

Vérification et Validation du Logiciel

Cours 1 — Introduction : place de la V&V, test et preuve

Stefania Dumbrava

ENSIIE & Télécom SudParis
stefania.dumbrava@ensiie.fr

Pourquoi la V&V ?

La V&V dans le cycle de vie

Test et preuve

Notions de base du test

Le processus de test

À l'issue de ce cours, vous saurez :

- ▶ situer la V&V dans le cycle de vie : cycle en V, méthodes agiles, intégration continue ;
- ▶ distinguer **validation** et **vérification**, **test** et **preuve** ;
- ▶ employer le vocabulaire du test :
erreur, défaut, défaillance, cas de test, oracle, verdict ;
- ▶ classer un test selon son niveau, son objectif et son mode de sélection ;
- ▶ écrire en OCaml un test unitaire et un test fondé sur les propriétés.

Pourquoi la V&V ?

Quand le logiciel défaille : quelques cas documentés

Système	Année	Défaut en cause et conséquence
Therac-25	1985--87	Situation de compétition (<i>race condition</i>) ; verrous matériels supprimés. Six surdosages massifs, plusieurs décès.
Ariane 5, vol 501	1996	Conversion flottant 64 bits → entier signé 16 bits dans un module repris d'Ariane 4, non revalidé pour la nouvelle trajectoire. Désintégration du lanceur ≈ 40 s après le début du vol.
Mars Climate Orbiter	1999	Défaut d'interface : impulsions produites en livre-force-seconde, attendues en newton-seconde. Perte de la sonde.
Knight Capital	2012	Déploiement incomplet ; un indicateur réutilisé réactive du code obsolète. ≈ 440 M\$ de pertes en 45 minutes.
Heartbleed (OpenSSL)	2014	Longueur annoncée dans une requête, jamais vérifiée : fuite de jusqu'à 64 Ko de mémoire par requête.
CrowdStrike Falcon	2024	Modèle de contenu à 21 champs, code n'en fournissant que 20 : lecture hors limites, $\approx 8,5$ millions de postes Windows bloqués.

Leveson & Turner (1993) ; rapport de la commission d'enquête Ariane 501 (1996) ; NASA MCO Mishap Board (1999) ; SEC (2013) ; CVE-2014-0160 ; CrowdStrike, *Root Cause Analysis* (août 2024).

Le coût de la V&V... et celui de son absence

Coût de l'insuffisance des tests

Le NIST estime à **59,5 milliards de dollars par an** le coût, pour l'économie américaine, d'une infrastructure de test inadéquate. 22,2 milliards seraient évitables.

Part du test dans le développement

Ordre de grandeur classique : **environ la moitié** du temps et du coût d'un projet (Myers) ; davantage pour les logiciels critiques certifiés (avionique, DO-178C).

À retenir

La V&V est une part majeure de l'effort d'ingénierie et la négliger coûte bien plus cher.

G. Tassey, *The Economic Impacts of Inadequate Infrastructure for Software Testing*, NIST Planning Report 02-3, 2002

G. J. Myers, *The Art of Software Testing*, Wiley, 1979.

Validation ou vérification ?

Validation (validation externe)

« *Construisons-nous le bon produit ?* »

- ▶ Le système répond-il au besoin du client ?
- ▶ Ses performances sont-elles suffisantes ?

Référence : **l'utilisateur et son besoin.**

Vérification (validation interne)

« *Construisons-nous le produit correctement ?* »

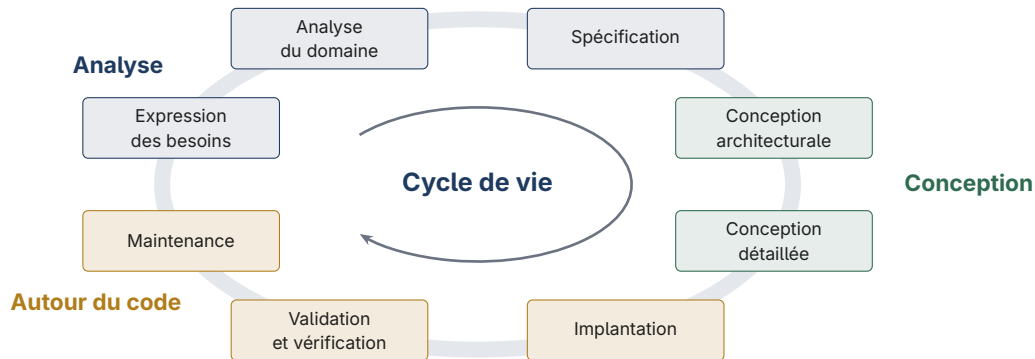
- ▶ Le code est-il conforme à la spécification ?
- ▶ Chaque étape respecte-t-elle la précédente ?

Référence : **un document**
(cahier des charges, spécification).

Un logiciel parfaitement *vérifié* vis-à-vis d'une spécification erronée n'est pas *validé*.

La V&V dans le cycle de vie

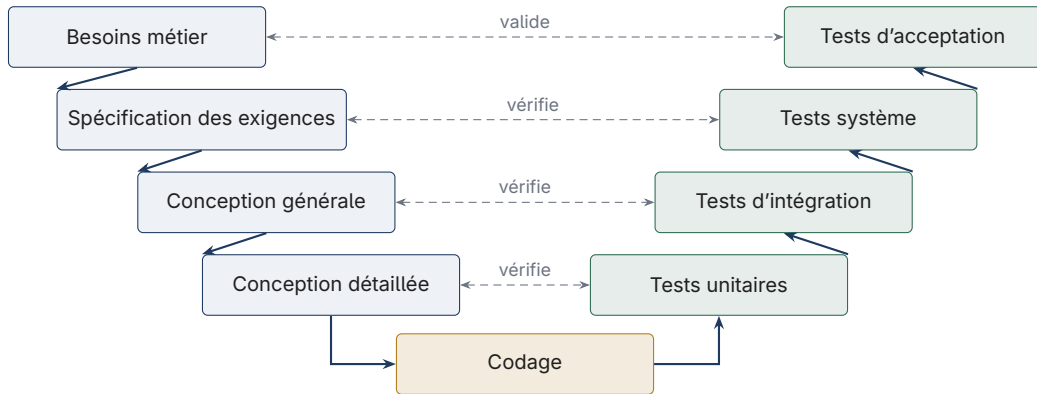
Une vision naïve : la V&V comme phase terminale



Dans ce schéma, la V&V n'intervient qu'*après* le code.

Les défauts de spécification ou de conception sont découverts trop tard.

Le cycle en V : à chaque étape de conception, son niveau de test



Les tests de chaque niveau sont **conçus en même temps** que l'étape de conception correspondante, et exécutés en remontant le V.

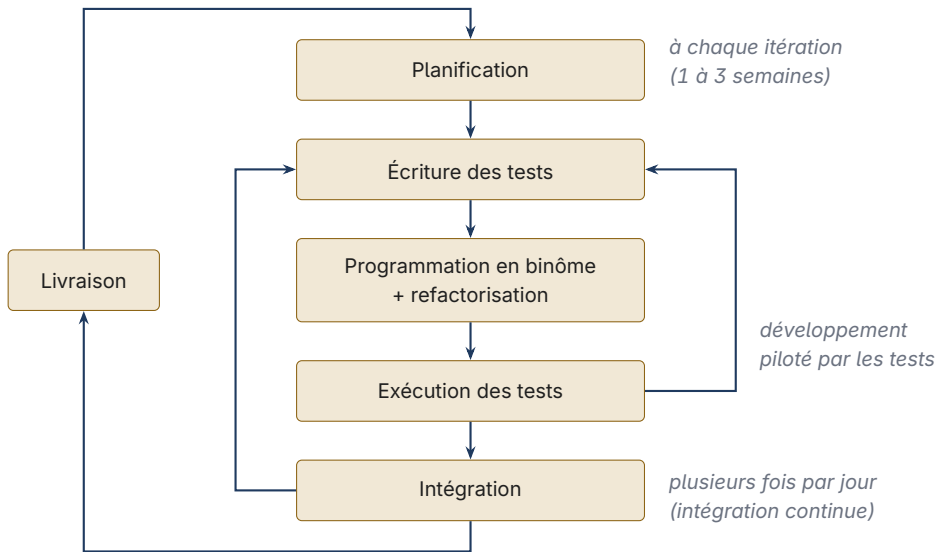
Méthodes agiles : cycles **itératifs et incrémentaux**, adaptés aux besoins évolutifs d'un client impliqué tout au long du projet.

Extreme Programming (XP) — K. Beck

- ▶ itérations courtes (**une à trois semaines**) ;
- ▶ **développement piloté par les tests** (TDD) :
le test est écrit *avant* le code qu'il exerce ;
- ▶ tests automatisés pour chaque fonctionnalité, rejoués en permanence ;
- ▶ programmation en binôme, refactorisation continue.

Les tests automatisés ne garantissent pas la correction mais...
ils rendent le changement *sûr* : toute régression détectée est signalée immédiatement.

Le cycle XP



Intégration continue (CI)

Pousser plus loin l'**automatisation des vérifications** :

- ▶ un gestionnaire de versions partagé (aujourd'hui presque toujours git) ;
- ▶ chaque développeur intègre ses changements dans la branche principale **au moins une fois par jour** ;
- ▶ chaque intégration déclenche une construction complète (*build*) et l'exécution automatique de la suite de tests ;
- ▶ une construction cassée est réparée en priorité : *la branche principale doit toujours passer 100 % des tests.*



Test et preuve

- ▶ **Validation** : essentiellement par le **test** (et la revue avec le client).
- ▶ **Vérification** : par le **test** et par les **méthodes formelles**.

Dans ce cours :
le test, puis la **preuve formelle de programmes**.

Autres méthodes formelles du cursus

- ▶ *model checking*
(Concepts et Model Checking, S4)
- ▶ analyse statique
(PROG2, S5)
- ▶ développement par raffinement,
dès la spécification
(MFDLS, S4)

Le but du test : trouver des erreurs

Le test est une méthode **dynamique** : il faut *exécuter* le programme.

G. J. Myers, *The Art of Software Testing* (1979)

« Tester, c'est exécuter un programme dans l'intention d'y trouver des erreurs. »

E. W. Dijkstra, *Notes on Structured Programming* (1970)

"Program testing can be used to show the presence of bugs, but never to show their absence!"

Conséquence :

un test qui passe n'apporte qu'un *degré de confiance*, jamais une garantie.

Deux usages du test

Construire la qualité

- ▶ pendant la conception et le codage ;
- ▶ en partie par les développeurs (tests unitaires) ;
- ▶ but : trouver *le plus tôt possible* le plus de défauts possible.

Un test **réussi** est un test qui **révèle un défaut**.

Démontrer la qualité à un tiers

- ▶ une fois le produit terminé ;
- ▶ par une équipe dédiée, distincte des développeurs ;
- ▶ but : convaincre (client, hiérarchie, organisme de certification).

Un test **réussi** est un test qui **passe** ; la suite de tests doit être *représentative* (et, pour les systèmes critiques, elle est elle-même audité).

Le test exhaustif est presque toujours impossible

Considérons une fonction de deux entiers 32 bits :

```
val f : int32 -> int32 -> int32
```

- ▶ $2^{32} \times 2^{32} = 2^{64} \approx 1,8 \times 10^{19}$ entrées possibles ;
- ▶ à 10^9 tests par seconde : **environ 585 ans**.

Et c'est sans compter l'état interne, l'ordre des appels, l'environnement, la concurrence...

Conséquence

Il faut **sélectionner** un petit nombre de tests pertinents : c'est tout l'enjeu des critères de sélection (test fonctionnel, test structurel) étudiés par la suite.

- ▶ **Testabilité** : le système sous test doit être déterministe (ou son non-déterminisme maîtrisé) et initialisable dans un état connu.
- ▶ **Hypothèses de sélection** : on suppose le comportement *uniforme* sur certains sous-domaines (un représentant suffit) ou *régulier* (tester jusqu'à une certaine taille suffit). Réaliste, mais pas toujours vrai.
- ▶ **Portée du résultat** : on n'obtient qu'un degré de confiance ; même le critère de couverture le plus exigeant n'exclut pas une erreur.
- ▶ **Passage à l'échelle** : le test manuel suit mal la taille du code et le niveau d'exigence
⇒ il faut **automatiser**.

Le but de la preuve formelle

La preuve est une méthode **statique** : on raisonne sur le code sans l'exécuter.

Objectif

Étant donnée une spécification formelle S d'un programme P , établir par une **preuve mathématique** que toute exécution de P satisfait S : P est correct par rapport à S .

Il faut :

- ▶ une **spécification formelle** de P ;
- ▶ une **méthode de preuve** (p. ex. logique de Hoare);
- ▶ des **règles de preuve** : un calcul.

Triplet de Hoare

$$\{x \geq 0\} \quad y := x + 1 \quad \{y > 0\}$$

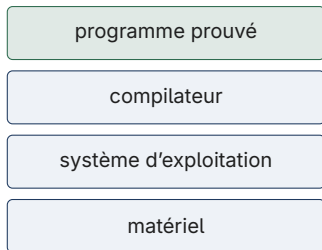
Si la précondition est vraie avant l'exécution et que celle-ci termine, la postcondition est vraie après (*correction partielle*).

- ▶ **Pas de correction absolue** : la correction est *relative à une spécification*, elle-même possiblement fausse ou incomplète (problème de *validation*).
- ▶ **Coût** : spécifier formellement et prouver est difficile. On cible surtout des propriétés de sûreté (accès dans les bornes, pas de pointeur nul, pas d'exception non rattrapée) et les parties critiques.
- ▶ **Propriétés difficiles à spécifier** : comportement de l'utilisateur et de l'environnement physique ; ressources (temps, mémoire), spécifiables mais plus difficilement.
- ▶ **Limite théorique** (théorème de Rice, 1953) : toute propriété sémantique non triviale des programmes est indécidable. Aucun outil ne prouve tout automatiquement : annotations, interaction ou approximations sont nécessaires.

Un programme prouvé peut-il planter ?

Oui : la preuve porte sur un *modèle*.

Tout ce qui est en dessous doit aussi être digne de confiance.



- **Compilateur** : **CompCert**, compilateur C vérifié en Coq/Rocq (X. Leroy, Inria). Le test aléatoire Csmith a trouvé 79 bogues dans GCC et 202 dans LLVM, mais *aucun* dans la partie vérifiée de CompCert.
- **Noyau** : **seL4**, micro-noyau dont le code C est prouvé conforme à sa spécification (NICTA, 2009).
- **Matériel** : depuis le bogue de division du Pentium (FDIV, 1994), Intel vérifie formellement ses unités de calcul.

Test et preuve sont complémentaires

	Test	Preuve formelle
Nature	dynamique : exécute le programme	statique : raisonne sur le code
Couverture	un nombre fini d'exécutions	<i>toutes</i> les exécutions
Résultat	trouve des défauts ; confiance partielle	garantit l'absence de défauts <i>vis-à-vis de S</i>
Référence	une spécification, éventuellement informelle	une spécification formelle
Ce qu'il voit	le système réel : compilateur, OS, matériel, environnement	un modèle du système
Coût	faible au départ, croît avec le niveau d'exigence	élevé, expertise requise
Automatisation	très bonne (exécution) ; oracle et sélection difficiles	partielle (assistants de preuve, solveurs)

En pratique on combine les deux : on prouve les composants critiques et l'on teste le système complet dans son environnement réel.

Notions de base du test

Erreur, défaut, défaillance



- ▶ Le test révèle des **défaillances** ; le débogage remonte au **défaut** ; l'analyse des causes remonte à l'**erreur**.
- ▶ Une défaillance peut être *bénigne* (résultat inattendu, le système continue) ou aller jusqu'à la *panne* (arrêt total ou partiel du service).
- ▶ Vocabulaire de l'ISTQB et de l'IEEE. En sûreté de fonctionnement (Laprie), on dit plutôt : *faute* → *erreur* (état interne erroné) → *défaillance*.

Un défaut ne provoque pas toujours une défaillance

Spécification : `max_liste l` renvoie le plus grand élément de la liste non vide `l`.

```
let rec max_liste = function
| [] -> invalid_arg "max_liste"
| [x] -> x
| x :: t ->
    let m = max_liste t in
    if x < m then x else m
(* défaut : < au lieu de > *)
```

Entrée	Sortie	Verdict
[7]	7	succès
[4; 4]	4	succès
[1; 3]	1	échec

- ▶ [7] : le défaut n'est **pas atteint**;
- ▶ [4; 4] : atteint, mais l'état reste **correct**;
- ▶ [1; 3] : atteint, état erroné, **propagé** jusqu'à la sortie.

Pour qu'un test révèle un défaut, il faut **atteindre**, **infecter**, **propager** (modèle RIP), puis que l'oracle **observe** l'écart.

Origines et types de défauts

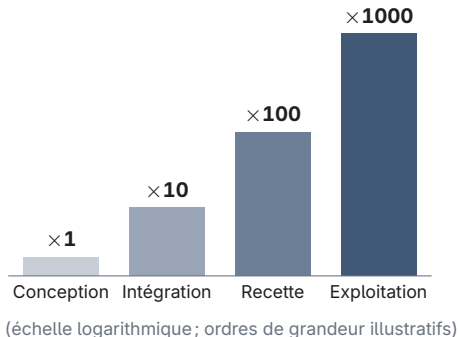
Causes

- ▶ erreur humaine ;
- ▶ spécification mal comprise ou mal interprétée ;
- ▶ erreur de réalisation ;
- ▶ oubli lors de la conception.

Type	Exemple
Calcul	$x + 2$ au lieu de $y + 2$
Logique	$a > b$ au lieu de $a < b$
Entrée/sortie	mauvais format, mauvaise conversion
Traitement des données	accès hors bornes (Heartbleed)
Interface	unités incohérentes entre modules (Mars Climate Orbiter)
Définition des données	type trop petit pour la plage de valeurs (Ariane 5)

Beaucoup de logiciels fonctionnent correctement malgré de nombreux défauts, parce que ceux-ci ne sont **jamais exercés** en exploitation... jusqu'au jour où le contexte change (cf. Ariane 5).

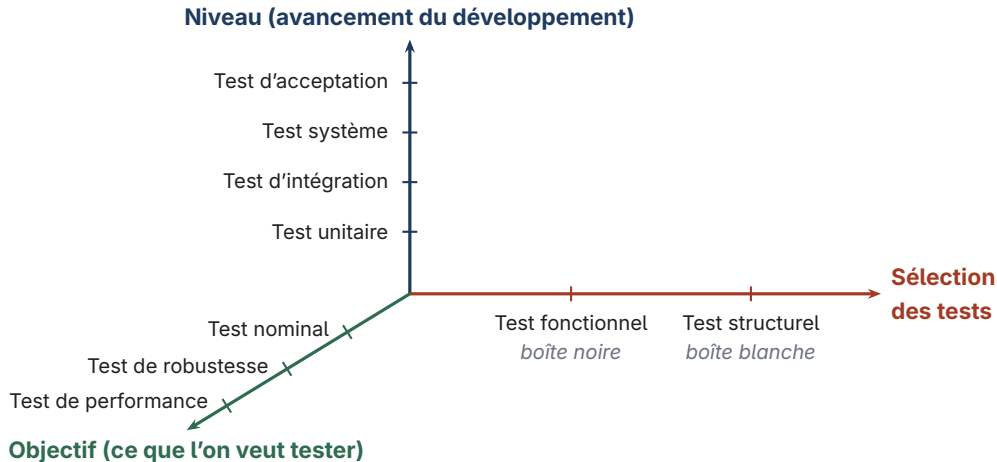
Plus un défaut est découvert tard, plus il coûte cher



- ▶ Côté **utilisateur** : interruption de service, erreurs de calcul ou de facturation, accidents graves.
- ▶ Côté **fournisseur** : corriger après livraison coûte souvent **jusqu'à 100 fois plus** qu'en phase d'exigences ou de conception (Boehm & Basili).

⇒ détecter les défauts **au plus tôt** ;
⇒ commencer les activités de tests **dès la spécification**.

Les trois dimensions du test



Tout test se situe sur ces trois axes : p. ex. un *test unitaire, structurel, de robustesse*.

Niveaux de test : unitaire et intégration

Test unitaire

Teste un composant **isolément** (fonction, module, classe).

- ▶ répond à des scénarios établis à partir de la conception détaillée ;
- ▶ exécuté dans l'environnement de développement, avec un *framework* : Alcotest ou OUnit (OCaml), JUnit (Java), pytest (Python)... ;
- ▶ simple à mettre en œuvre, rapide, rejoué à chaque modification.

Test d'intégration

Teste un ensemble de composants **qui viennent d'être assemblés**.

- ▶ *tests de validité* : le nouveau module fait ce qui est demandé ;
- ▶ *tests de non-régression* : l'ajout d'un module ne dégrade pas le comportement fonctionnel ou non fonctionnel (stabilité, performances) de l'existant.
Essentiel avec l'intégration continue.

Niveaux de test : système et acceptation

Test système

Teste le **système complet** sur son futur site d'exploitation, dans les conditions opérationnelles... et au-delà (surcharge, défaillance matérielle, etc.).

Cible : le système sur l'architecture du client.

Test d'acceptation (recette)

Dernière étape avant la mise en exploitation.

- ▶ conditions de test définies par le **futur utilisateur**, et non par le développeur ;
- ▶ révèle souvent des omissions ou des erreurs dans l'expression des besoins ;
- ▶ il s'agit alors d'erreurs de **validation**, et non de vérification.

Objectifs : ce que l'on veut tester

Test nominal

Aussi appelé *test de bon fonctionnement*.

Entrées **valides**, conformes à la spécification.

Test de robustesse

Entrées **invalides** ou inattendues : le système doit les rejeter proprement, sans planter ni corrompre son état.

Test de performance

- ▶ *test de charge* : montée en charge réaliste ;
- ▶ *test de stress* : demandes anormales de ressources.

Règle usuelle : les tests nominaux passent **avant** les tests de robustesse ; inutile d'éprouver les cas d'erreur si le cas normal échoue.

Le *fuzzing*, qui génère massivement des entrées malformées, est un test de robustesse automatisé, très utilisé en sécurité.

Le processus de test

Valable pour tout type de test :

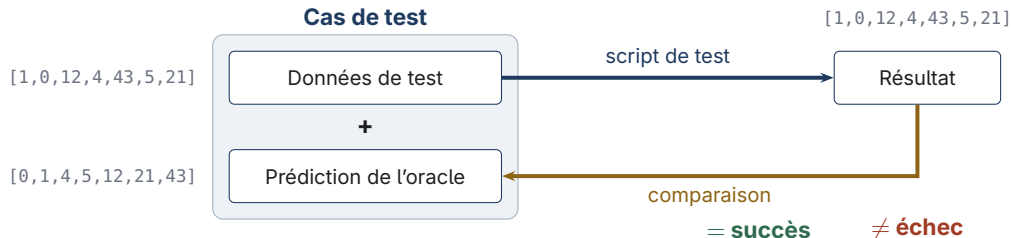
1. définir un **cas de test** (CT) : le scénario à appliquer ;
2. le concrétiser par des **données de test** (DT) ;
3. prédire le résultat attendu : la **prédiction de l'oracle** ;
4. exécuter le **script de test** qui applique le CT aux DT ;
5. comparer le résultat obtenu à la prédiction : le **verdict** ;
6. consigner le résultat : **succès** ou **échec**.

Une **suite de tests** est un ensemble de cas de test, avec leurs données et leurs oracles.

Processus de test : un exemple

Programme : trier un tableau d'entiers en **supprimant les doublons**.

Cas de test : un tableau de N entiers distincts.



Ici le résultat n'est pas trié : verdict **échec**.

Un premier test unitaire en OCaml (Alcotest)

```
let test_nominal () =  
  Alcotest.(check (list int)) "tri sans doublons"  
    [0; 1; 4; 5; 12; 21; 43]          (* oracle *)  
    (tri_unique [1; 0; 12; 4; 43; 5; 21]) (* DT *)  
  
let test_doublons () =  
  Alcotest.(check (list int)) "doublons supprimés"  
    [1; 2; 3] (tri_unique [3; 1; 2; 3; 1])  
  
let () =  
  Alcotest.run "tri_unique"  
    [ ("unitaires",  
      [ Alcotest.test_case "cas nominal" 'Quick test_nominal;  
        Alcotest.test_case "doublons"      'Quick test_doublons ]) ]
```

Sur une implémentation
défectueuse qui conserve les
doublons (extrait de la sortie) :

```
[OK]    unitaires 0 cas nominal.  
[FAIL] unitaires 1 doublons.  
FAIL doublons supprimés  
Expected: '[1; 2; 3]'  
Received: '[1; 1; 2; 3; 3]'
```

Le premier test, *sans doublons*, ne
pouvait pas révéler ce défaut : le
choix des données de test
est décisif.

- ▶ **Choisir** les cas et les données de test (critères de sélection).
- ▶ **Prévoir** le résultat attendu : le *problème de l'oracle*.
- ▶ **Exécuter** le programme sur les données de test : en test unitaire, le code environnant est souvent incomplet $\Rightarrow$ doublures de test (*stubs, mocks*) ; difficile pour les systèmes embarqués et cyber-physiques.
- ▶ **Calculer le verdict** : délicat lorsque la sortie est approchée (calcul flottant), temporelle ou physique.
- ▶ **Décider quand s'arrêter** : mesure de *couverture*, avec un critère bien choisi.
- ▶ **Rejouer** les tests à chaque modification (test de non-régression) : coût de maintenance de la suite de tests, besoin de priorisation.

Doublures de test

Pour tester un composant isolément, on remplace ses dépendances (base de données, réseau, capteur, horloge...) par des **doublures** :

Doublure	Rôle
<i>Dummy</i>	objet passé en paramètre mais jamais utilisé
<i>Stub</i> (bouchon)	renvoie des réponses prédéfinies aux appels du composant testé
<i>Spy</i>	bouchon qui enregistre en plus les appels reçus, pour les vérifier après coup
<i>Mock</i>	objet programmé avec des <i>attentes</i> sur les appels qu'il doit recevoir ; le test échoue si elles ne sont pas satisfaites
<i>Fake</i>	implémentation simplifiée mais fonctionnelle (p. ex. base de données en mémoire)

Attention : on teste alors le composant *dans un environnement supposé*.
Les tests d'intégration restent indispensables.

Le problème de l'oracle

L'**oracle** décide si le résultat obtenu est acceptable (parfois, c'est un expert du domaine).

Oracles complets et automatisables

- ▶ résultat simple à vérifier (p. ex. solution d'une équation : on la réinjecte) ;
- ▶ comparaison à une référence : table de résultats, version précédente, autre implémentation (*test dos à dos*).

Oracles partiels mais automatisés

- ▶ oracle minimal (pas de plantage), assertions et *contrats* ;
- ▶ **propriétés** vérifiées sur des entrées générées (*property-based testing*) ;
- ▶ *relations métamorphiques* : sans connaître la sortie exacte, on sait relier deux exécutions ($\sin(\pi - x) = \sin x$).

Test fondé sur les propriétés (QCheck)

```
let oracle = List.sort_uniq compare (* référence *)

let dos_a_dos =
  QCheck.Test.make ~count:1000
    ~name:"dos à dos avec List.sort_uniq"
    QCheck.(list small_int)
    (fun l -> tri_unique l = oracle l)

let croissante =
  (* oracle partiel *)
  QCheck.Test.make ~count:1000
    ~name:"strictement croissante"
    QCheck.(list small_int)
    (fun l -> strictement_croissante (tri_unique l))

let () =
  QCheck_runner.run_tests_main [dos_a_dos; croissante]
```

L'outil génère 1000 listes aléatoires, puis **réduit** (*shrinking*) tout contre-exemple trouvé :

```
Test strictement croissante
failed (9 shrink steps):
[4; 4]
```

Piège instructif

Avec `list int`, les deux tests *passent* : des entiers tirés sur 63 bits ne se répètent presque jamais, donc aucun doublon n'est généré. Le **générateur** fait partie du test.

Sélection des tests

Le test exhaustif est impraticable : **comment choisir les scénarios à jouer ?**

	Test fonctionnel <i>boîte noire</i>	Test structurel <i>boîte blanche</i>	Test aléatoire / statistique <i>boîte noire</i>
Source	la spécification (cas d'utilisation, <i>user stories</i> ...)	le code source (instructions, branches, chemins)	le domaine des entrées, éventuellement un profil d'usage
Atouts	indépendant du code ; oracle plus facile	critères précis et mesurables ; script de test facile	automatisable à grande échelle ; trouve des cas imprévus
Limites	spécifications parfois imprécises ; concrétisation des DT	ne voit pas les fonctionnalités <i>oubliées</i> ; oracle difficile	oracle indispensable et souvent délicat ; cas rares peu atteints

Ces familles sont **complémentaires**.

Prochain cours : le **test fonctionnel** (partitions, valeurs limites, combinatoire).

Synthèse

- ▶ **Validation** : le bon produit ;
- ▶ **Vérification** : le produit correct vis-à-vis de sa spécification.
- ▶ La V&V accompagne *tout* le cycle de vie : cycle en V, TDD, intégration continue. Plus un défaut est trouvé tard, plus il coûte.
- ▶ Le **test** trouve des défauts mais ne prouve pas leur absence ; la **preuve** garantit la correction, mais seulement vis-à-vis d'une spécification et d'un modèle. Les deux se complètent.
- ▶ Erreur (humaine) → défaut (dans le code) → défaillance (observable). Un défaut doit être atteint, infecter l'état et se propager pour être révélé.
- ▶ Un test se caractérise par son **niveau**, son **objectif** et son **mode de sélection** ; il repose sur un cas de test, des données, un oracle et un verdict.
- ▶ Les deux grandes difficultés : **sélectionner** les tests et disposer d'un **oracle**.

Lectures recommandées (1/2)

Ouvrages

- ▶ P. Ammann, J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2^e éd., 2017.
- ▶ A. P. Mathur. *Foundations of Software Testing*. Pearson, 2008.
- ▶ G. J. Myers, C. Sandler, T. Badgett. *The Art of Software Testing*. Wiley, 3^e éd., 2011.
- ▶ I. Sommerville. *Software Engineering*. Pearson, 10^e éd., 2015.
- ▶ G. Meszaros. *xUnit Test Patterns*. Addison-Wesley, 2007.
- ▶ A. Zeller, R. Gopinath, M. Böhme, G. Fraser, C. Holler. *The Fuzzing Book*. CISPAs, 2024 : <https://www.fuzzingbook.org>.
- ▶ ISTQB. *Glossary of Testing Terms* : <https://glossary.istqb.org>.

Études de cas

- ▶ N. G. Leveson, C. S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer* 26(7), 1993.
- ▶ J.-L. Lions et al. *Ariane 5 Flight 501 Failure: Report by the Inquiry Board*, 1996.
- ▶ G. Tasse. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. NIST Planning Report 02-3, 2002.

Lectures recommandées (2/2)

Articles de référence

- ▶ C.A.R. Hoare. An axiomatic basis for computer programming. *CACM* 12(10), 1969.
- ▶ E. W. Dijkstra. *Notes on Structured Programming*. EWD249, 2^e éd., 1970.
- ▶ B. W. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.
- ▶ G. Bernot, M.-C. Gaudel, B. Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal* 6(6), 1991.
- ▶ K. Claessen, J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ICFP*, 2000.
- ▶ B. Boehm, V. R. Basili. Software defect reduction top 10 list. *IEEE Computer* 34(1), 2001.
- ▶ A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE TDSC* 1(1), 2004.
- ▶ X. Leroy. Formal verification of a realistic compiler. *CACM* 52(7), 2009.
- ▶ G. Klein et al. seL4: formal verification of an OS kernel. *SOSP*, 2009.
- ▶ X. Yang, Y. Chen, E. Eide, J. Regehr. Finding and understanding bugs in C compilers. *PLDI*, 2011.
- ▶ E. T. Barr et al. The oracle problem in software testing: a survey. *IEEE TSE* 41(5), 2015.