
          then x::l
          else hd(l)::insert(x,tl(l))
```

$\vdash$ P

Model-checking

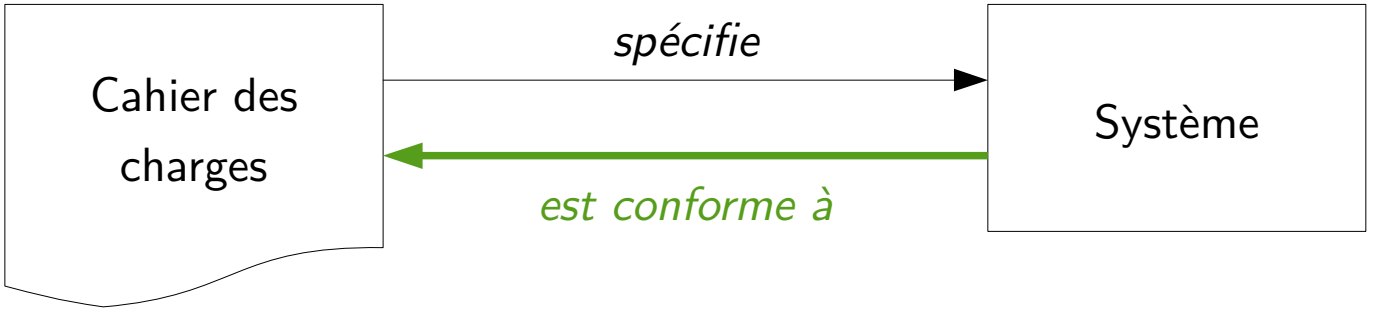


$\models$ P

Objectif : Prouver la correction du système

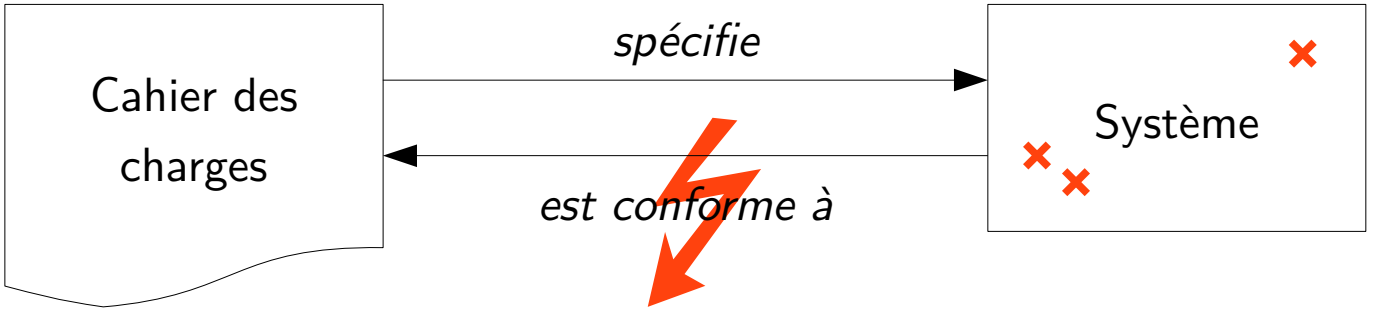
Méthode de validation

Validation : Le système est-il conforme au cahier des charges ?



Méthode de validation

Validation : Le système est-il conforme au cahier des charges ?



Objectif : Détecter les non conformités

Comparaison des méthodes de V&V

Test :

- ✔ Nécessaire : exécution du système réel, découverte d'erreurs à tous les niveaux (spécification, conception, implantation)
- ✗ Pas suffisant : exhaustivité impossible

Preuve :

- ✔ Exhaustif
- ✗ Mise en œuvre difficile, limitation de taille

Model-checking :

- ✔ Exhaustif, partiellement automatique
- ✗ Mise en œuvre moyennement difficile (modèles formels, logique)

Comparaison des méthodes de V&V

Méthodes complémentaires :

- Test non exhaustif mais facile à mettre en œuvre
- Preuve exhaustive mais très technique
- Model-checking exhaustif et moins technique que la preuve

Mais :

- Preuve et model-checking limités par la taille du système et vérifient des propriétés sur un modèle du système (distance entre le modèle et le système réel ?)
- Test repose sur l'exécution du système réel, quelles que soient sa taille et sa complexité

Définitions du test

Norme 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

Définitions du test

« *Tester peut révéler la présence d'erreurs mais jamais leur absence* »

↳ Vérification partielle : le test ne peut pas montrer la conformité du système (nécessité d'une infinité de tests)

« *Tester, c'est exécuter le programme dans l'intention d'y trouver des anomalies ou des défauts* »

↳ Objectif : détection des bugs

Bug ?

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

Limites de la vérification

Indécidabilité : Un problème est indécidable s'il n'existe pas d'algorithme capable de le résoudre dans le cas général

Ex : Problèmes indécidables

- L'exécution d'un programme termine
- Deux programmes calculent la même chose
- Un programme est une implantation correcte de sa spécification

↳ Il n'existe pas d'algorithme permettant de prouver la correction de n'importe quel programme

Limites du test

Explosion combinatoire : Nombre d'exécutions possibles d'un programme potentiellement infini

Mais test = processus fini

↳ Nécessité d'approcher l'infini (l'extrêmement grand) par le fini (heuristique)

Évolution du test

Aujourd'hui, le test de logiciels :

- est la méthode la plus utilisée pour assurer la qualité des logiciels
- fait l'objet d'une pratique trop souvent artisanale

Demain, le test de logiciels devrait être :

- une activité rigoureuse
- fondée sur des modèles et des théories
- de plus en plus automatique

Échauffement

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

Échauffement

Cas valides

triangle scalène valide
triangle isocèle valide + permutations
triangle équilatéral valide
triangle plat (a+b=c) + permutations

Cas invalides

pas un triangle (a+b<c) + permutations
une valeur à 0
toutes les valeurs à 0
une valeur négative
une valeur non entière
mauvais nombre d'arguments

Échauffement

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

Échauffement

16 cas correspondant aux défauts constatés dans des implantations de cette spécification

Moyenne des résultats obtenus par un ensemble de développeurs expérimentés : 55%

↳ La construction de tests est une activité difficile, encore plus sur de grandes applications

Vocabulaire du test

Objectif de test : comportement du système 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

---

Glossary of Testing Terms, ISTQB (International Software Testing Qualifications Board)

Exemple : tri d'une liste d'entiers

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]	

Exemple : tri d'une liste d'entiers

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

Problème de l'oracle

Ex : Trouver le minimum d'une liste d'entiers

Entrée : [4; 2; 3; 6] Sortie attendue : 2

Oracle : Égalité entre entiers OK

Ex : Calculer l'itinéraire le plus rapide entre deux villes

Entrée : Paris – Lyon Sortie attendue : ...A6...

Oracle : Égalité des chemins ? Non

Ex : Problème du sac à dos (résolu avec une heuristique)

Oracle : Résultat raisonnablement éloigné du résultat optimal ?? Non

Problème de l'oracle

Ex : Trouver le minimum d'une liste d'entiers

Entrée : [4; 2; 3; 6] Sortie attendue : 2

Oracle : Égalité entre entiers OK

Ex : Calculer l'itinéraire le plus rapide entre deux villes

Entrée : Paris – Lyon Sortie attendue : ...A6...

Oracle : Trajet de 4h17 (quel que soit l'itinéraire choisi) OK

Ex : Problème du sac à dos (résolu avec une heuristique)

Oracle : Résultat = résultat optimal + 5% OK

Problème de l'oracle

Oracle : En général, résultat attendu = ensemble de conditions si plusieurs solutions possibles et énumération impossible

Risques : Échec d'un programme conforme si définition trop stricte du résultat attendu

↳ Faux positif (*false-fail*)

Voir l'exemple du calcul d'itinéraire dans lequel on impose un chemin.

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

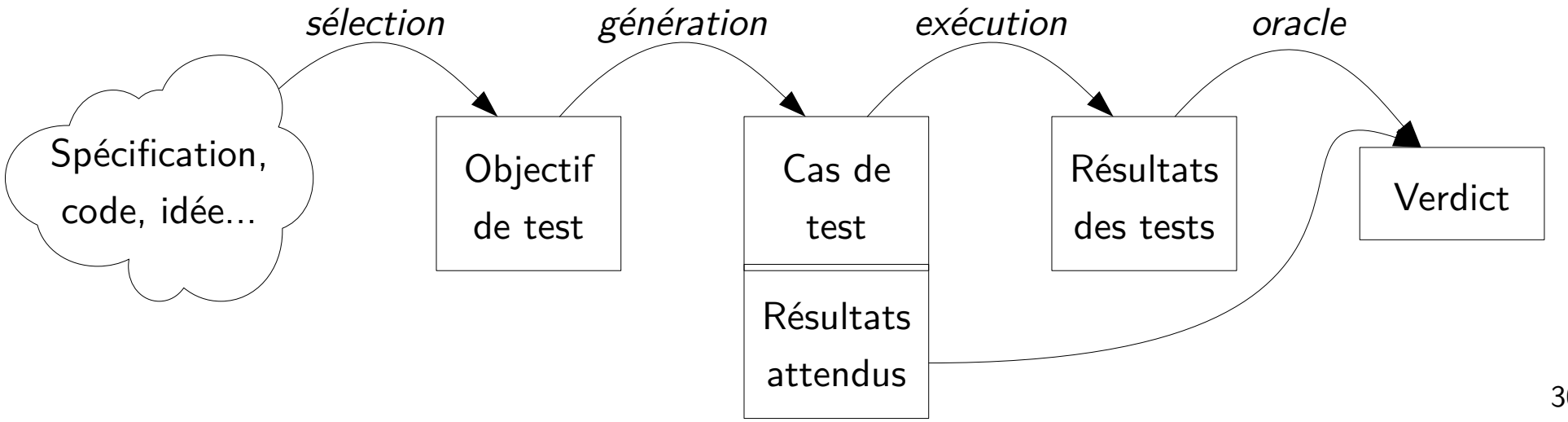
Faux négatif (*false-pass*) : fait passer un programme incorrect

Validité indispensable, complétude impossible en pratique

↳ Toujours s'assurer que les tests sont valides

Processus de test

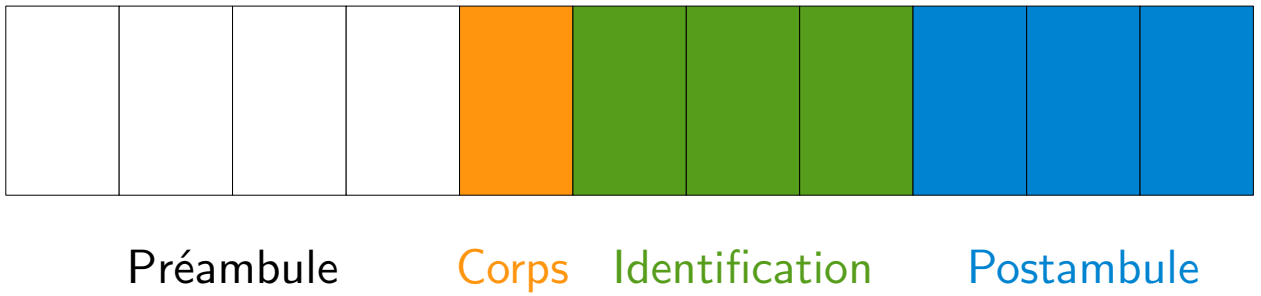
1. Choisir les comportements à tester (objectifs de test)
2. Choisir des données de test permettant de déclencher ces comportements + décrire le résultat attendu pour ces données
3. Exécuter les cas de test sur le système + collecter les résultats
4. Comparer les résultats obtenus aux résultats attendus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



Exécution d'un test

Scénario de test :

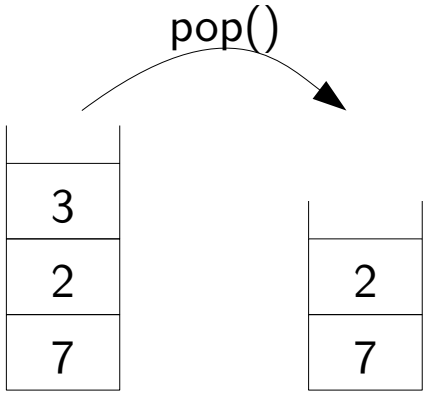
- Préambule : Suite d'actions amenant le programme dans l'état nécessaire pour exécuter le cas de test
- Corps : Exécution des fonctions du cas de test
- Identification (facultatif) : Opérations d'observation rendant l'oracle possible
- Postambule : Suite d'actions permettant de revenir à un état initial



Exécution d'un test

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

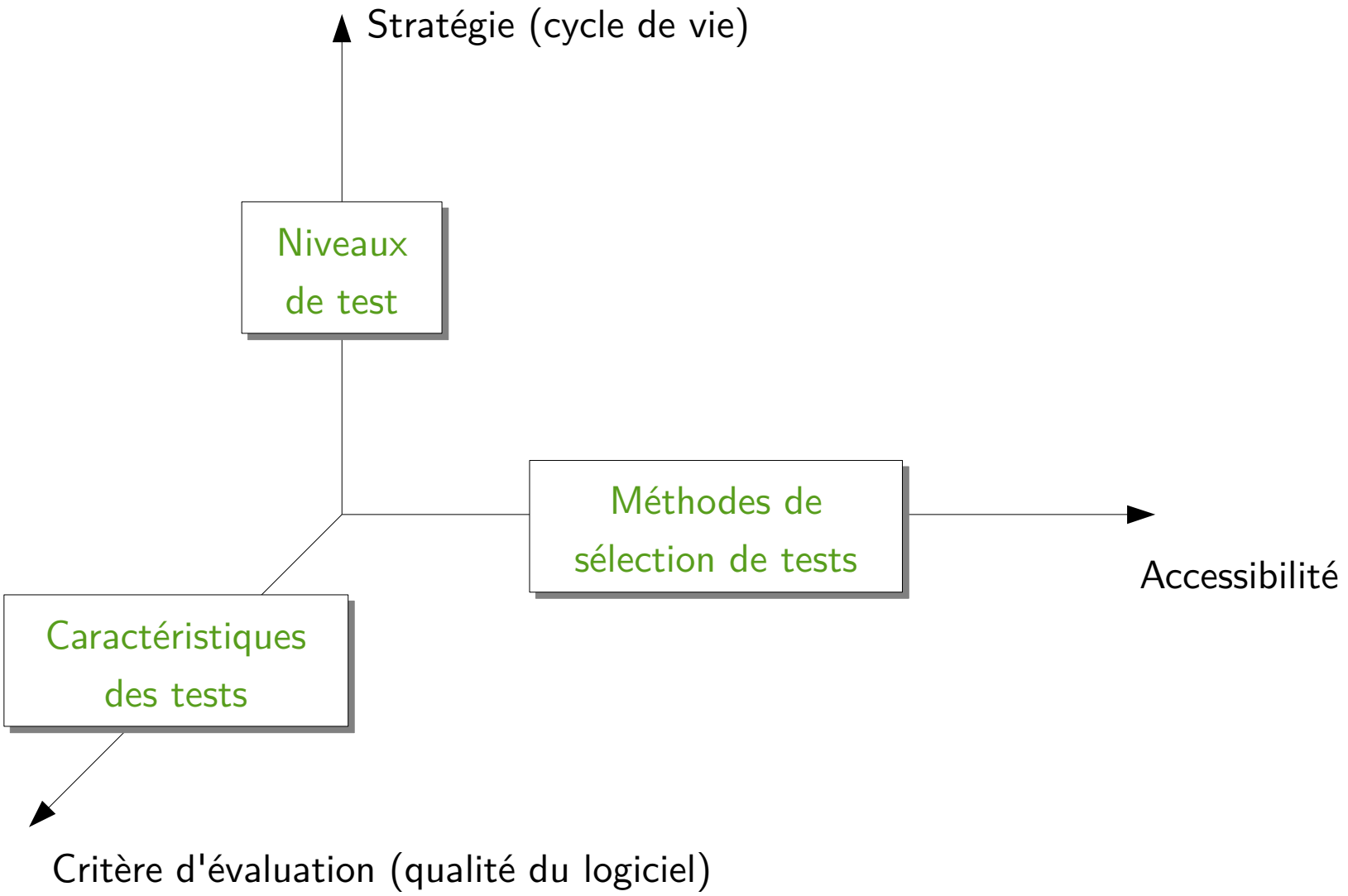
Cas de test :



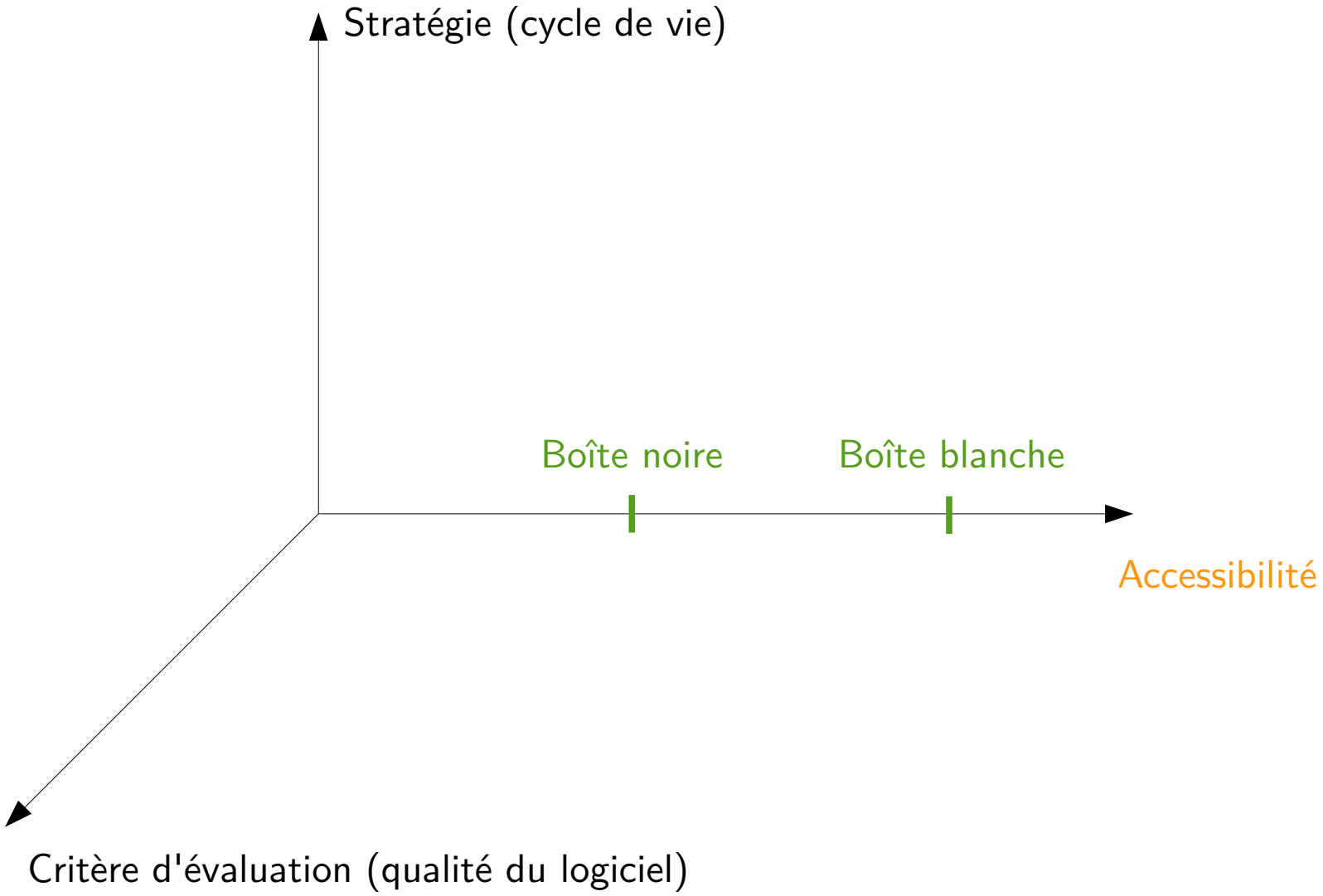
Exécution du test :

Préambule	push(7)
	push(2)
	push(3)
Corps	pop()
Identification	top() = 2
	pop()
	top() = 7
	pop()
	top() = *empty*

Types de tests

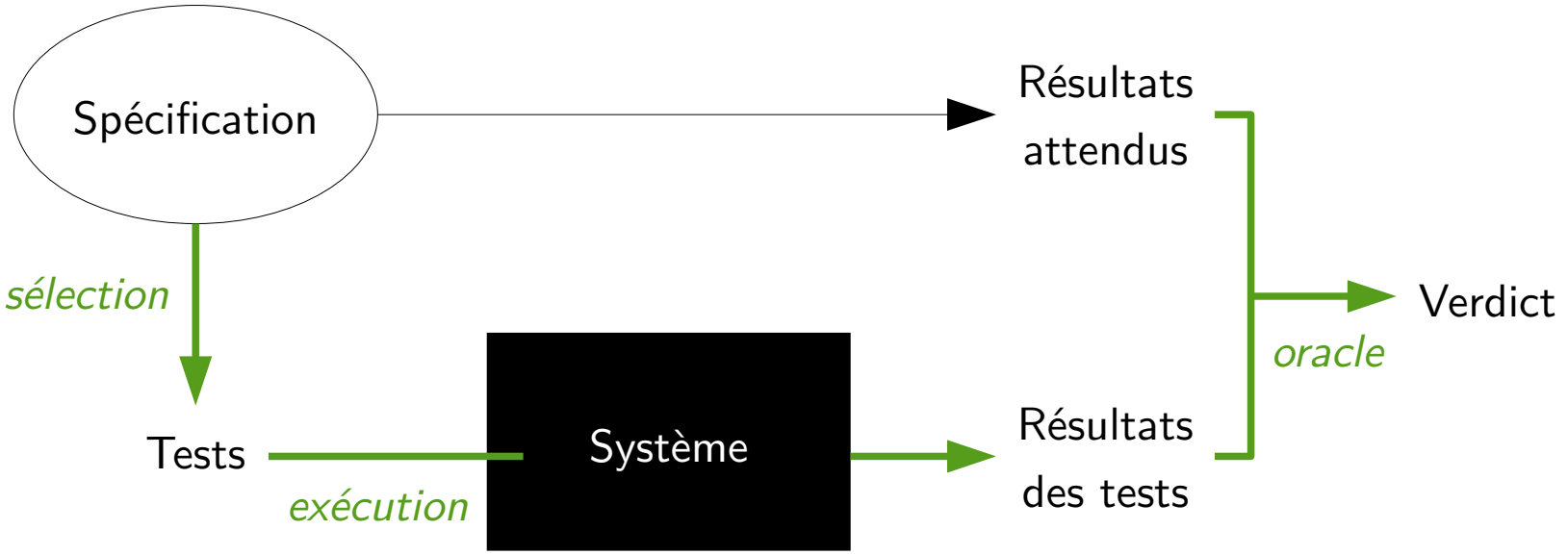


Types de tests



Test boîte noire

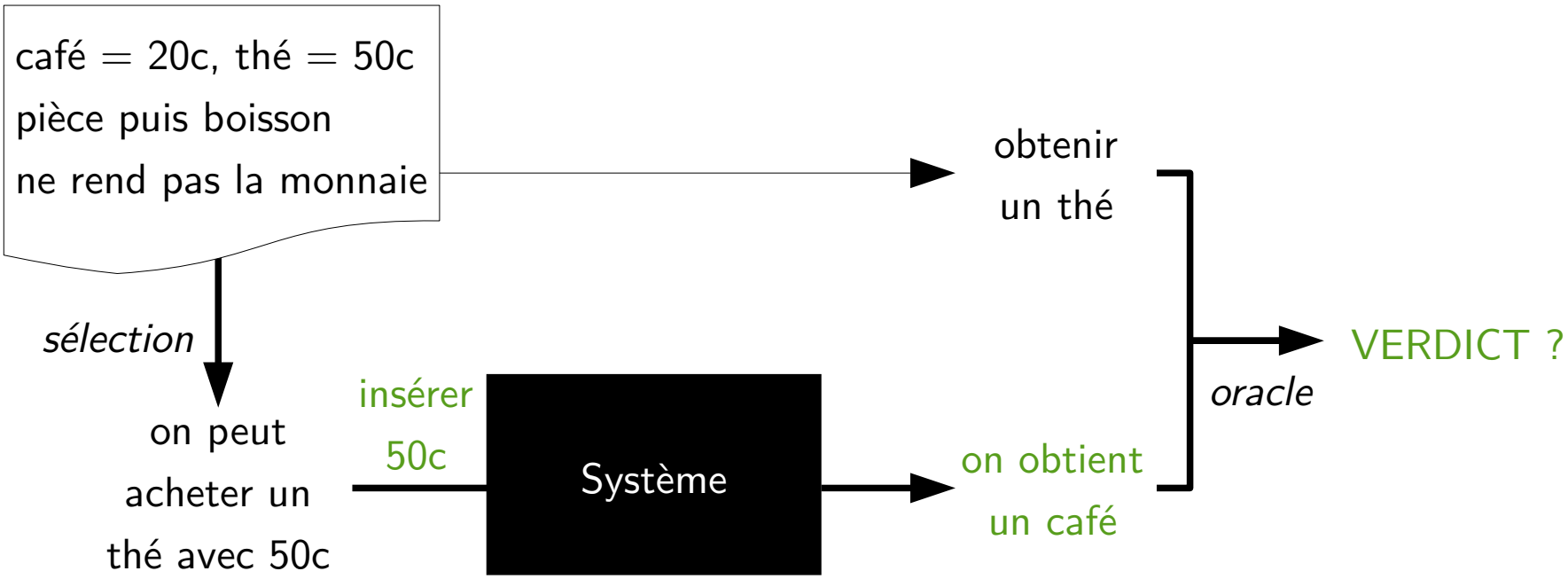
Sélection des tests à partir d'une spécification du système (formelle ou informelle), sans connaissance de l'implantation



Possibilité de construire les tests pendant la conception, avant le codage

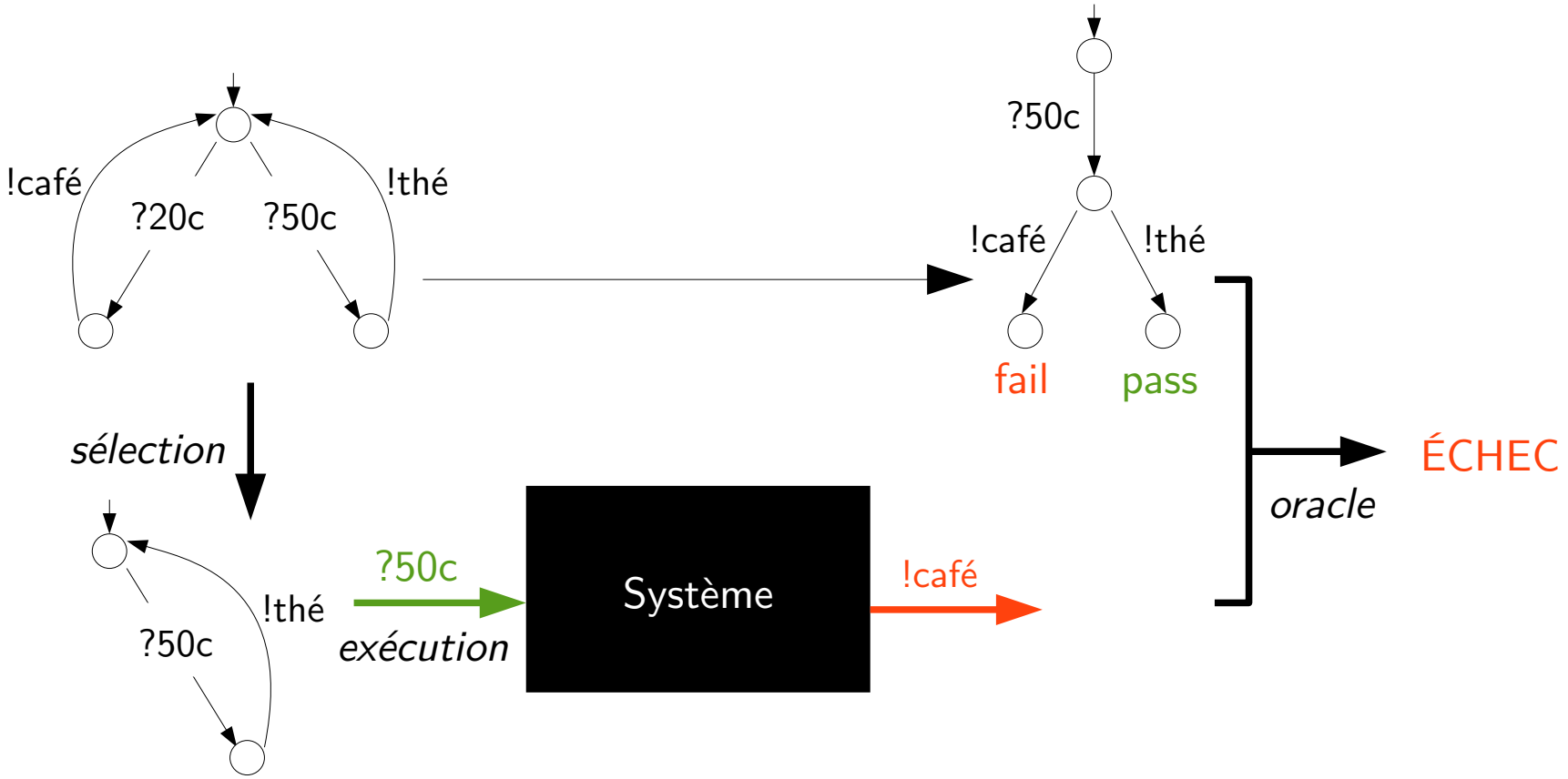
Test boîte noire

Ex : Distributeur de café et thé



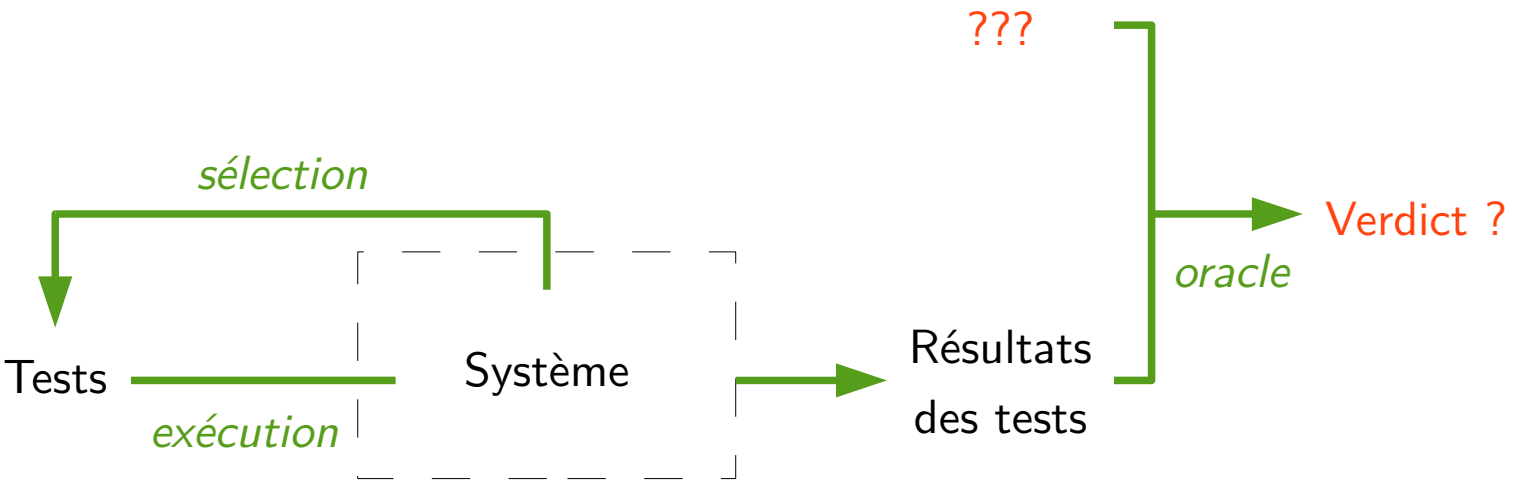
Test boîte noire

Ex : Distributeur de café et thé



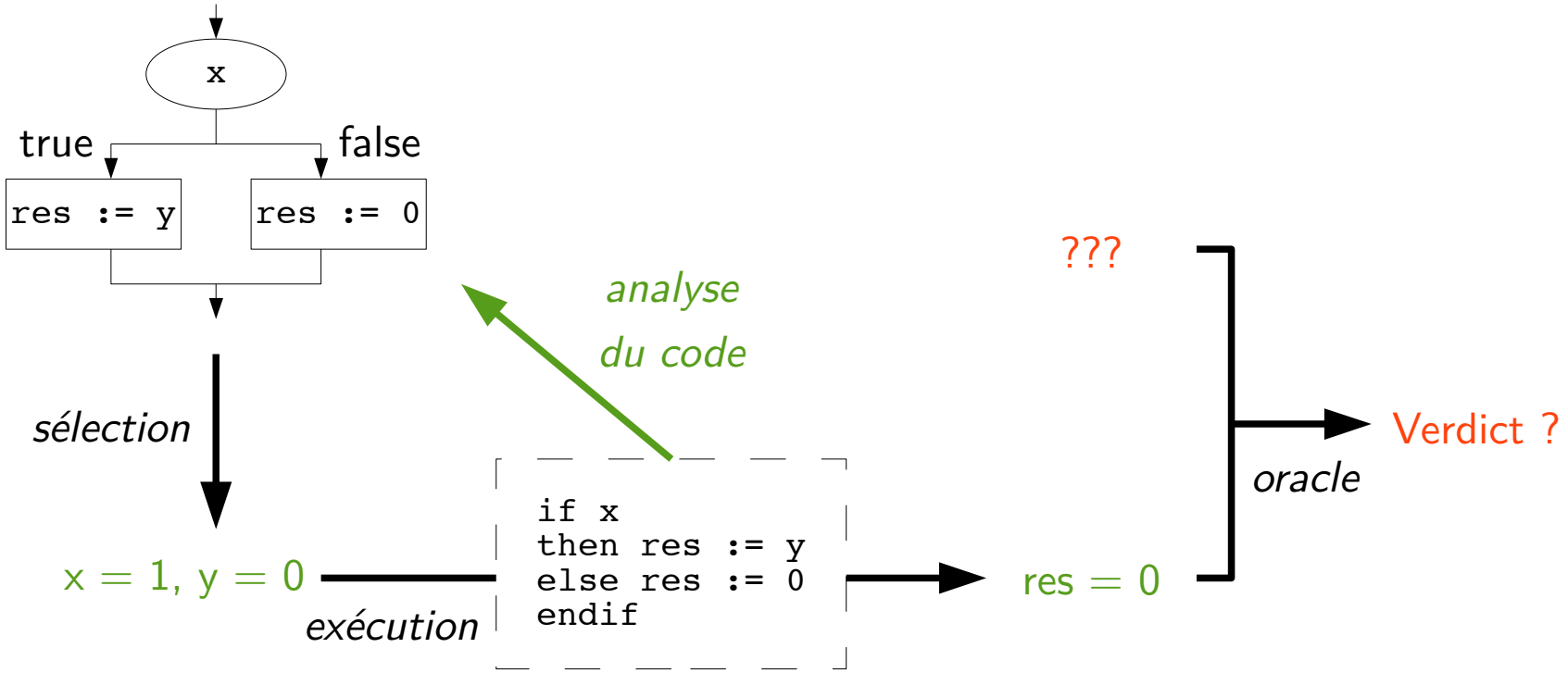
Test boîte blanche

Sélection des tests à partir de l'analyse du code source du système

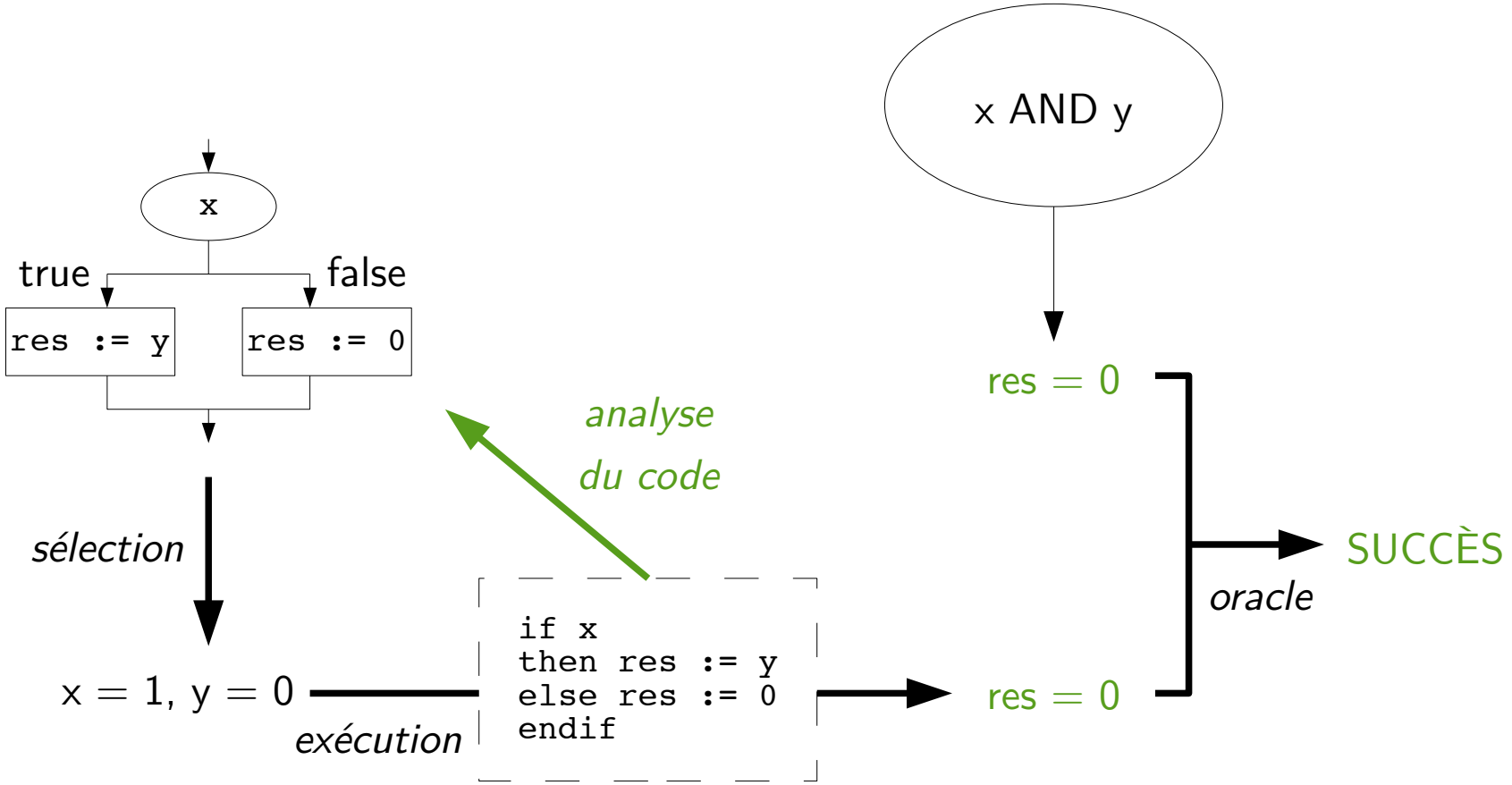


Construction des tests uniquement pour du code déjà écrit

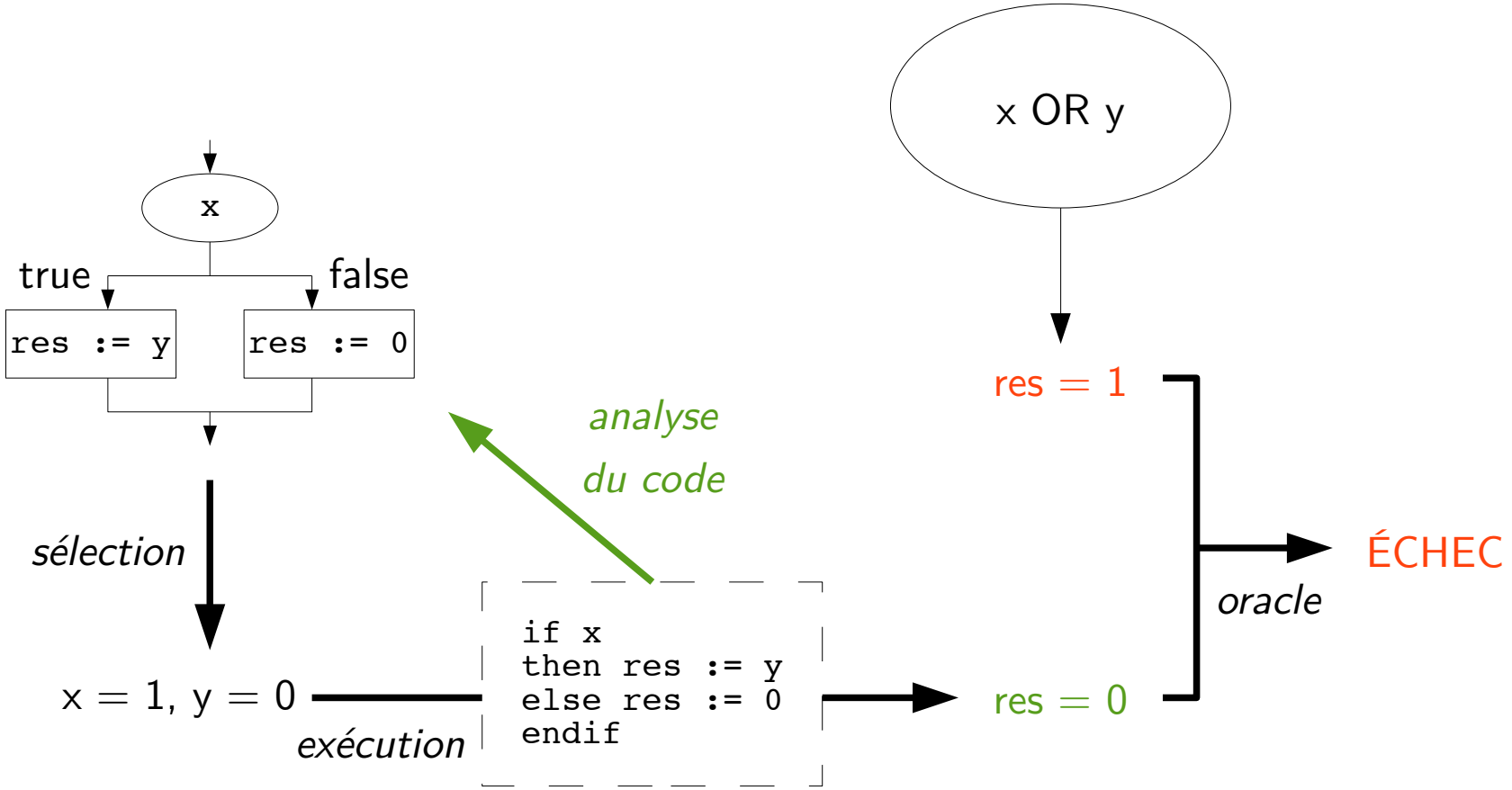
Test boîte blanche



Test boîte blanche

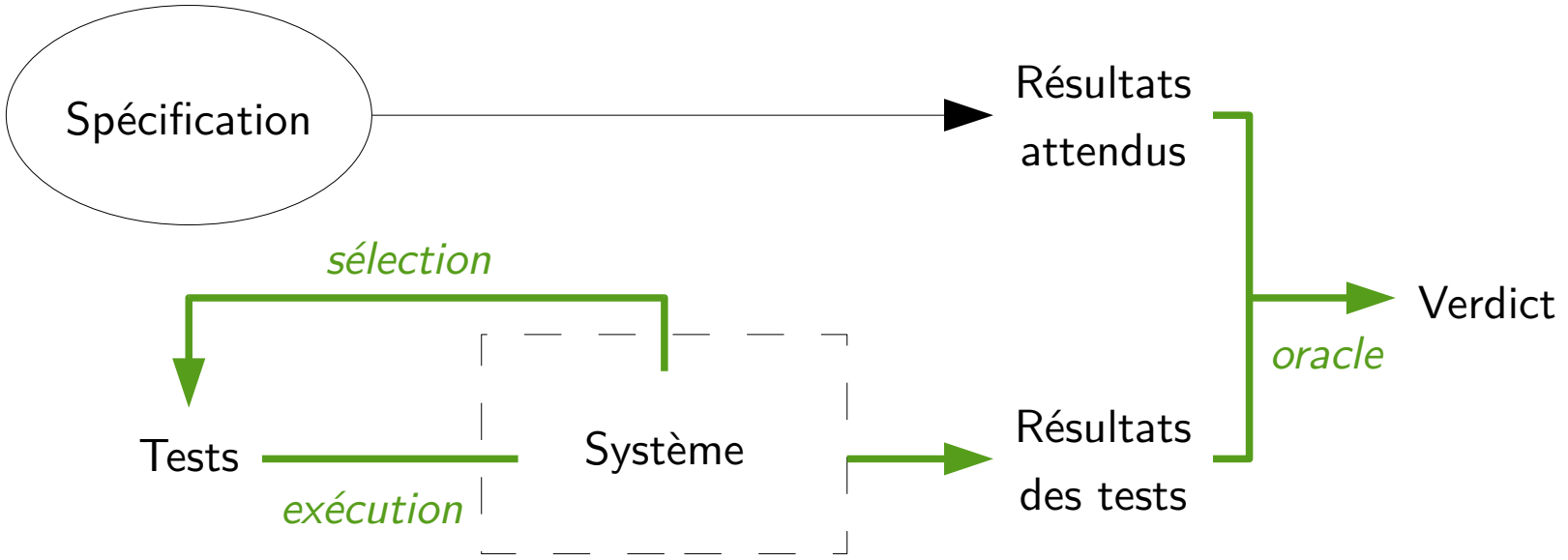


Test boîte blanche



Test boîte blanche

Sélection des tests à partir de l'analyse du code source du système



Construction des tests uniquement pour du code déjà écrit

Boîte noire vs. boîte blanche

Complémentarité : détection de fautes différentes

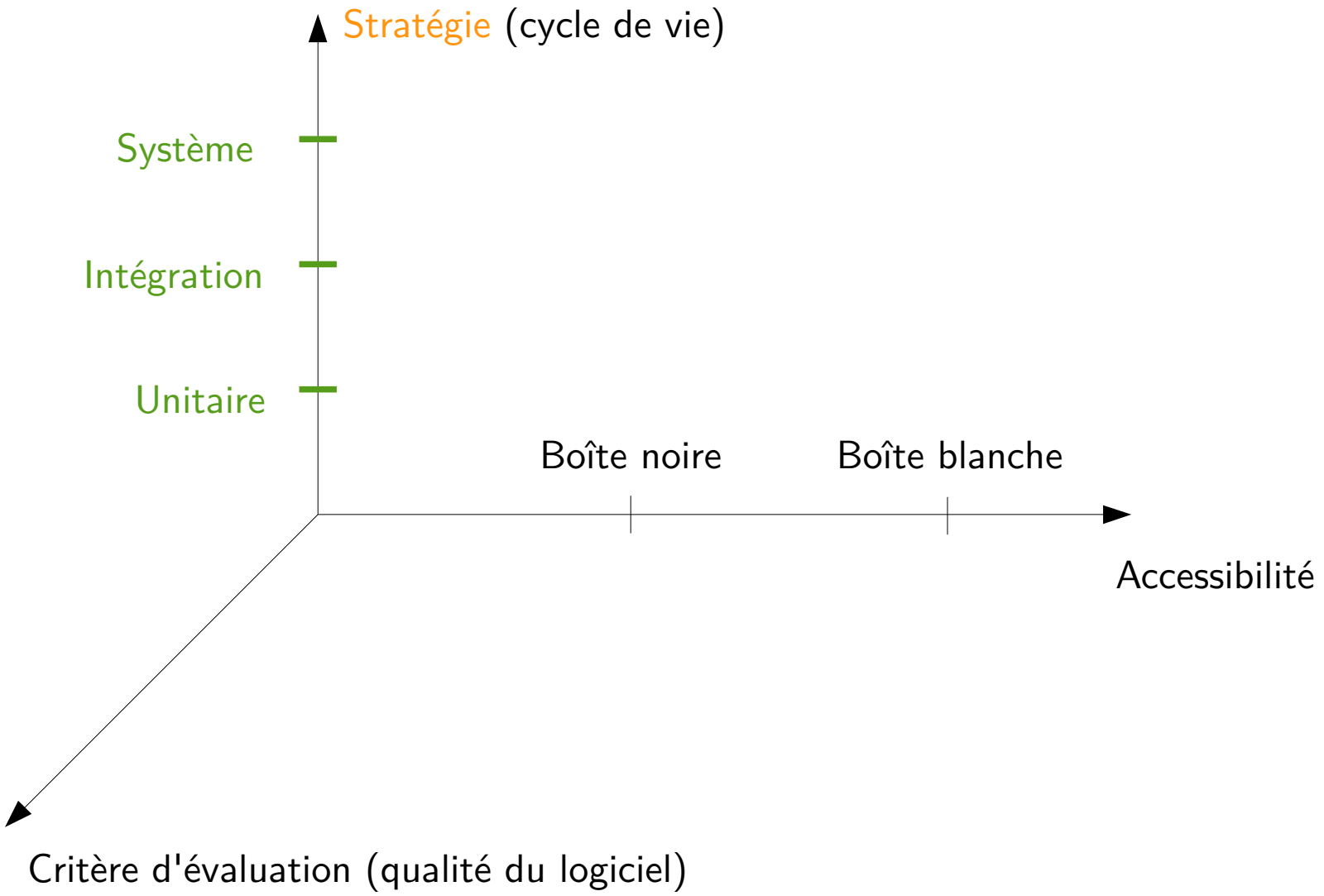
- Boîte noire : détecte les oublis ou les erreurs par rapport à la spécification
- Boîte blanche : détecte les erreurs de programmation

Ex : Addition d'entiers modulo 100 000

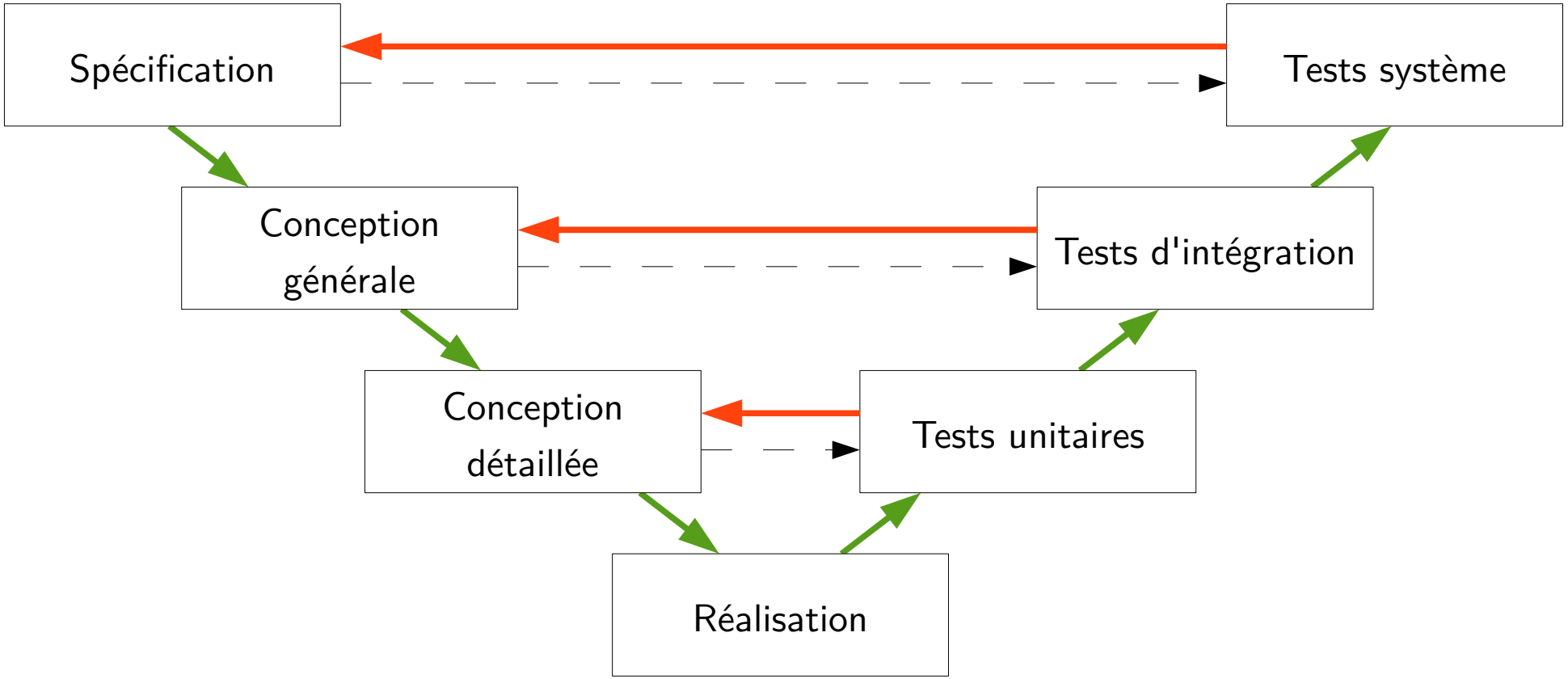
```
Function sum(x,y : integer) : integer
   if (x = 600 and y = 500)
   then sum := x - y
   else sum := x + y
```

- Boîte noire : détecte l'erreur par rapport à la spécification
- Boîte blanche : détecte l'erreur pour les valeurs (600,500)

Types de tests



Cycle de vie du logiciel



Test unitaire

Test des unités de programme de façon isolée, indépendamment les unes des autres, c'est-à-dire sans appel à une fonction d'un autre module, à une base de données...

↳ méthodes, classes, modules, composants

Ex : GPS

- Algorithme de calcul d'itinéraire sur des exemples de graphes construits à la main

Test d'intégration

Test de la composition des modules via leur interface

↳ communications entre modules, appels de procédures...

Ex : GPS

- Lecture des données depuis la base de données
- Communications avec l'IHM

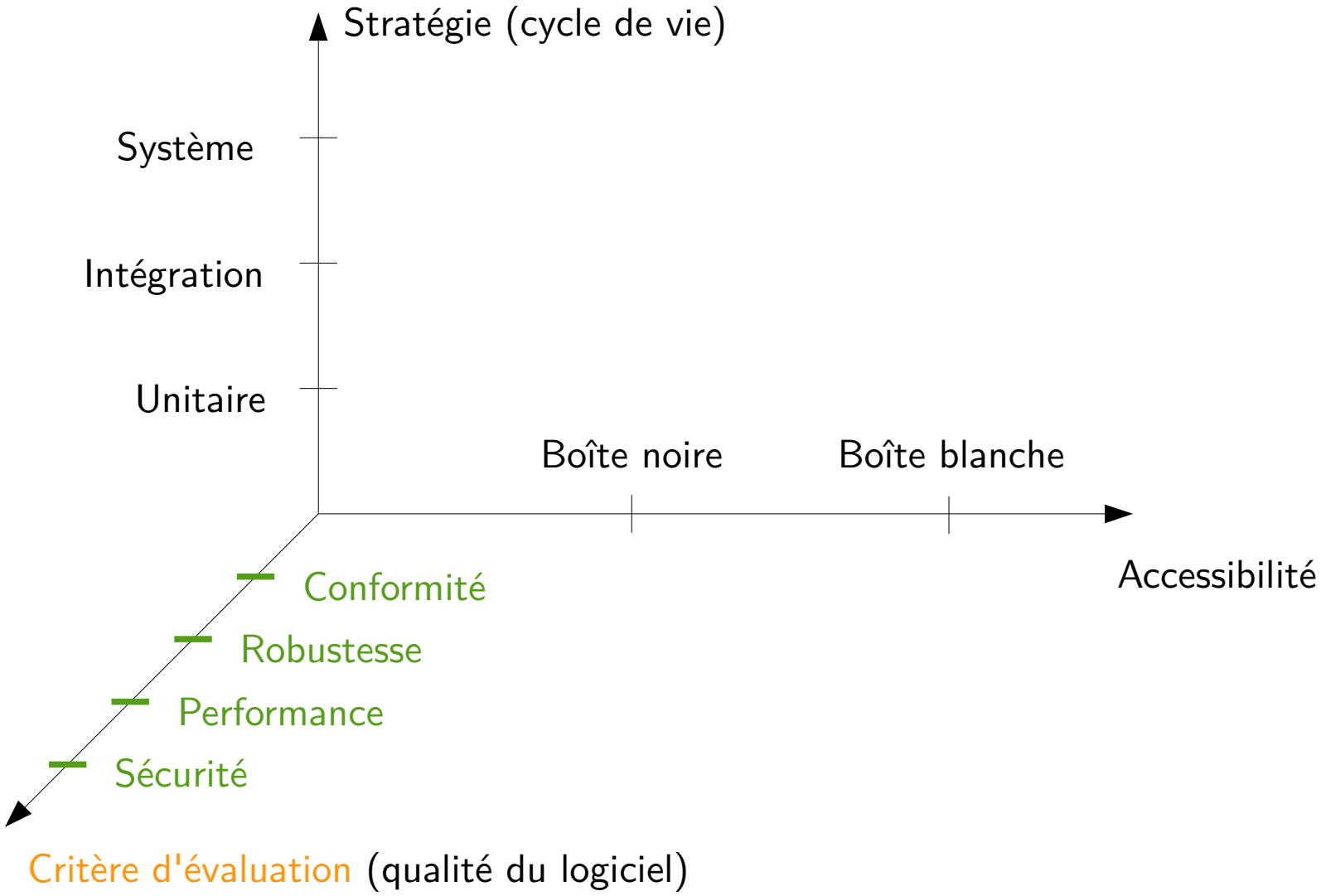
Test système

Test de la conformité du produit fini par rapport au cahier des charges, effectué en boîte noire au travers de son interface

Ex : GPS

- Utilisation du logiciel sur des scénarios réalistes et complets

Types de tests



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

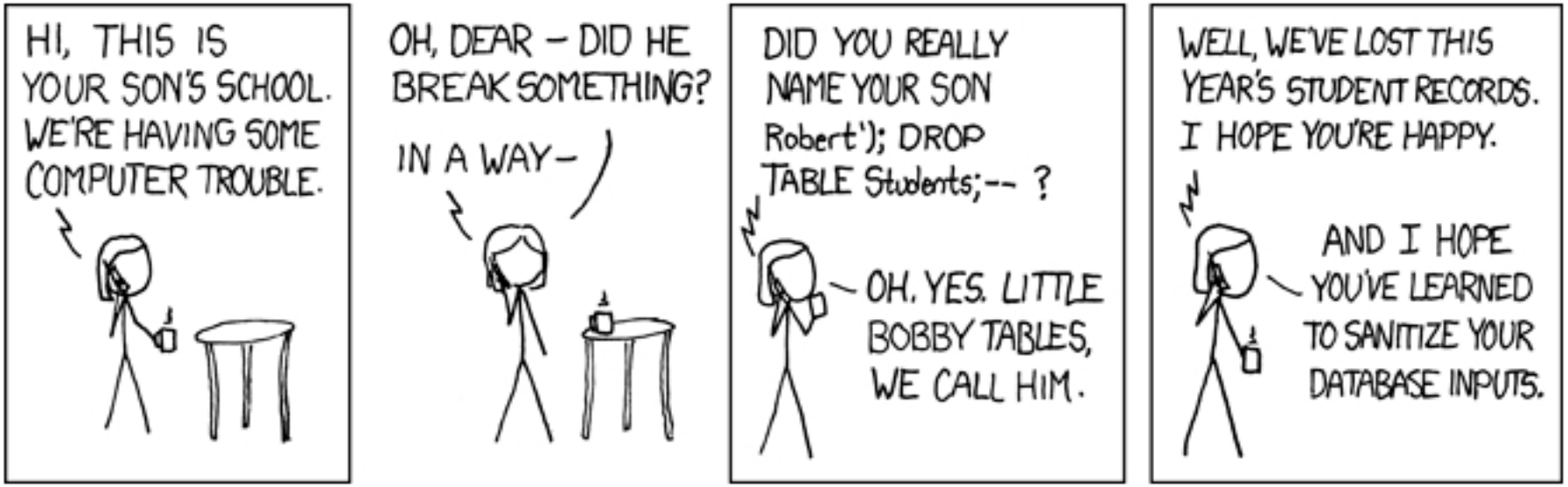
But : Assurer que le système ne possède pas de vulnérabilités permettant une attaque de l'extérieur

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 robustesse/sécurité



Test de performance

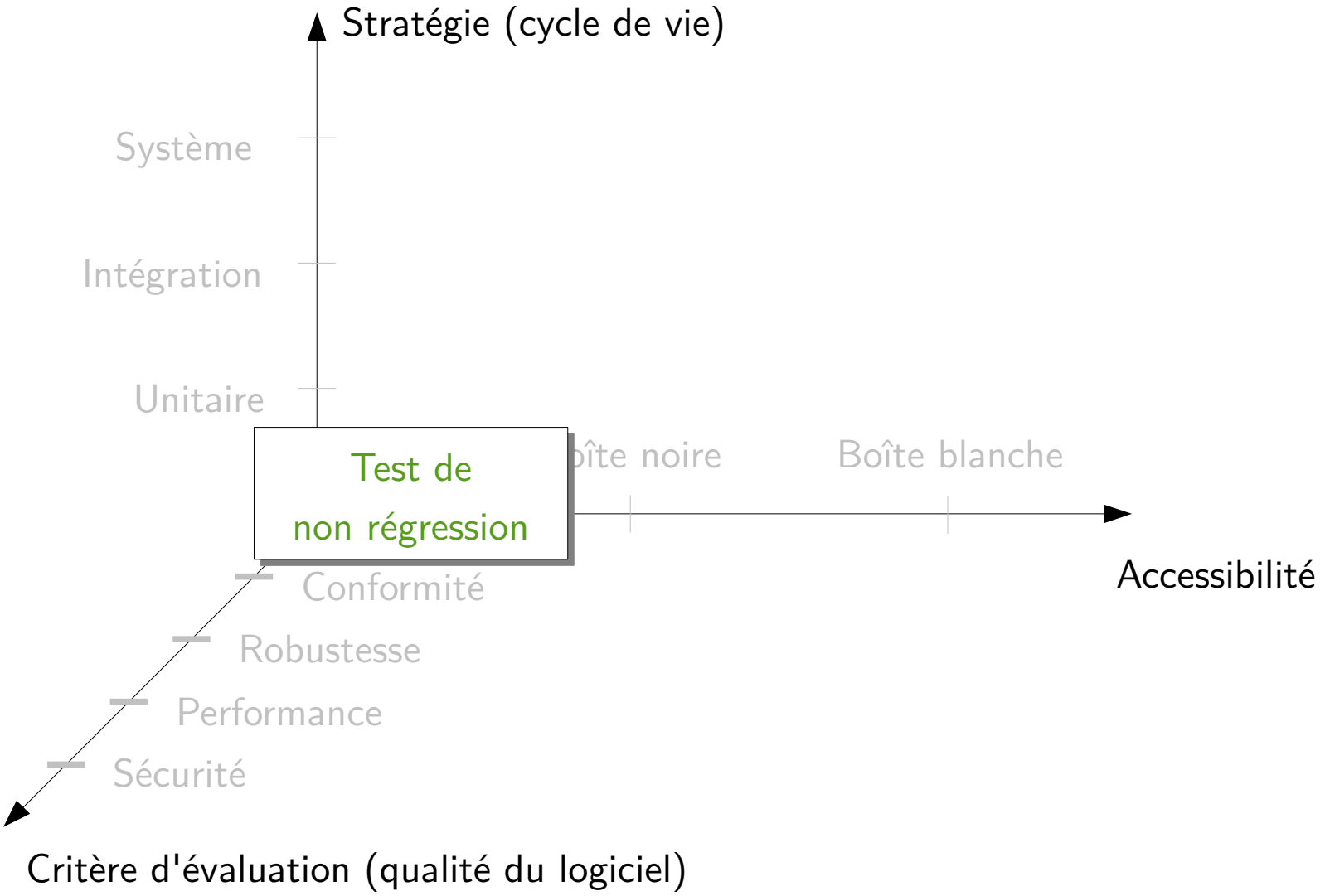
But : Assurer que le système garde des temps de réponse satisfaisants à différents niveaux de charge

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

Un type de test transversal



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