Polytech Paris-Sud
Formation initiale 4e année
Spécialité informatique
Année 2016-2017

Vérification et validation

Cours 4

Introduction au test et méthodes informelles

Delphine Longuet
delphine.longuet@lri.fr

http://www.lri.fr/~longuet/Enseignements/16-17/Et4-VV

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

Un bon test est un test qui permet de découvrir un défaut en déclenchant une anomalie

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.

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	(2,3,5)/(1,1,2)	scalène/isocèle

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 pour 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...)

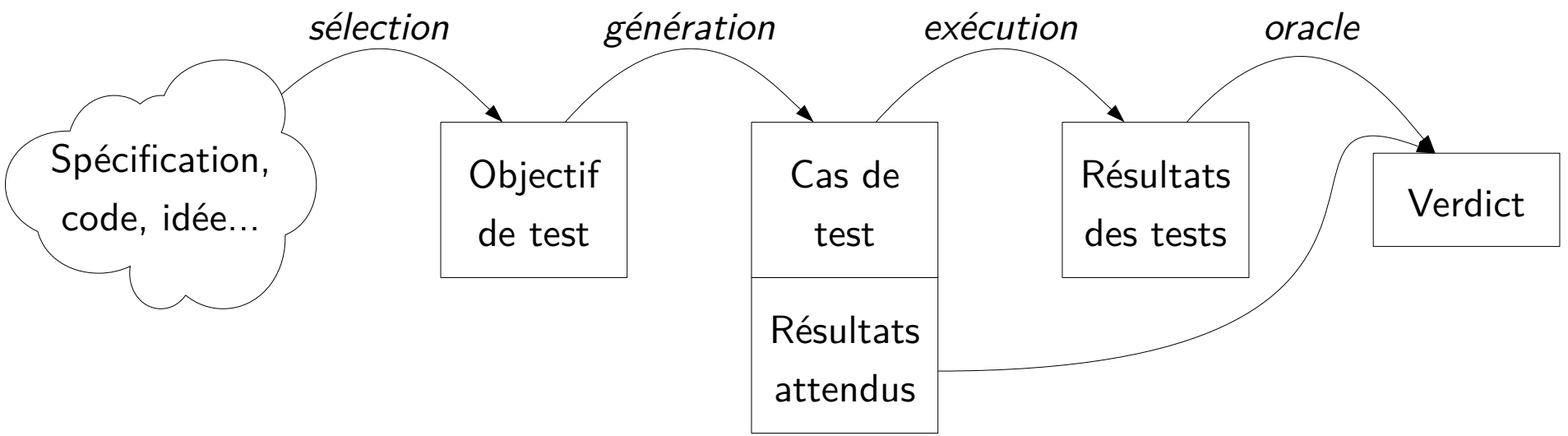
↳ Cas de test : données d'entrée et résultats attendus associés à un objectif de test

---

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

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



Problème de l'oracle

Oracle : décision de la réussite de l'exécution d'un test, en comparant le résultat obtenu au résultat attendu

Problème : décision pouvant être complexe

- types de données sans prédicat d'égalité
- système non déterministe : sortie possible mais pas celle attendue
- heuristique : approximation du résultat optimal attendu

Solution : description du résultat attendu, selon les cas :

- la valeur attendue
- énumération des valeurs possibles
- ensemble de conditions

Problème de l'oracle

Trouver le minimum d'une liste d'entiers

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

Oracle : Égalité sur les entiers ✔

Calculer l'itinéraire le plus rapide entre deux villes

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

Oracle : Égalité des chemins ? ✘

Problème du sac à dos (résolu avec une heuristique)

Oracle : Résultat raisonnablement éloigné du résultat optimal ? ✘

Problème de l'oracle

Trouver le minimum d'une liste d'entiers

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

Oracle : Égalité sur les entiers ✔

Calculer l'itinéraire le plus rapide entre deux villes

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

Oracle : Égalité des chemins ? ✘

Trajet de 4h17 (quel que soit l'itinéraire choisi) ✔

Problème du sac à dos (résolu avec une heuristique)

Oracle : Résultat raisonnablement éloigné du résultat optimal ? ✘

Résultat = résultat optimal + 5% ✔

Problème de l'oracle

Oracle : En général, résultat attendu = ensemble de conditions si plusieurs résultats 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

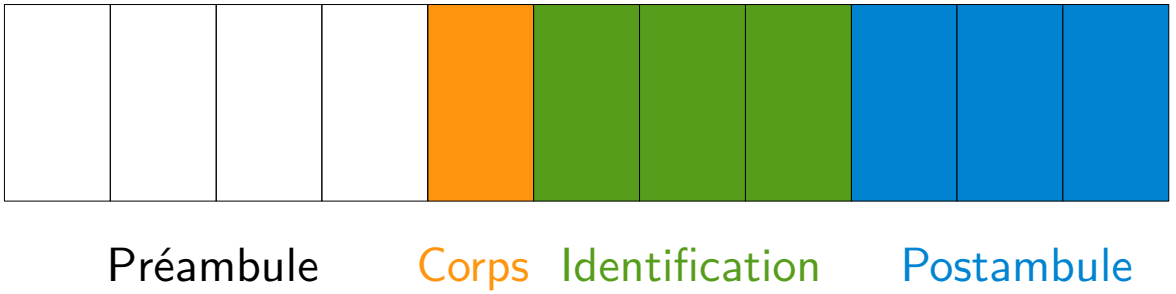
Faux négatifs (false-pass) : fait réussir un programme incorrect

Validité indispensable, complétude impossible en pratique

↳ Toujours s'assurer de la validité des tests

Scénario de test (unitaire)

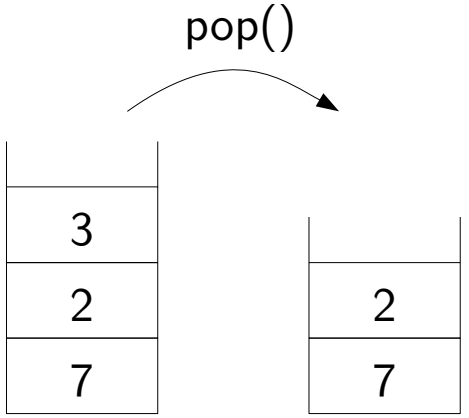
- Préambule : Suite d'actions amenant le programme dans l'état nécessaire pour exécuter le cas de test
- Corps : Exécution des fonctions du cas de test
- Identification : Opérations d'observation rendant l'oracle possible
- Postambule : Suite d'actions permettant de revenir à un état initial



Scénario de test (unitaire)

Pop (supprimer le sommet d'une pile)

Cas de test :



Scénario du test :

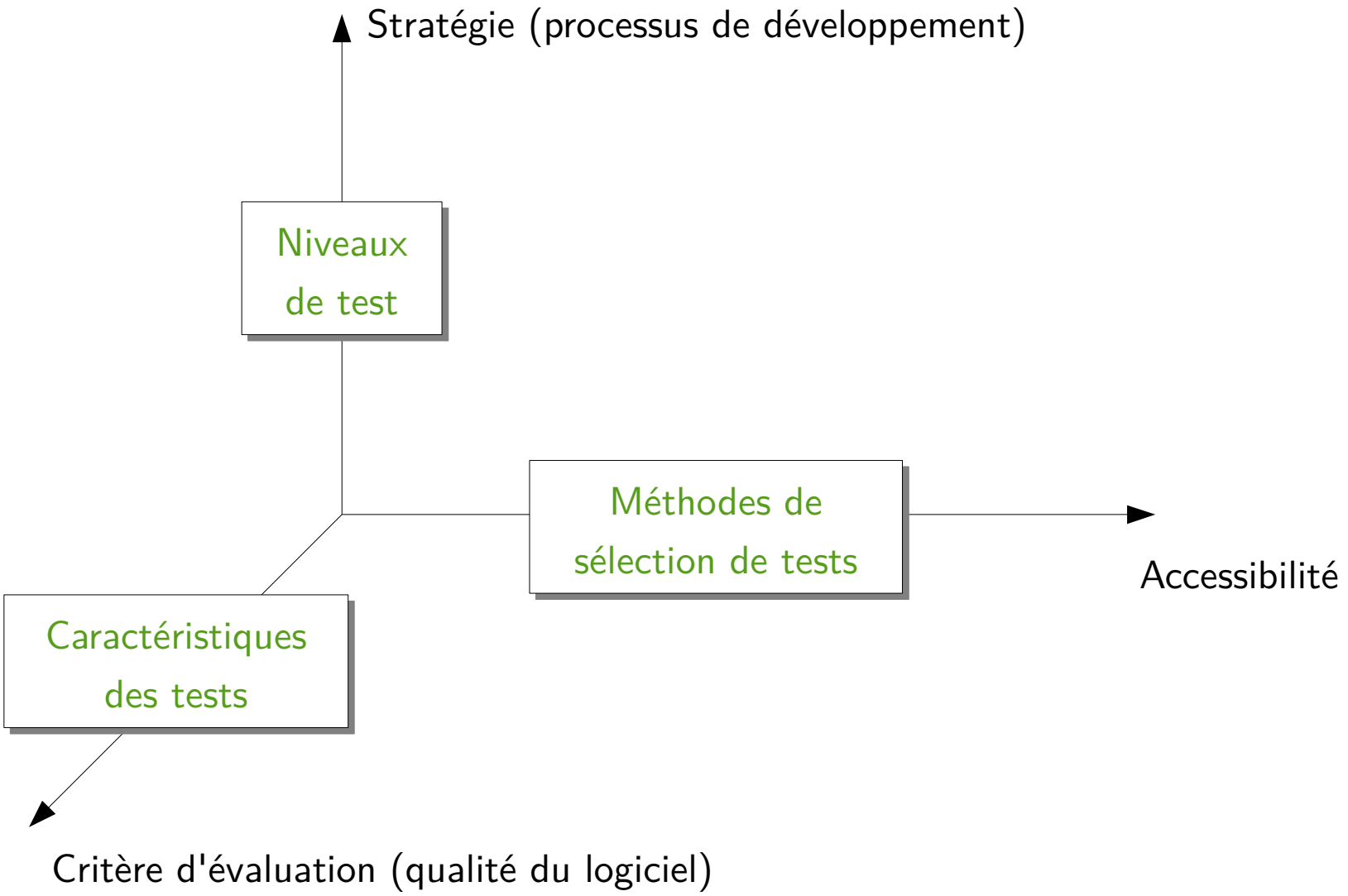
Préambule	Pile p = new Pile()
	p.push(7)
	p.push(2)
	p.push(3)
Corps	p.pop()
Identification	p.top() = 2
	p.pop()
	p.top() = 7
	p.pop()
	p.empty() = true
Postambule	free(p)

Attention

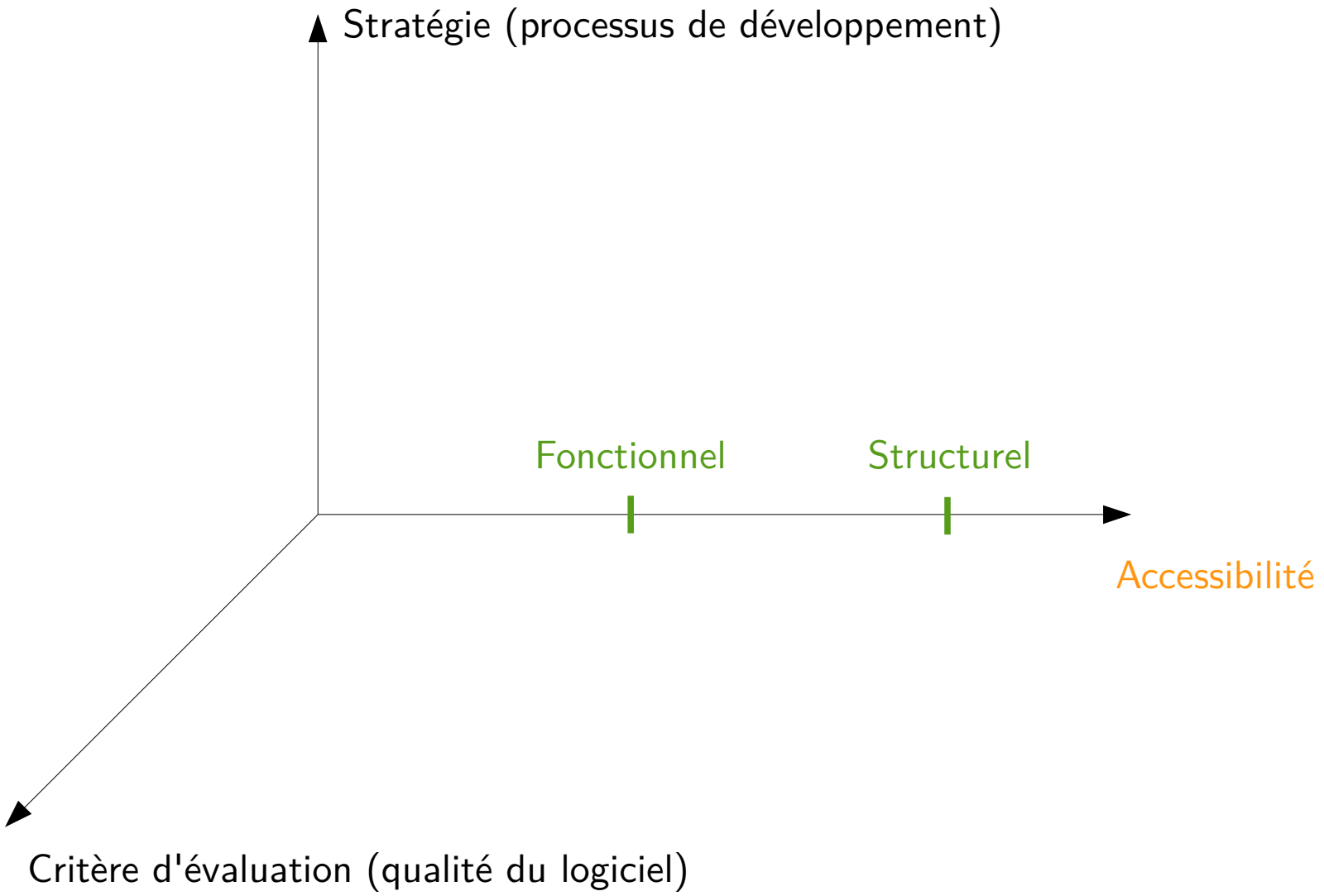
- Les tests doivent être exécutés dans des conditions et sur des données connues et contrôlées.
- Le résultat d'un test doit être observable.
- Il doit exister un moyen sûr de comparer le résultat observé au résultat attendu afin de déterminer le succès ou l'échec du test.
- Les cas de test ne doivent pas être redondants : plusieurs cas ne doivent pas cibler la même faute
- Un cas de test correspond à un comportement cible unique : pas de choix pendant l'exécution du test

Classification des types de test

Types de tests

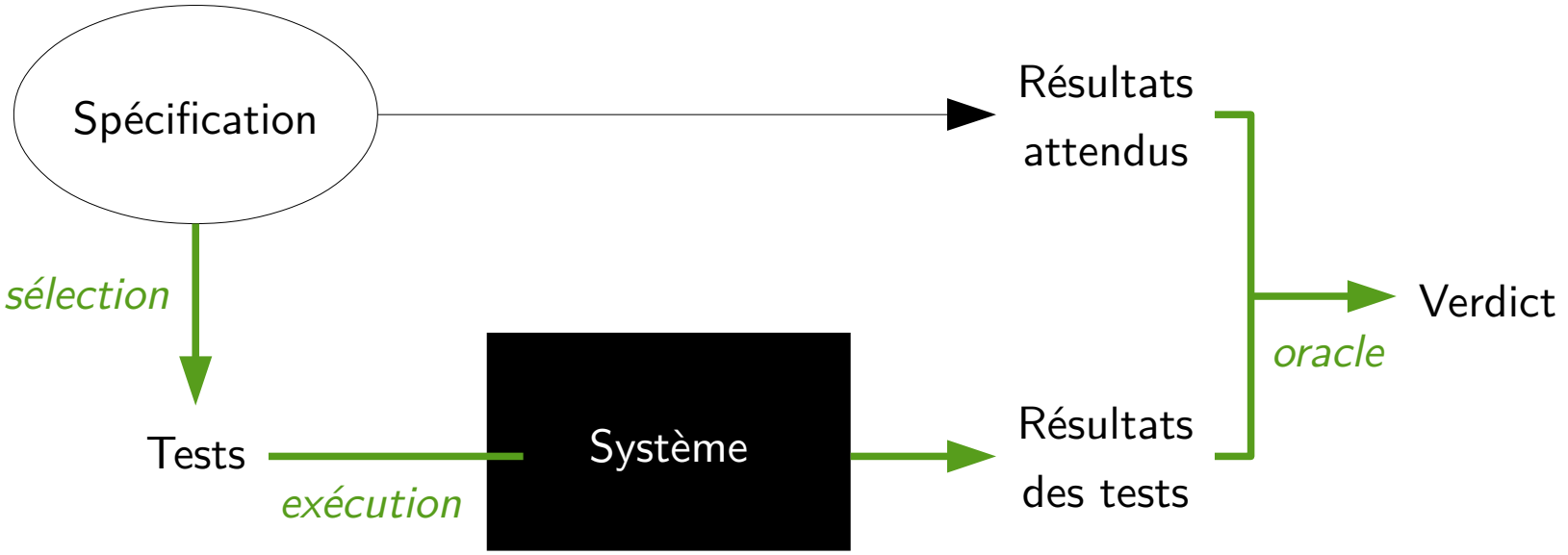


Types de tests



Test fonctionnel (ou test boîte noire)

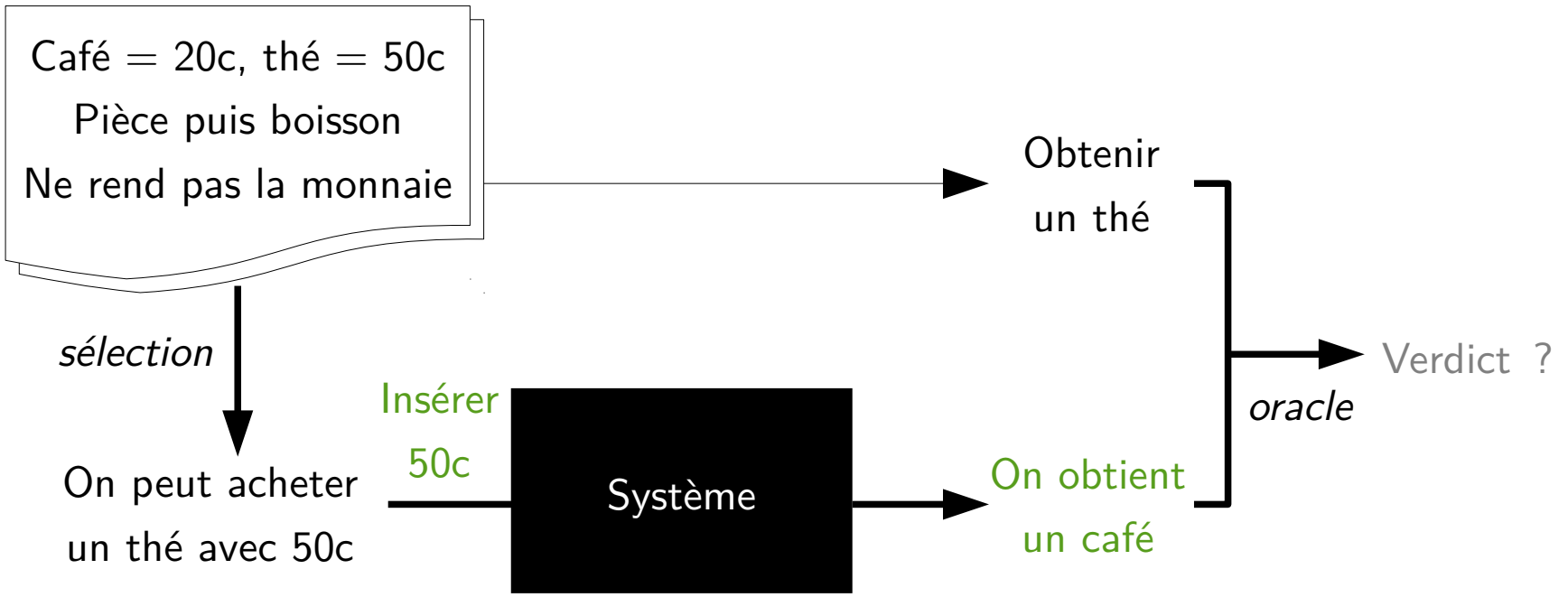
Sélection des tests à partir d'une spécification du système (formelle ou informelle), sans connaissance de l'implantation



Possibilité de construire les tests pendant la conception, avant la programmation

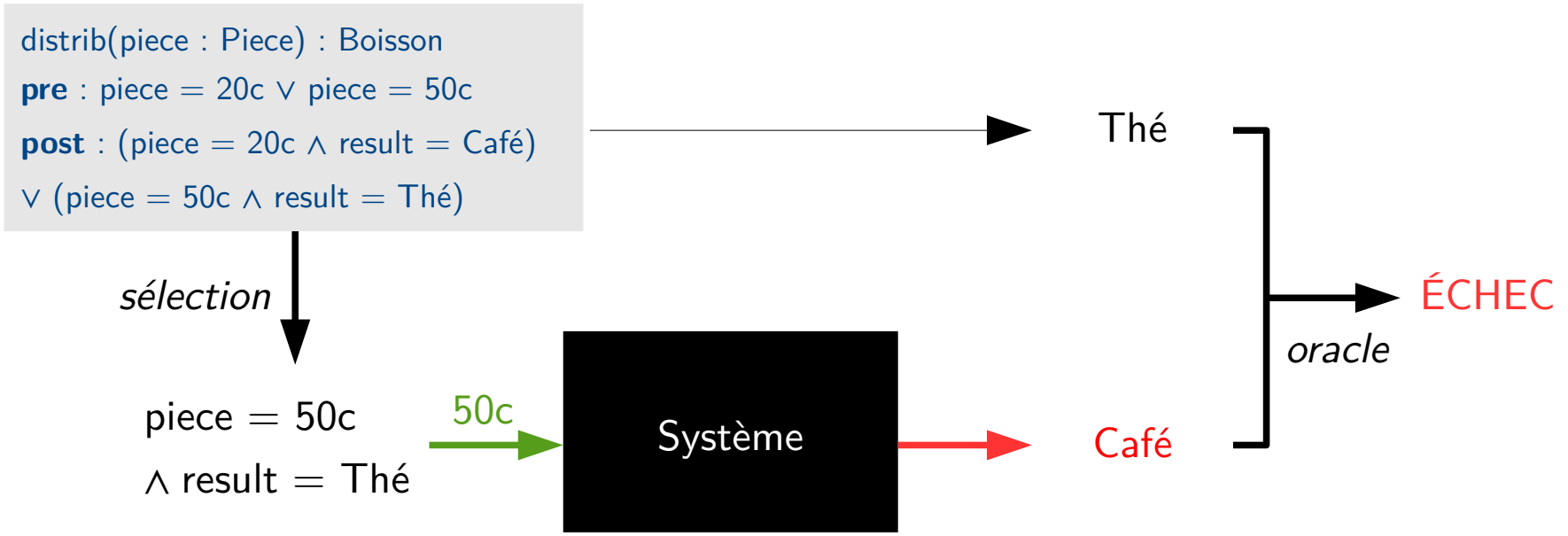
Test fonctionnel

Distributeur de café et thé



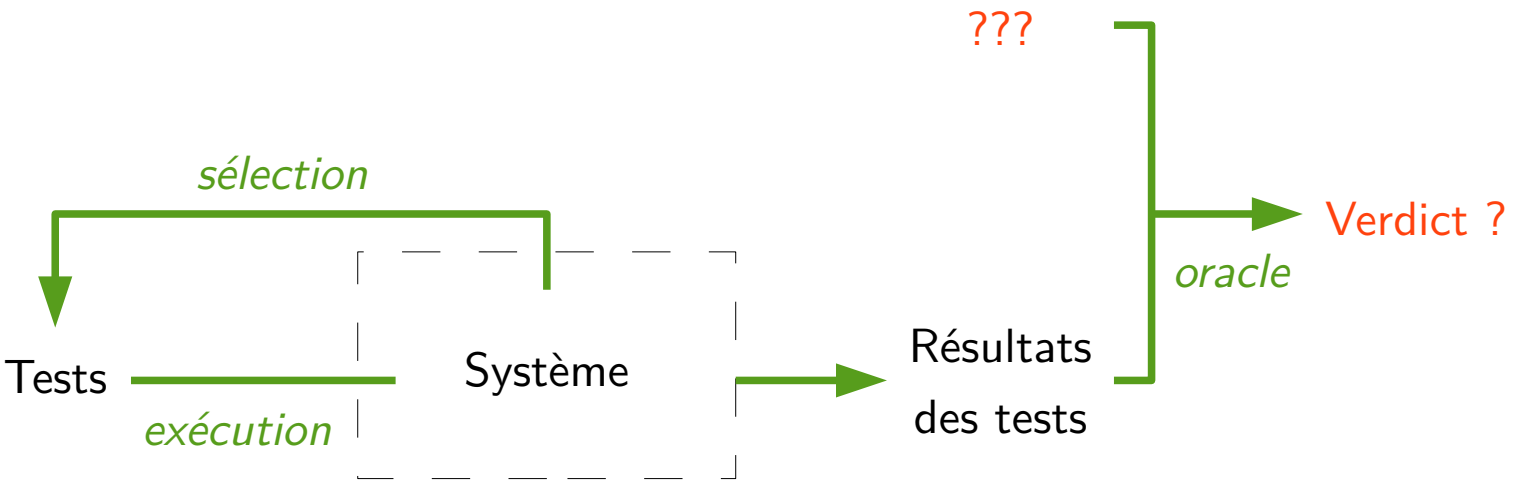
À partir d'une spécification formelle

Distributeur de café et thé



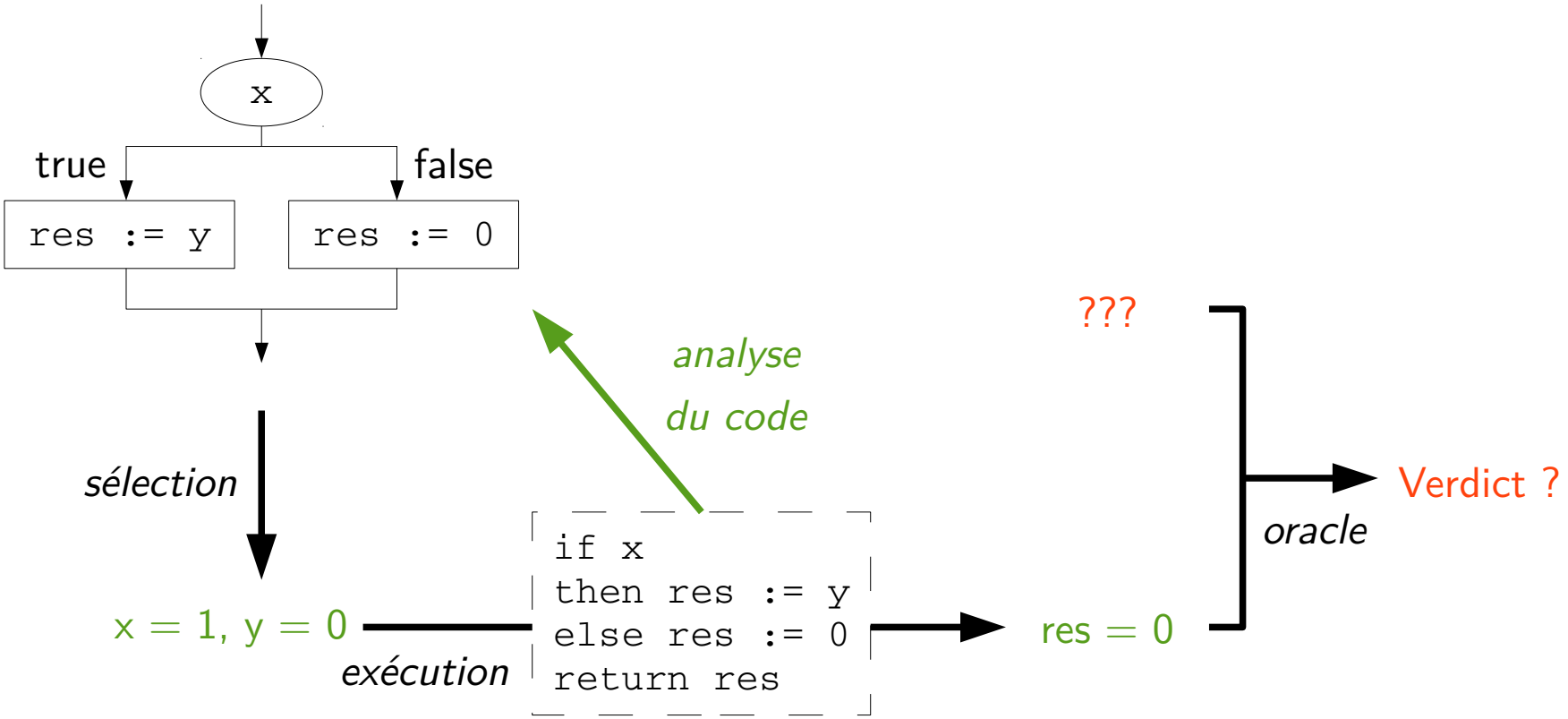
Test structurel (ou test boîte blanche)

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

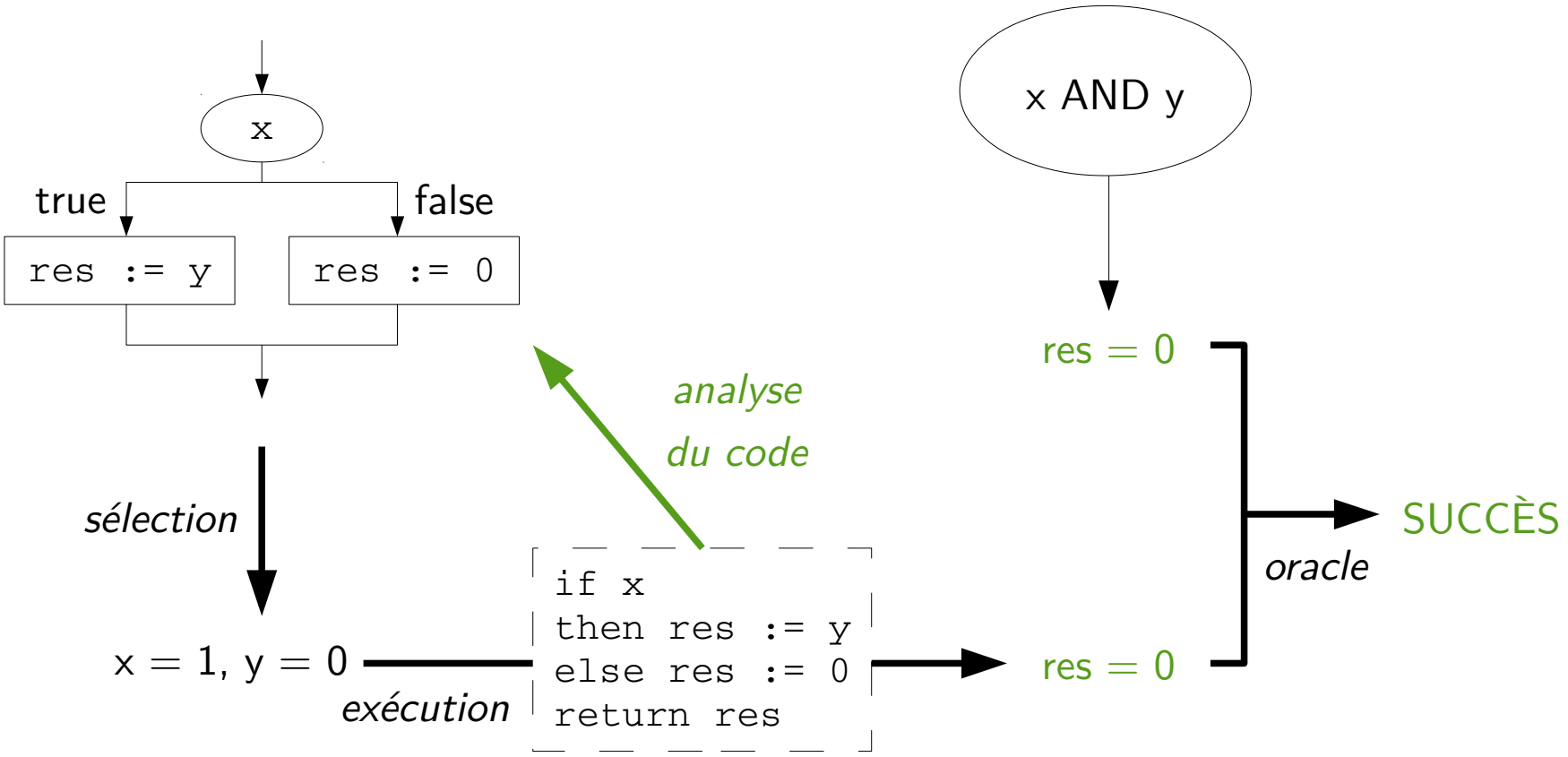


Construction des tests uniquement pour du code déjà écrit

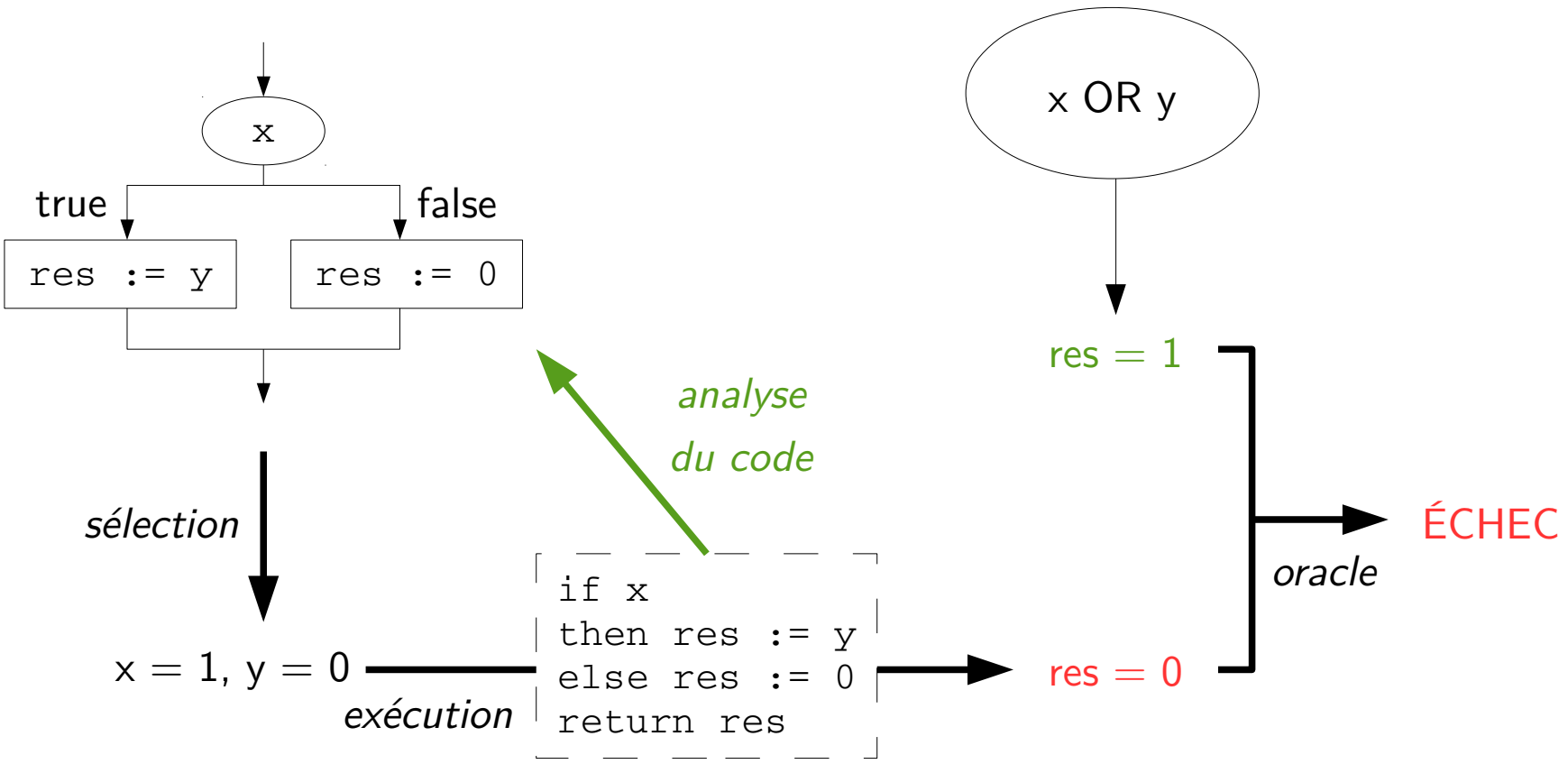
Test structurel



Test structurel

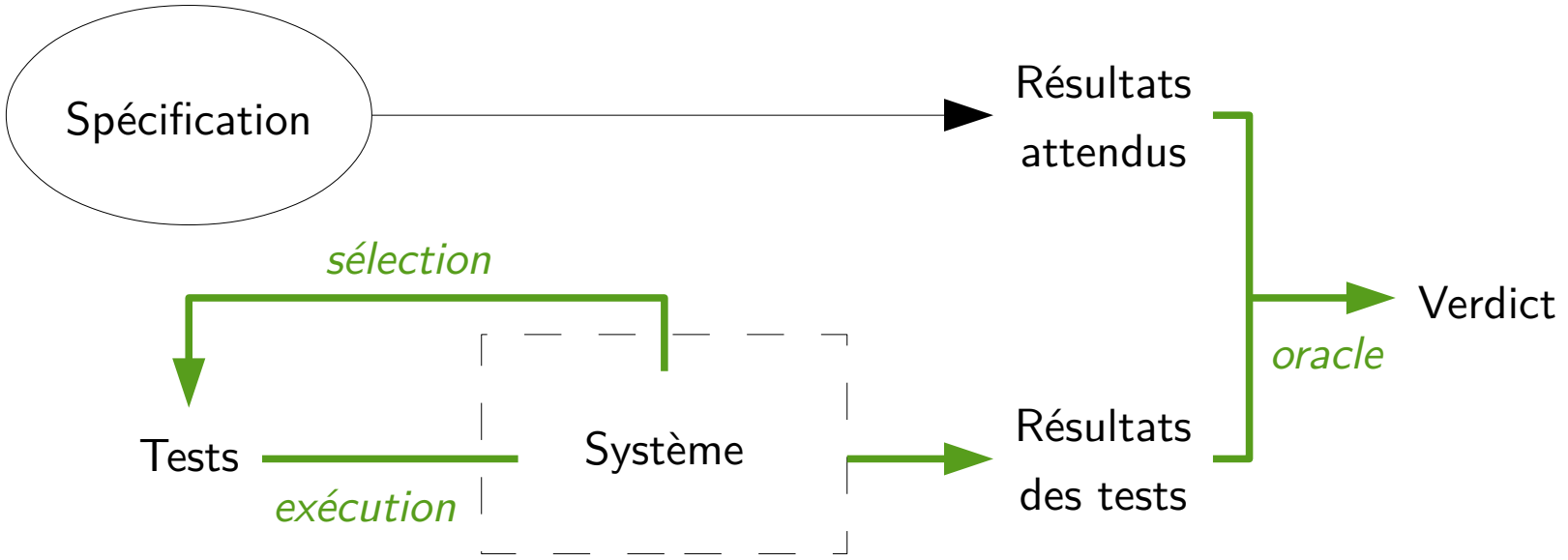


Test structurel



Test structurel

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



Construction des tests uniquement pour du code déjà écrit

Test fonctionnel vs. structurel

Complémentarité : détection de fautes différentes

- Test fonctionnel : détecte les oublis ou les erreurs par rapport à la spécification
- Test structurel : détecte les erreurs de programmation

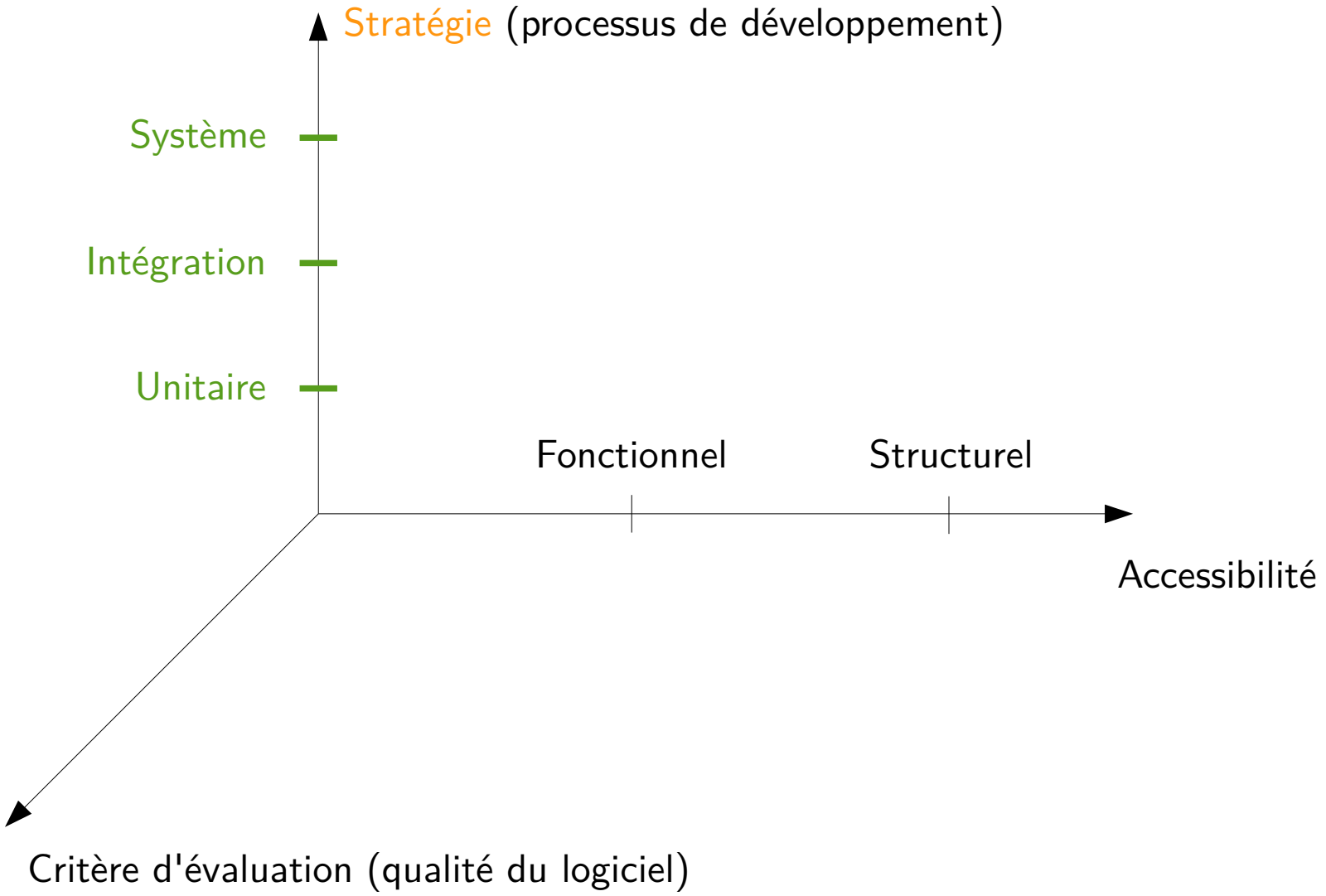
Addition d'entiers modulo 100 000

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

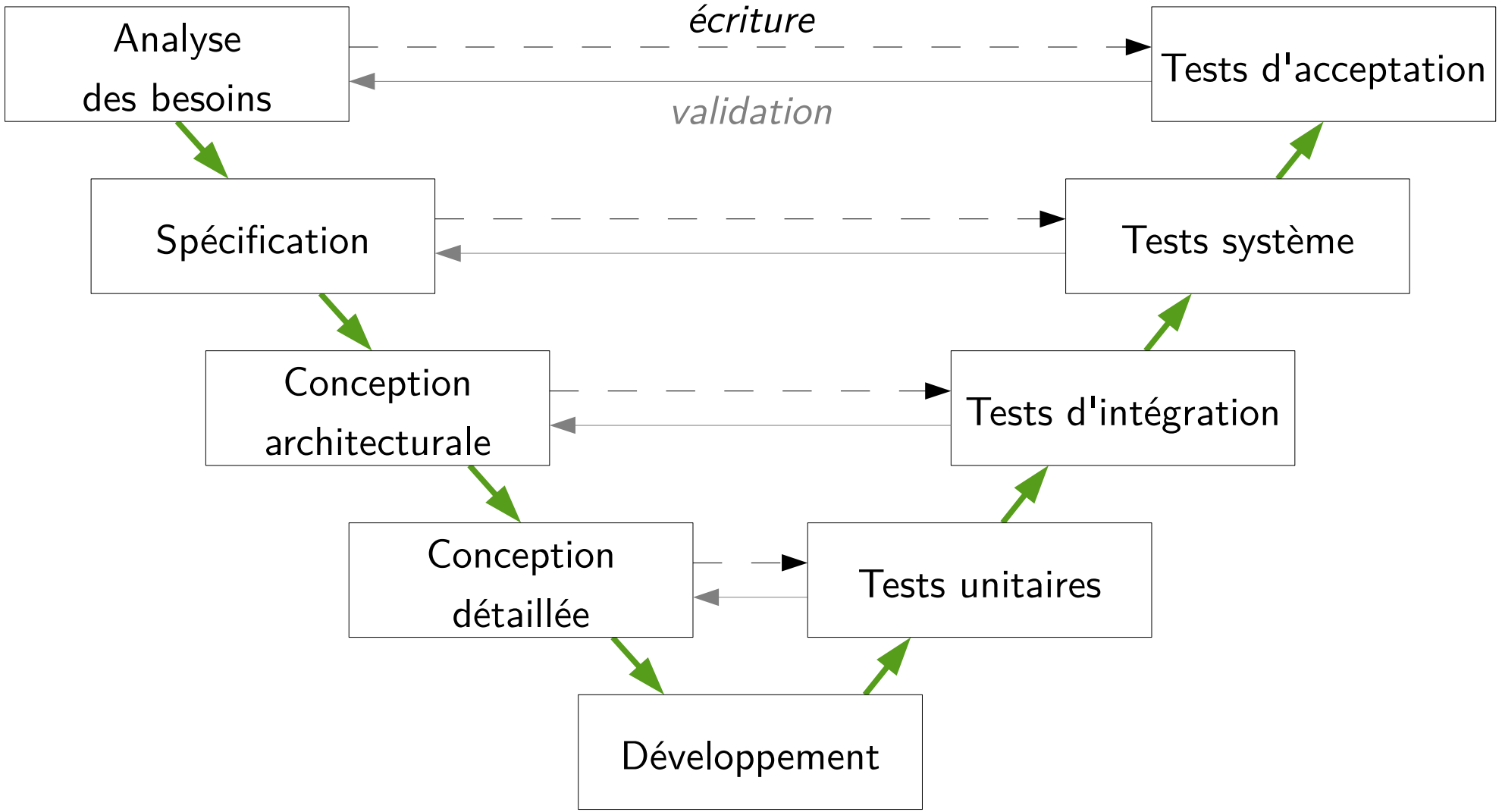
Test fonctionnel : détecte l'erreur par rapport à la spécification

Test structurel : détecte l'erreur pour les valeurs (600,500)

Types de tests



Processus de développement en V



Tests unitaires

Objectif : s'assurer que chaque unité de programme (méthode, classe, composant) remplit sa fonction, indépendamment du reste du système

- Tests effectués en isolant chaque partie du système
- Simulation nécessaire des parties dépendantes (bouchons)
- Tests ciblés sur les fonctionnalités associées à la méthode, à la classe, au composant
- (Possibilité d'utiliser des outils de mesure de la qualité des tests en terme de couverture de code)

Méthode

- Test incrémental : méthode, classe, package, composant

Tests d'intégration

Objectif : s'assurer que les composants interagissent correctement entre eux, par exemple :

- appels de méthodes extérieures (conditions sur les paramètres)
- accès à une base de données
- récupération des données saisies via l'interface utilisateur

Méthode

- Tests ciblés sur les interactions entre composants (à tous les niveaux)
- Tests construits à partir de :
  - l'architecture du système
  - la spécification des interfaces des composants

Tests système

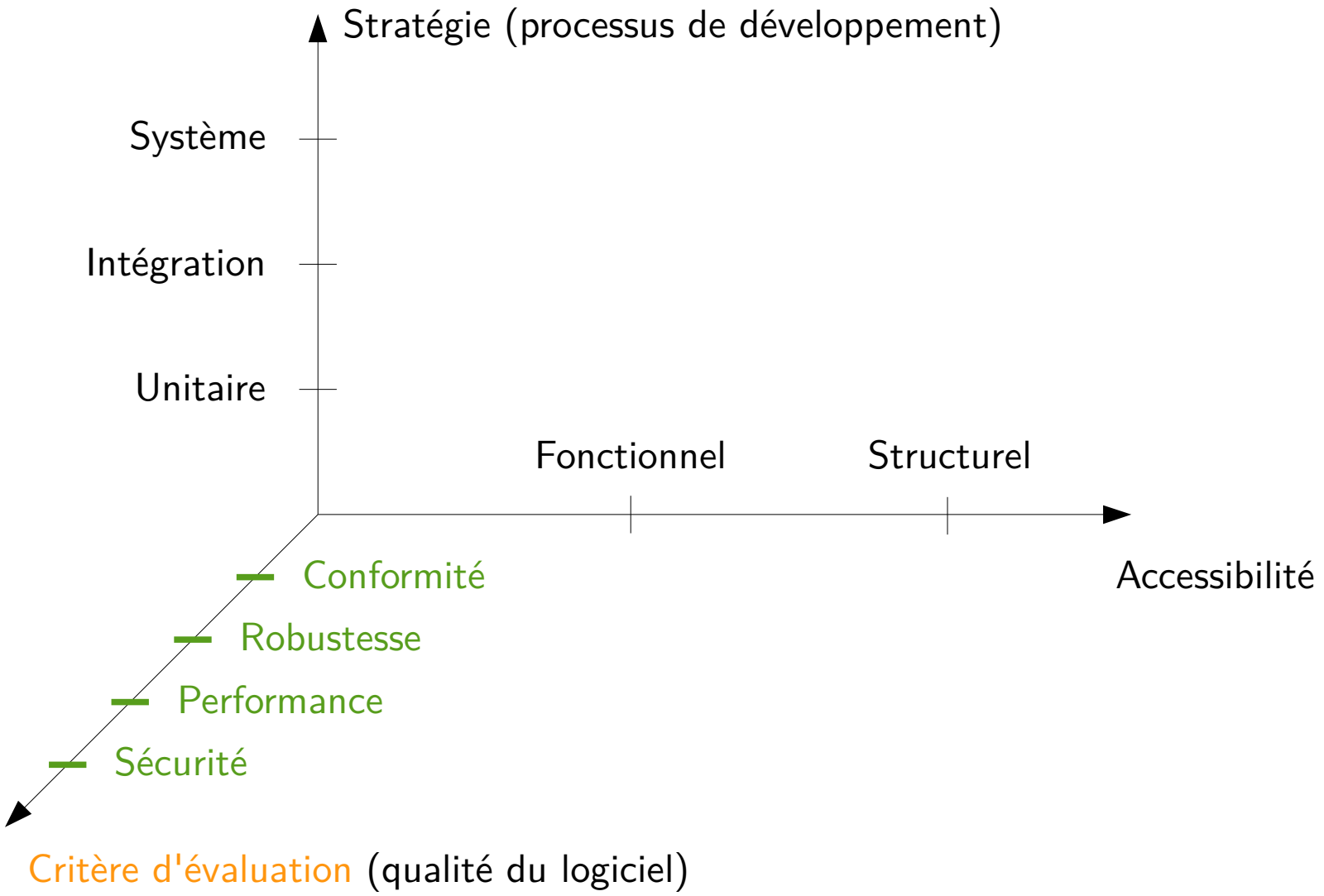
Objectif : s'assurer de la conformité du produit fini par rapport à son cahier des charges

- Tests effectués en boîte noire au travers de l'interface
- Tests ciblés sur les exigences décrites dans la spécification
- Construction possible des tests à partir des cas et des scénarios d'utilisation

Intérêt de la planification des tests système en phase d'analyse

- Validation du produit fini par rapport aux spécifications initiales
- Permet de s'assurer qu'on n'a pas dévié des spécifications

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

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...)

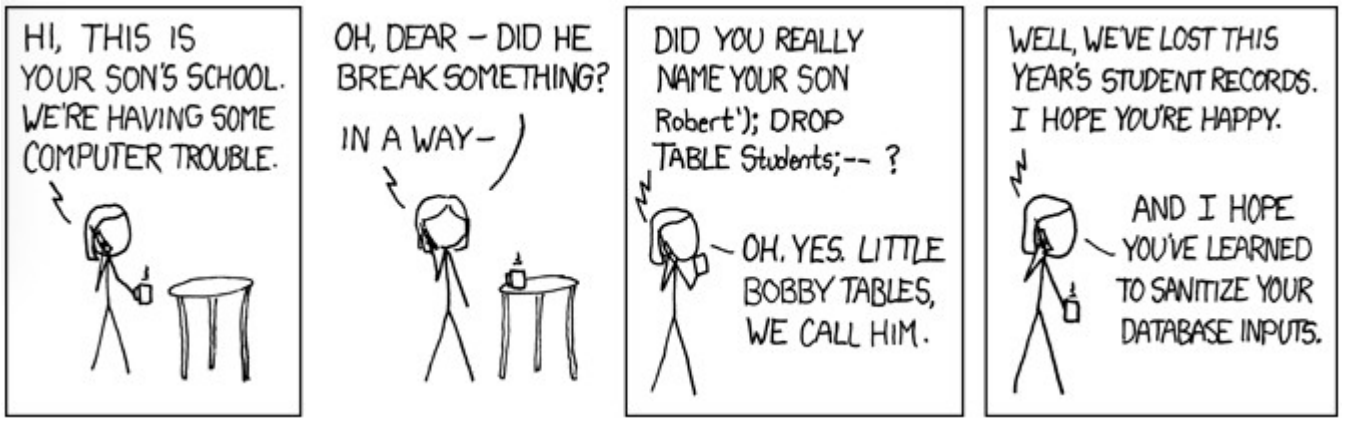
Service de paiement en ligne :

- 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



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

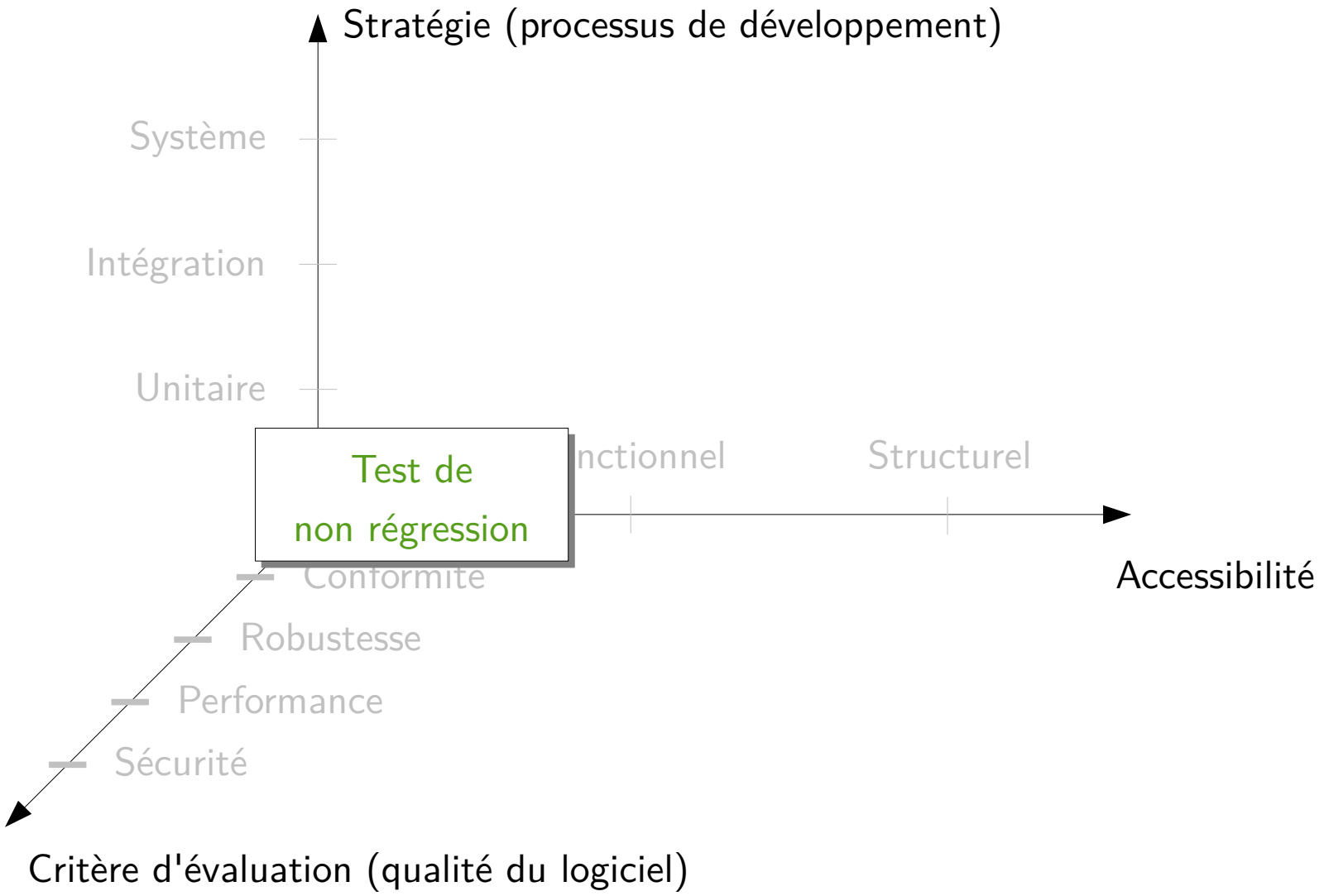
But : Assurer que le système garde des temps de réponse satisfaisants à différents niveaux de charge

Méthode : Simulation à différents niveaux de charge d'utilisateurs pour mesurer les temps de réponse du système, l'utilisation des ressources...

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

Méthodes informelles de sélection de tests

Problématique du test

Impossible d'exécuter un programme sur toutes ses entrées possibles

↳ Nécessité de sélectionner les tests

Objectif : détecter un maximum de défauts avec un minimum de tests

Ensemble de tests idéal

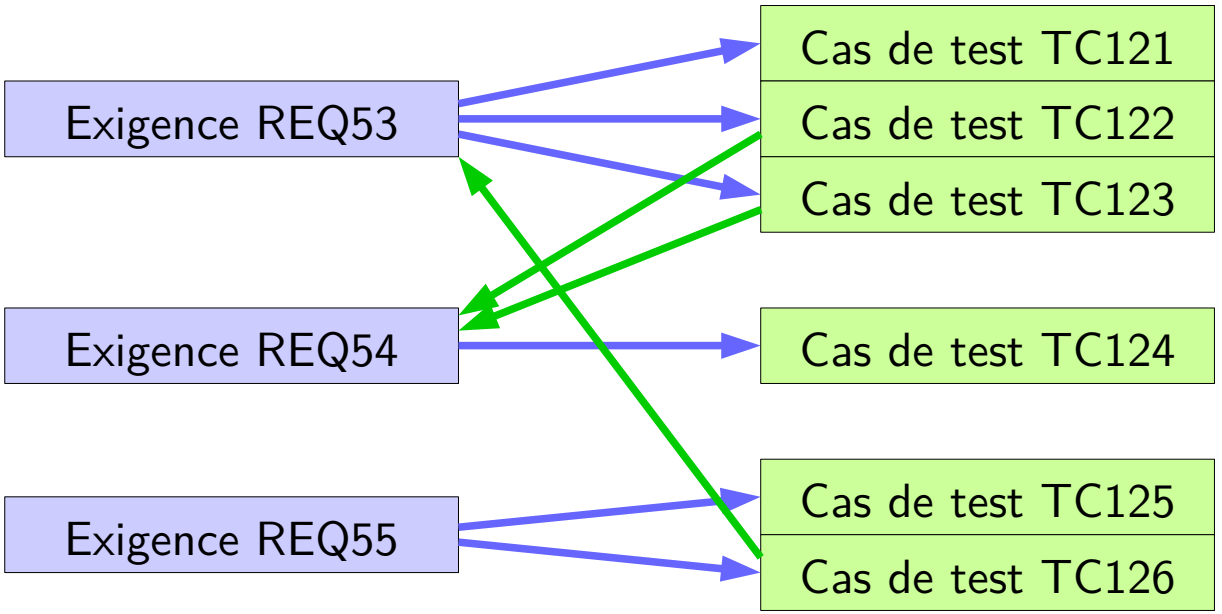
- praticable : qu'on peut exécuter en temps fini et raisonnable
- représentatif : susceptible de trouver un grand nombre de fautes

Couverture des exigences

Cahier des charges fonctionnel : liste numérotée d'exigences fonctionnelles et non fonctionnelles

Principe :

1. Découpage : construire au moins un test pour chaque exigence
2. Regroupement : déterminer exigences couvertes pour chaque test

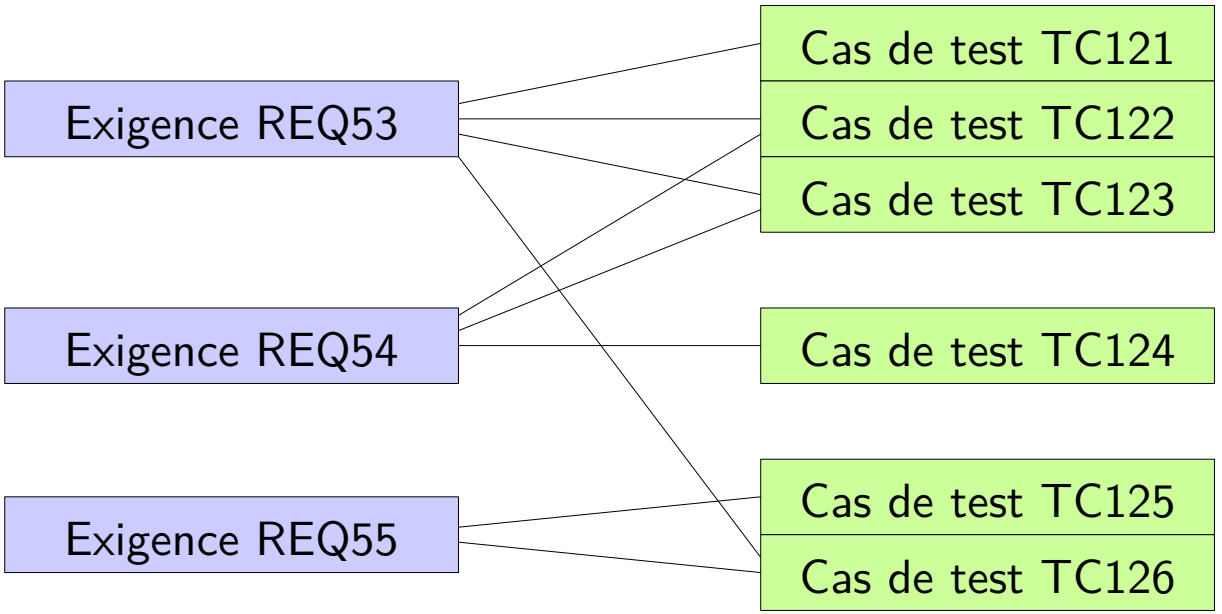


Couverture des exigences

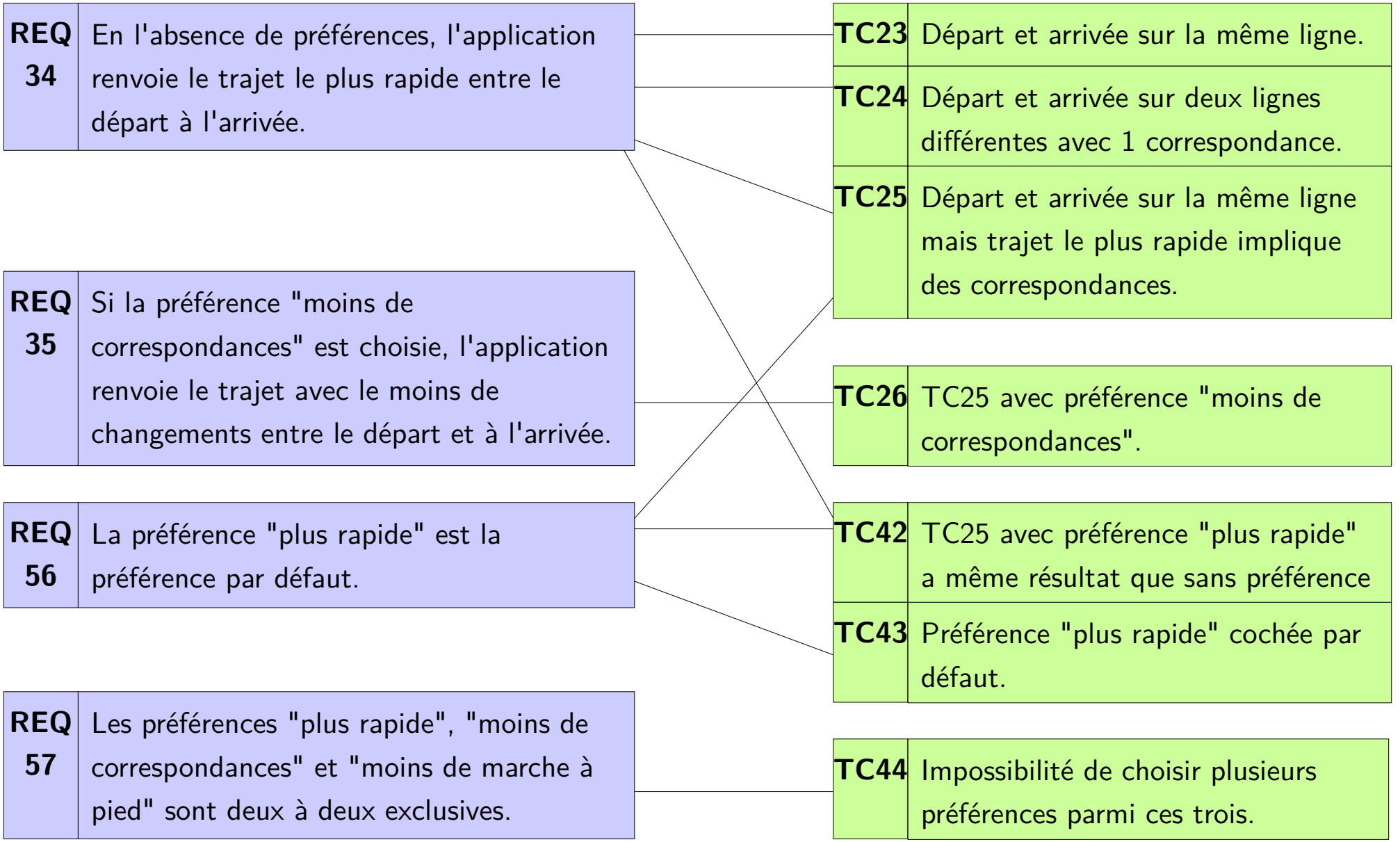
Intérêt :

- Couverture : tous les points du cahier des charges sont testés
- Traçabilité : tests à modifier si évolution d'une exigence

Difficulté restante : déterminer les cas de test pour une exigence



Couverture des exigences



Analyse partitionnelle

Principe : Définir des classes d'équivalence sur les domaines des entrées (ou des sorties) d'un système pour en déduire différents objectifs de test

Hypothèse : Pour chaque classe d'équivalence, toutes les entrées génèrent le même comportement

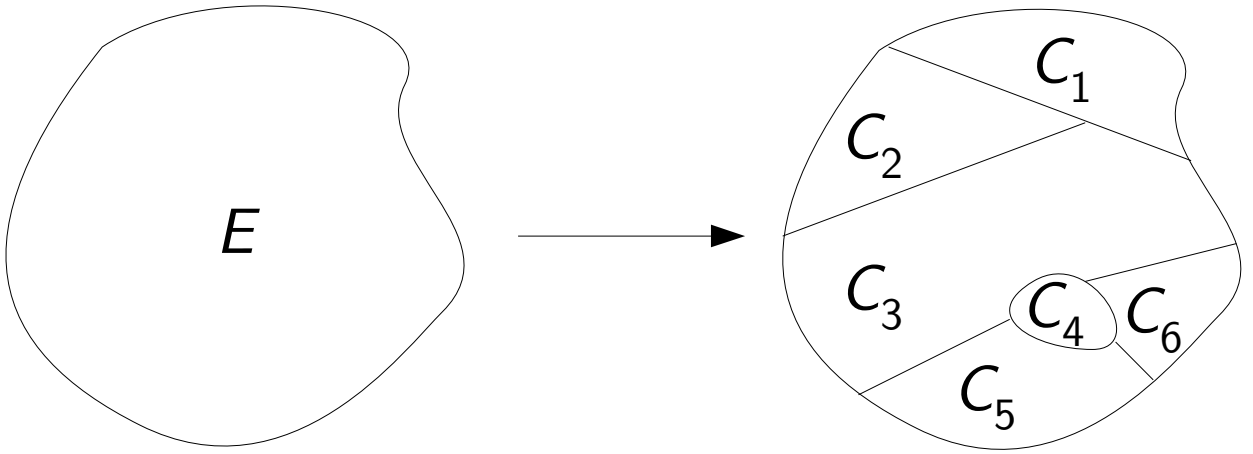
↳ Détection du même défaut

But : Tester chaque comportement de façon indépendante, en fonction du défaut cherché

Analyse partitionnelle

Construction à partir de l'ensemble E des entrées du programme d'une partition de E en un ensemble de classes C_i telles que :

- E est l'union des C_i
- les classes C_i sont deux à deux disjointes

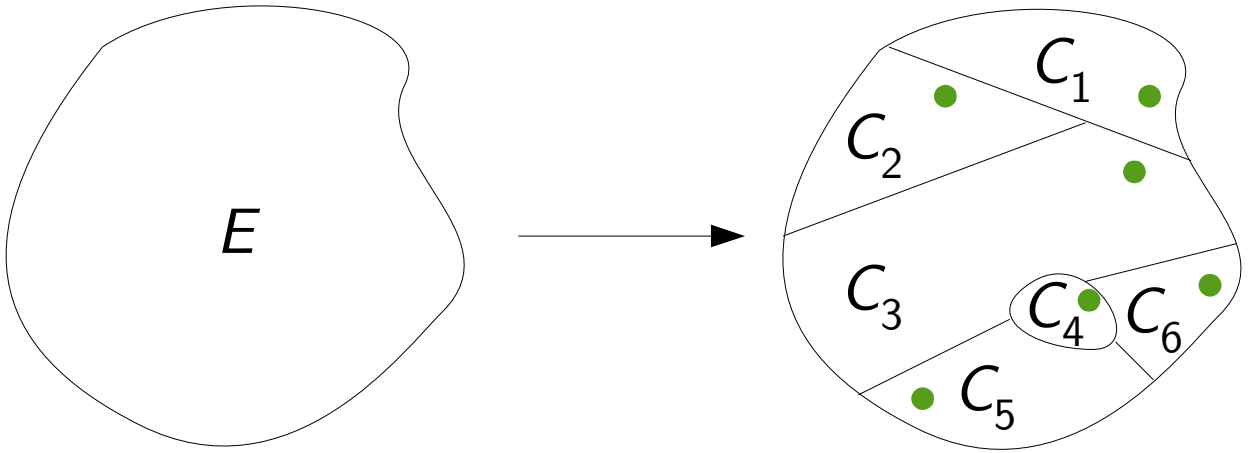


Analyse partitionnelle

Hypothèse d'uniformité : une seule valeur dans chaque C_i suffit à tester le comportement représenté par C_i

$$\exists\ d \in C_i,\ \text{correct}(\text{Prog},d) \Rightarrow \forall\ d \in C_i,\ \text{correct}(\text{Prog},d)$$

S'il existe une valeur d dans C_i telle que le programme est correct pour d, alors le programme est correct pour toutes les valeurs de C_i



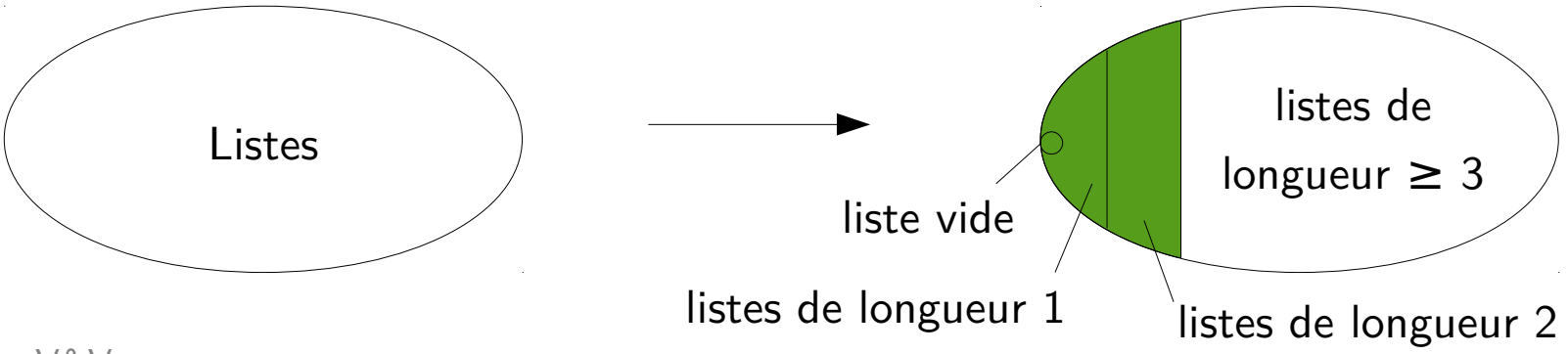
Analyse partitionnelle

Types de données munis d'une mesure : listes (longueur), arbre (hauteur ou nombre de nœuds), collection (nombre d'éléments)...

Hypothèse de régularité : il suffit de tester le programme pour les valeurs d dont la mesure $M(d)$ est inférieure à une constante N

$$\exists\ N,\ (\forall\ d,\ M(d) \leq N \Rightarrow \text{correct}(\text{Prog},d))$$
$$\Rightarrow\ \forall\ d,\ \text{correct}(\text{Prog},d))$$

Si le programme est correct pour toutes les valeurs de mesure inférieure à N, alors il est correct pour toutes les valeurs d'entrée



Analyse partitionnelle

Triangles

Le programme prend en entrée trois réels, interprétés comme étant les longueurs des côtés d'un triangle. Si ces longueurs forment un triangle, le programme renvoie la propriété du triangle correspondant (scalène, isocèle ou équilatéral) ainsi que la propriété de son plus grand angle (aigu, droit ou obtus).

Donner les classes d'équivalence associées à ce programme ainsi qu'un cas de test possibles pour chacune de ces classes.

Test aux limites

Principe : Construire des tests pour les valeurs limites du programme

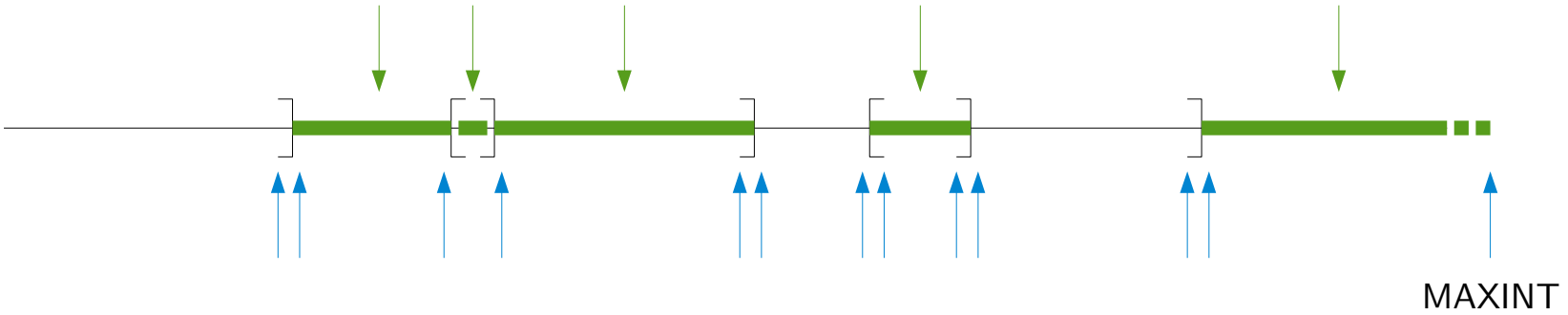
- Condition de boucle
- Valeurs très grandes
- Valeurs non valides

Utilisations :

- Conjointement à l'analyse partitionnelle : données choisies aux bornes des intervalles des classes d'équivalence (suppose une relation d'ordre sur les entrées)
- Pour le test de robustesse : données choisies en dehors des valeurs autorisées

Méthode générale

1. Pour chaque paramètre d'entrée, calculer les classes d'équivalence sur les domaines de valeurs de ce paramètre
2. S'il y a plusieurs paramètres, faire le produit cartésien des classes obtenues
3. Choisir des données de test



- Une valeur pour chaque classe obtenue
- Des valeurs aux limites

Analyse partitionnelle + test aux limites

Analyse partitionnelle :

- ✔ Réduction du nombre de cas de test
- ✗ Choix des classes délicat

Test aux limites :

- ✔ Heuristique solide pour le choix des données au sein des classes
- ✔ Production de tests de conformité et de tests de robustesse
- ✗ Relation d'ordre sur les entrées nécessaire
- ✗ Explosion combinatoire des données de test

Inconvénient majeur : Caractère intuitif ou subjectif de la partition en classe et de la notion de limite

↳ Difficulté pour caractériser la couverture des tests

Cas de nombreux paramètres

30 imprimantes

4 valeurs

+ une option complexe

1 entier

5 valeurs + sous-options

2 valeurs

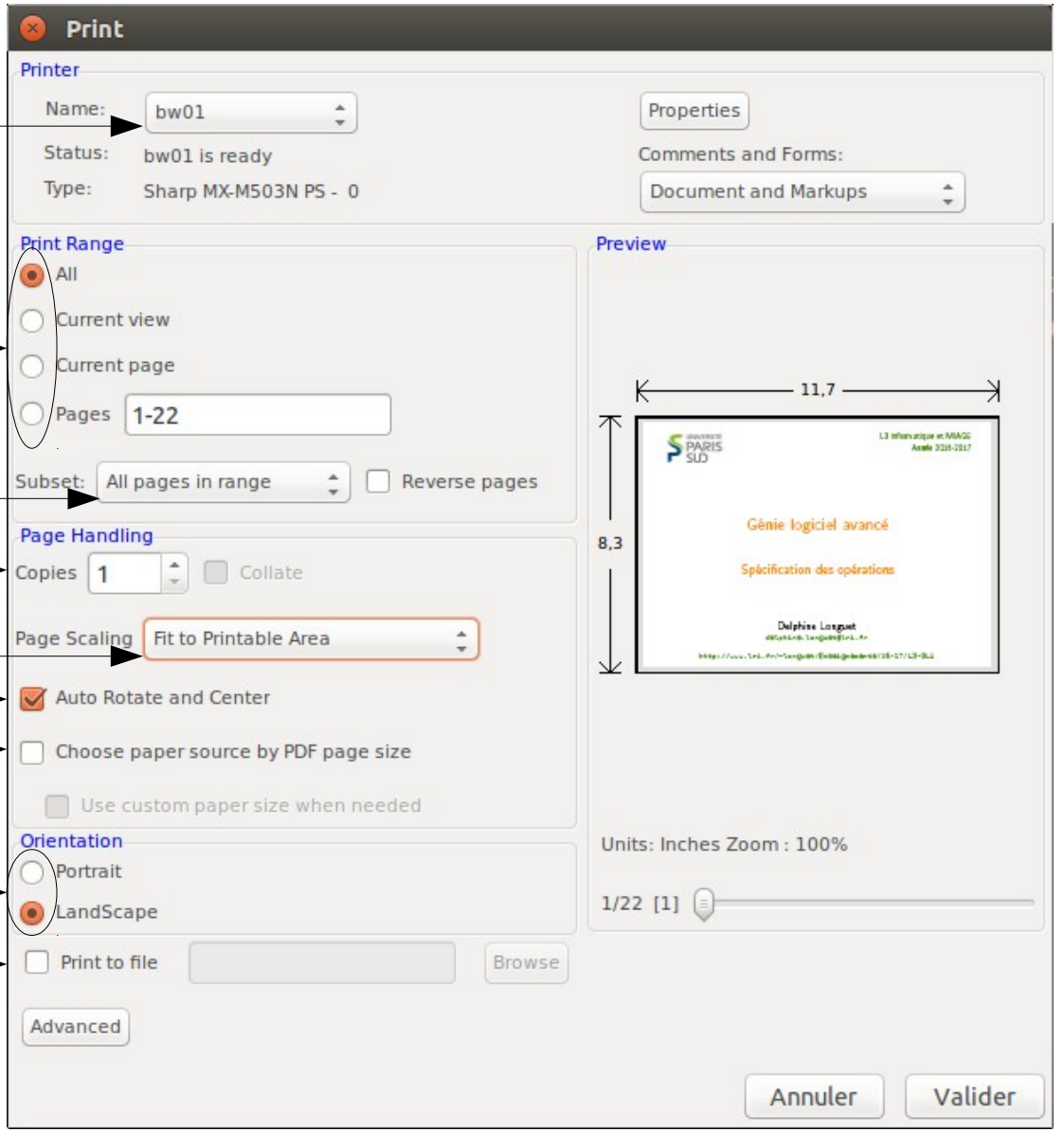
2 valeurs

2 valeurs

2 valeurs

+ choix du fichier

Plus les paramètres avancés...



Test pairwise

Constatation : Explosion combinatoire des valeurs d'entrées dans le cas de nombreux paramètres prenant des ensembles de valeurs finis

Solution : Tester un sous-ensemble des combinaisons de valeurs tel que chaque combinaison de n variables est testée

↳ Pairwise : $n = 2$

Idée sous-jacente : Majorité des fautes détectées par combinaisons de deux valeurs de variables

Complexité : Produit du cardinal des deux plus grands ensembles de valeurs

Test pairwise

Exemple : Trois variables x, y et z :

$x = 1$ ou 2 $\qquad$ $y =$ Q ou R ou S $\qquad$ $z = 5$ ou 6

Couvrir toutes les combinaisons (12) : 12 tests

Couvrir tous les couples (16) : 6 tests

x	y	z	Couples couverts
1	Q	5	(1,Q) (Q,5) (1,5)
1	R	6	(1,R) (R,6) (1,6)
1	S	5	(1,S) (S,5) (1,5)
2	Q	6	(2,Q) (Q,6) (2,6)
2	R	5	(2,R) (R,5) (2,5)
2	S	6	(2,S) (S,6) (2,6)

Test pairwise

Test pairwise

- ✔ Peut être utilisé avec l'analyse partitionnelle de plusieurs paramètres
- ✔ Génération outillée (voir `http://www.pairwise.org`)
- ✗ Combinaison aléatoire des valeurs
- ✗ Résultat attendu à fournir manuellement

Extension : Combinaisons de trois valeurs, quatre valeurs, etc.

Mais : Explosion du nombre de tests

Test pairwise

Test de configuration

On veut tester l'impression de fichiers depuis plusieurs applications sur des OS et via des réseaux différents. Donner les tests permettant de couvrir tous les couples de valeurs.

OS	Réseau	Imprimante	Application
Windows 7	IP	HP35	Word
Linux	Wifi	Canon900	Excel
Mac OS X	Bluetooth		PowerPoint
			AcrobatReader