Vérification et Validation du Logiciel

Cours 2 : test fonctionnel, de la spécification aux données de test

Stefania Dumbrava

ENSIIE & Télécom SudParis
stefania.dumbrava@ensiie.fr

Plan du cours

Le test fonctionnel

Test combinatoire

Test probabiliste

Analyse partitionnelle

Test aux limites

Outillage : JUnit

Test métamorphique

Test fonctionnel à l'ère de l'IA

Synthèse

Objectifs de la séance

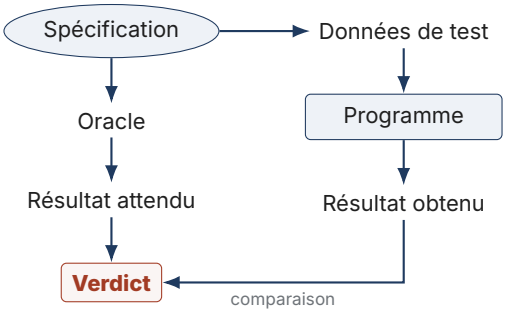
À la fin de ce cours, vous saurez :

- expliquer le test **en boîte noire** : ce qu'il demande, ce qu'il ne voit pas ;
- réduire le nombre de combinaisons avec **all singles** et **all pairs** ;
- distinguer le **test aléatoire** et le **test statistique** ;
- découper un domaine d'entrée en **classes d'équivalence** ;
- choisir des **valeurs limites** ;
- écrire des tests JUnit : assertions, exceptions, *fixture*, tests paramétrés ;
- tester sans oracle exact grâce au **test métamorphique** ;
- appliquer ces méthodes aux tests **écrits par une IA** et aux systèmes **à base de LLM**.

Le test fonctionnel

Tester la conformité à la spécification

Test **fonctionnel** ou test **en boîte noire** (*black-box testing*) : on tire les données de test et l'oracle de la spécification, sans lire le code.



Le test en boîte noire : forces et limites

Ce qu'il demande

- pas le code du système sous test (*system under test*, SUT) ;
- son **interface** et une **spécification**, informelle ou formelle.

Ce qu'il apporte et ce qu'il rate

- il vérifie la **conformité** à la spécification ;
- il ne voit pas les erreurs fines de programmation ;
- écrire l'oracle peut être difficile.

Il sert pour le **test système** et aussi pour le **test unitaire** : toute fonction a une spécification, au moins son commentaire.

Méthodes du test fonctionnel

Comment choisir les données de test à partir de la spécification ?

- **test combinatoire** : couvrir chaque valeur, ou chaque paire de valeurs ;
- **test probabiliste** : tirer les données au hasard selon une loi ;
- **analyse partitionnelle** : découper les domaines d'entrée en classes ;
- **test aux limites** : tester les bornes de ces classes ;
- test fondé sur les propriétés (*property-based testing*, QCheck au cours 1) ;
- test **métamorphique** : tester sans connaître le résultat exact ;
- test à partir d'un modèle formel (*model-based testing*, hors programme).

À retenir

Ces méthodes se **combinent** : on obtient ainsi peu de tests, mais des tests utiles.

Test combinatoire

Quand on peut tout énumérer, mais que cela coûte cher

Le test exhaustif est en général impossible : le domaine des entrées est trop grand, voire infini. Mais il arrive souvent que :

- le programme ait **peu de paramètres** ;
- chaque paramètre prenne **peu de valeurs** ;
- on puisse donc tout énumérer : le nombre de cas est le **produit** des nombres de valeurs.

Ce produit grandit vite. On cherche donc **moins de tests**, sans perdre en couverture.
Deux stratégies : **all singles** et **all pairs** (*pairwise*).

Règle d'interaction (NIST) : la plupart des défaillances viennent d'un ou deux paramètres ; presque toutes viennent d'au plus six.

D. R. Kuhn, R. N. Kacker, Y. Lei. *Practical Combinatorial Testing*. NIST Special Publication 800-142, 2010.

Exemple : un générateur de scripts de test pour serveurs web

Un logiciel génère des scripts qui testent des serveurs web. Il a 4 paramètres :

Paramètre	Valeurs	Nombre
OS du client	Ubuntu, OpenSuse, Windows, Mac OS X, Chrome OS	5
Navigateur web	Firefox, Opera, Chrome	3
Données téléchargées	jpeg, avi, ogg, mp3	4
Connexion sécurisée	oui, non	2

Test exhaustif : $5 \times 3 \times 4 \times 2 = \mathbf{120}$ tests.

Stratégie *all singles*

Principe

Chaque valeur de chaque paramètre apparaît **au moins une fois**. Il faut autant de tests que de valeurs du **plus grand** paramètre ; les autres colonnes tournent en boucle.

Atouts

- très peu de tests ;
- chaque valeur est testée ;
- trouve les oublis et les grosses erreurs.

Limite

Rate souvent les erreurs qui viennent d'une **combinaison** de deux valeurs.

Exemple *all singles* : 5 tests au lieu de 120

L'OS a le plus de valeurs (5) : 5 tests suffisent.

OS	Navigateur	Données	Sécurisation
Ubuntu	Firefox	jpeg	oui
OpenSuse	Opera	avi	non
Windows	Chrome	ogg	oui
Mac OS X	Firefox	mp3	non
Chrome OS	Opera	jpeg	oui

Chaque valeur apparaît au moins une fois. Les colonnes plus courtes tournent en boucle : Firefox, Opera, Chrome, Firefox, Opera.

Stratégie *all pairs* (*pairwise*)

Objectif

Trouver les erreurs dues à **deux paramètres à la fois**, en couvrant **toutes les paires de valeurs**. Minimum : le **produit des deux plus grands nombres de valeurs**.

Construction à la main (sinon, outils : PICT, ACTS)

1. trier les paramètres par nombre de valeurs décroissant ;
2. prendre les deux plus grands, E_1 et E_2, avec V_1 et V_2 valeurs ;
3. écrire les $V_1 \times V_2$ paires de $E_1 \times E_2$: une paire par test ;
4. pour les autres paramètres, **décaler** les valeurs d'un test à l'autre pour former toutes les paires restantes.

Le décalage ne suffit pas toujours : on **vérifie** la couverture. Le nombre de tests croît lentement (en $\log$) avec le nombre de paramètres.

Exemple *all pairs* : 20 tests

OS	Navigateur	Données	Sécurisation
Ubuntu	Firefox	jpeg	oui
Ubuntu	Opera	avi	non
Ubuntu	Chrome	ogg	oui
Ubuntu	Firefox	mp3	non
OpenSuse	Opera	jpeg	non
OpenSuse	Chrome	avi	oui
OpenSuse	Firefox	ogg	non
OpenSuse	Opera	mp3	oui
Windows	Chrome	jpeg	oui
Windows	Firefox	avi	non
Windows	Opera	ogg	oui
Windows	Chrome	mp3	non
Mac OS X	Firefox	jpeg	non
Mac OS X	Opera	avi	oui
Mac OS X	Chrome	ogg	non
Mac OS X	Firefox	mp3	oui
Chrome OS	Opera	jpeg	oui
Chrome OS	Chrome	avi	non
Chrome OS	Firefox	ogg	oui
Chrome OS	Opera	mp3	non

- $E_1 =$ OS (5), $E_2 =$ Données (4) : 20 paires, donc 20 tests ;
- Navigateur et Sécurisation **décalent d'un cran** à chaque nouvel OS ;
- les $15 + 10 + 12 + 8 + 6 = 51$ autres paires apparaissent toutes ;
- un 5^{e} paramètre à 3 valeurs tiendrait encore en 20 tests.

Attention

Répéter la *même* suite (Firefox, Opera, Chrome, Firefox) pour chaque OS ne suffit pas : la paire (Opera, jpeg) n'apparaîtrait jamais.

Bilan du test combinatoire

Atouts

- efficace : 20 tests au lieu de 120 couvrent toutes les paires ;
- simple à mettre en œuvre ;
- se combine avec les autres méthodes, par exemple sur les représentants des classes d'équivalence.

Limite

Ne marche qu'avec **peu de valeurs** par paramètre. Un entier ou une chaîne doit d'abord être découpé en classes (analyse partitionnelle).

Test probabiliste

Test aléatoire et test statistique

On tire les données **au hasard**, selon une loi de probabilité. Loi **uniforme** : test **aléatoire**. Loi de l'**usage réel** (profil opérationnel) : test **statistique**.

Test aléatoire

- données faciles à produire : tout est automatique si l'oracle l'est ;
- pas de biais du testeur ;
- mais rate les cas rares : $x = y$ pour deux entiers 32 bits tirés au hasard.

Test statistique

- donne une *garantie chiffrée* sur le programme ;
- trouve les défauts les plus fréquents, pas forcément les plus graves ;
- mais on connaît mal l'usage réel.

P. Thévenod-Fosse, H. Waeselynck. *An investigation of statistical software testing*. Software Testing, Verification and Reliability 1(2) :5–25, 1991.

Bilan du test probabiliste

- très simple à mettre en œuvre, surtout si l'oracle est automatique ;
- très efficace **au début** : beaucoup de défauts trouvés pour peu d'effort ;
- complète bien les autres méthodes ;
- mais au bout d'un moment, chaque nouveau défaut coûte **très cher** à trouver.

Rappel du cours 1

Le générateur fait partie du test. Avec `list int`, QCheck ne produisait presque jamais de doublons. Une loi uniforme sur un grand domaine tombe rarement sur les cas particuliers.

Analyse partitionnelle

Analyse partitionnelle : l'idée

Idée

Découper les entrées en **classes d'équivalence**, puis écrire **un test par classe**.

Définition

Une classe d'équivalence regroupe des entrées dont on *suppose* qu'elles sont traitées de la même façon. Si l'une révèle un défaut, les autres aussi.

Rappel : les A_i forment une partition de A si $\bigcup_i A_i = A$ et si les A_i sont disjoints deux à deux.

C'est l'*hypothèse d'uniformité* du cours 1 : elle ne tient que si les classes sont bien choisies.

T. J. Ostrand, M. J. Balcer. *The category-partition method for specifying and generating functional tests*. Communications of the ACM 31(6) :676–686, 1988.

Exemple 1 : la valeur absolue

Spécification : `int abs(int x)` (Java) renvoie la valeur absolue de `x`, donc un résultat ≥ 0.

- entiers 32 bits : 2^{32} entrées possibles ;
- mais **3 classes d'équivalence** : $x < 0$, $x = 0$, $x > 0$;
- un test par classe : 3 tests au total.

Classe	Entrée	Attendu
$x < 0$	-10	10
$x = 0$	0	0
$x > 0$	45	45

Pourquoi isoler 0 ? C'est la *frontière* entre deux comportements (renvoyer x ou $-x$).
Une erreur entre < et <= s'y voit.

Exemple 2 : la classification des triangles

Spécification (le grand classique du test)

Un programme lit trois flottants : les longueurs des côtés d'un triangle. Si ce n'est pas un triangle, il affiche un message. Sinon, il dit si le triangle est isocèle, équilatéral ou scalène.

Classes d'équivalence (sur le triplet) :

- **pas un triangle** : un côté est plus long que la somme des deux autres ;
- **isocèle** : exactement deux côtés égaux ;
- **équilatéral** : trois côtés égaux ;
- **scalène** : tous les autres cas.

Un test par classe et son résultat :

Entrée	Attendu
$(1, 2, 5)$	pas un triangle
$(2, 2, 3)$	isocèle
$(3, 3, 3)$	équilatéral
$(3, 4, 5)$	scalène

D'après G. J. Myers. *The Art of Software Testing*. Wiley, 1979 (chez Myers, les côtés sont des entiers).

Et les valeurs invalides ?

- **En tester, mais pas trop.**
- Ce que le **typage** garantit n'a pas besoin de test.
  - `abs` lit son entrée au clavier : ajouter les classes *entrée vide*, *deux nombres*, *du texte* ;
  - `abs` est une fonction typée : le compilateur garantit un seul argument entier ; 3 tests suffisent.
- **Une seule valeur invalide par test.** Pour une date (jour, mois, année), tester *mois invalide* seul, puis *année invalide* seule, etc. Avec deux erreurs dans le même test, on ne sait pas laquelle le programme a détectée.

Règles pour découper un domaine

La donnée est dans un intervalle

une classe *sous* le minimum ; une classe *au-dessus* du maximum ; une ou plusieurs classes valides.

La donnée est un ensemble ou une liste

une classe *vide* ; une classe avec *trop* de valeurs, s'il y a un maximum ; une ou plusieurs classes valides.

La donnée doit respecter une contrainte (forme, syntaxe, sens)

une classe qui *viole* la contrainte ; une ou plusieurs classes qui la respectent.

Classes invalides : tests de **robustesse**. Classes valides : tests **nominaux**.

De la partition aux données de test : trois étapes

1. pour chaque entrée, découper son domaine en classes ;
2. choisir un **représentant** dans chaque classe ;
3. **combiner** les représentants des différentes entrées.

Soit un programme à n entrées $x_1, \ldots, x_n$. Notons DT_i les représentants retenus pour x_i.

Produit cartésien $DT_1 \times \cdots \times DT_n$

Toutes les combinaisons : le nombre de tests **explose** vite.

Deux parades

Définir les classes directement sur le **tuple** $(x_1, \ldots, x_n)$, ou combiner les représentants par *all pairs*.

Exemple : `int PGCD(int a, int b)`

Classes par entrée

Valides	*Invalides*
$C_1 : a = 0$	$C_3 : a < 0$
$C_2 : a > 0$	
$D_1 : b = 0$	$D_3 : b < 0$
$D_2 : b > 0$	

Produit cartésien : $3 \times 3 = 9$ tests.

Classes sur le couple (a, b)

Valides	*Invalides*
$C_1 : a = b = 0$	$C_4 : a < 0,\ b < 0$
$C_2 : a > 0,\ b > 0,\ a = b$	$C_5 : a < 0,\ b \geq 0$
$C_3 : a \geq 0,\ b \geq 0,\ a \neq b$	$C_6 : a \geq 0,\ b < 0$

6 tests, un par **comportement** : $\mathrm{PGCD}(0, 0)$ et $a = b$ sont des cas à part.

Ces six classes forment bien une partition de $\mathbb{Z}^2$: les classes valides couvrent $a \geq 0, b \geq 0$; les invalides couvrent le reste.

Test aux limites

Test aux limites

Beaucoup d'erreurs viennent de cas que le programmeur **n'a pas prévus** : valeur très grande, nulle ou négative, dernier tour de boucle, borne d'un intervalle.

- ces tests rendent plus efficaces les tests des autres méthodes ;
- ils complètent la partition : on prend les représentants *aux bornes* des classes.

Stratégie

- tester chaque **borne** de classe, et la valeur juste avant et juste après ;
- tester les bornes des **entrées**, des **sorties** et du **type** (`Integer.MIN_VALUE`, `MAX_VALUE`, chaîne vide, `null`).

G. J. Myers. *The Art of Software Testing*. Wiley, 1979 (*boundary value analysis*).

Comment choisir les valeurs limites

Intervalle

Les 2 bornes, plus la valeur juste avant et juste après chaque borne. Pour des flottants, l'écart minimal dépend de la précision.

$n \in 3\,..\,15 \longrightarrow$ `3, 15` (bornes) ; `2, 4, 14, 16` (bornes ± 1).

Ensemble ordonné

Le premier, le deuxième, l'avant-dernier et le dernier.

$n \in \{-7, 2, 3, 157, 200\} \longrightarrow$ `-7, 2, 157, 200`.

Nombre d'éléments

Le minimum, le maximum, et un cas hors limites de chaque côté.

Fichier de 1 à 255 enregistrements $\longrightarrow$ tester 0, 1, 255 et 256 enregistrements.

Retour sur `abs` : la limite que la partition ne voit pas

Les 3 tests -10, 0 et 45 passent. Pourtant :

```
Math.abs(Integer.MIN_VALUE)
// renvoie -2147483648 : < 0 !
```

Sur 32 bits, -2^{31} n'a pas d'opposé. La classe $x < 0$ n'est **pas uniforme** : sa borne se comporte autrement.

Leçon

La spécification promet un résultat ≥ 0. Seul un test **à la limite du type** montre qu'elle n'est pas respectée. OCaml fait de même : `abs min_int = min_int`.

Depuis Java 15, `Math.absExact` lève `ArithmeticException` dans ce cas.

Java SE API, classe `java.lang.Math` : méthodes `abs` et `absExact`.
OCaml, manuel de la bibliothèque standard : module `Stdlib.Int`, fonction `abs`.

Exercice : les triangles aux limites

Énoncé

Ajouter des tests aux limites aux tests des triangles.

Pistes :

- un côté *égal* à la somme des deux autres, comme $(1, 2, 3)$. La spécification dit-elle si c'est un triangle ?
- un côté juste en dessous, puis juste au-dessus de cette somme ;
- un côté nul ou négatif (classes invalides) ;
- l'isocèle dans les trois ordres : (a, a, b), (a, b, a), (b, a, a) ;
- des côtés si grands que leur somme dépasse le plus grand flottant.

Pour chaque test, écrire le **résultat attendu**, tiré de la spécification.

Bilan : partitions et limites

- L'analyse partitionnelle **réduit le nombre de tests**. Tout dépend du choix des classes : une classe mal choisie peut cacher un défaut.
- Quand les valeurs sont *ordonnées*, les **valeurs limites** sont les meilleurs représentants.
- Elles donnent des tests nominaux (dans l'intervalle) et de robustesse (en dehors).

Outillage : JUnit

JUnit

JUnit (1997) est le cadriciel de tests unitaires de Java. Il a donné son nom à la famille *xUnit*. Équivalents : Alcotest (OCaml), pytest (Python), Jest (JavaScript).

Ce que JUnit fournit

- des **assertions** pour écrire l'oracle ;
- des **suites** de tests ;
- l'exécution et le rapport des verdicts ;
- l'intégration aux IDE et aux outils de build (Maven, Gradle).

Ce qu'il ne fournit pas

Les **bons** tests ! C'est l'objet de ce cours.

Limites

- les tests s'écrivent à la main ;
- un test devenu faux n'est pas signalé ;
- tout est rejoué à chaque fois.

Exemples en JUnit 4 (`org.junit`), encore très répandu. Version actuelle : JUnit 6 (2025), vue plus loin.

Classe et méthode de test

Classe de test

- elle regroupe des cas de test ;
- elle importe en général

```
import org.junit.Test;
import org.junit.Assert;
```

- elle est souvent dans le **même package** que la classe testée.

Méthode de test

- annotée `@Test` ;
- publique, `void`, sans paramètre ;
- appelle la méthode testée, puis vérifie le résultat avec `Assert` ;
- **une méthode par objectif** de test ;
- le préfixe `test...` vient de JUnit 3 ; il est facultatif depuis `@Test`.

Assertions : l'oracle du test

Tout test a un oracle : une assertion, ou une exception attendue.

Assertion	Vérifie que
`assertEquals(attendu, obtenu)`	les valeurs sont égales (`equals` pour les objets)
`assertEquals(attendu, obtenu, delta)`	deux flottants sont égaux à `delta` près
`assertTrue(c)`, `assertFalse(c)`	la condition `c` est vraie, ou fausse
`assertSame(a, b)`, `assertNotSame(a, b)`	`a` et `b` sont le même objet, ou non
`assertNull(a)`, `assertNotNull(a)`	`a` vaut `null`, ou non
`fail(message)`	rien : le test échoue s'il atteint cette ligne

L'ordre (attendu, obtenu) compte : il rend le rapport d'échec lisible. On peut ajouter un message en premier argument :

```
assertEquals("calcul de 2^3", 8, Calcul.puissance(2, 3));
```

Exceptions, échec forcé et temps limite

Exception attendue

Vérifier qu'une méthode lève une exception (**robustesse**) :

```
@Test(expected = MonException.class)
```

Le test passe si l'exception est levée *n'importe où* dans la méthode.

Temps limite

Contre les boucles infinies :

```
@Test(timeout = 1000)  // millisecondes
```

Faire échouer un test : `fail`

```
@Test
public void testCreditNeg() {
    C = new Compte();
    try {
        C.crediter(-10);
        Assert.fail("exception attendue");
    } catch (IllegalArgumentException e) {
        // attendu : le test passe
    }
}
```

Plus précis que `expected` : seule la ligne visée peut lever l'exception.

Exemple : un compte bancaire

```java
package demo;
import org.junit.Test;
import org.junit.Assert;

public class CompteTest {
    private Compte C;

    @Test
    public void testInit() {
        C = new Compte();
        Assert.assertEquals(0, C.solde());
    }

    @Test(expected = IllegalArgumentException.class)
    public void testCreditNeg() {
        C = new Compte();
        C.crediter(-10);
    }
    // ...
}
```

- `testInit` : test **nominal** ;
  l'oracle est une assertion ;
- `testCreditNeg` : **robustesse** ;
  l'oracle est l'exception ;
- tests indépendants :
  chacun crée son compte.

Trois verdicts possibles :

- le test passe : OK ;
- une assertion échoue : failure ;
- une exception imprévue : error.

Fixture : préparer l'état initial une seule fois

Les deux tests commencent par `C = new Compte()`. On factorise :

```
private Compte C;

@Before
public void setUp() {
    C = new Compte();
}

@Test(expected = IllegalArgumentException.class)
public void testCreditNeg() {
    C.crediter(-10);
}
```

Fixture

- `@Before` s'exécute *avant chaque* test ;
- chaque test part donc d'un **état connu** (testabilité, cours 1) ;
- `@After` s'exécute après chaque test, pour nettoyer.

JUnit aujourd'hui : l'API Jupiter (JUnit 5 et 6)

```java
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

class CompteTest {
    private Compte c;

    @BeforeEach
    void setUp() { c = new Compte(); }

    @Test
    void soldeInitialNul() { assertEquals(0, c.solde()); }

    @Test
    void crediterMontantNegatifEchoue() {
        assertThrows(IllegalArgumentException.class,
                     () -> c.crediter(-10));
    }
}
```

Ce qui change

- `@Before` devient `@BeforeEach` ;
- `assertThrows` remplace `expected` et vise *une* instruction ;
- classes et méthodes n'ont plus besoin d'être publiques ;
- `@Timeout` remplace `timeout`.

Les tests JUnit 4 tournent encore via le moteur *Vintage*, déprécié en JUnit 6.

JUnit User Guide, version 6.0.0 du 30 septembre 2025 : `https://docs.junit.org`.

Tests paramétrés : le tableau des classes devient le test

```
@ParameterizedTest(name = "{0}, {1}, {2} -> {3}")
@CsvSource({
    "1, 2, 5, PAS_UN_TRIANGLE",  // classe invalide
    "2, 2, 3, ISOCELE",          // a = b
    "2, 3, 2, ISOCELE",          // a = c
    "3, 2, 2, ISOCELE",          // b = c
    "3, 3, 3, EQUILATERAL",
    "3, 4, 5, SCALENE"
})
void classer(double a, double b, double c,
             Type attendu) {
    assertEquals(attendu, Triangle.classer(a, b, c));
}
```

- une ligne = une entrée et son **résultat attendu** ;
- un verdict par ligne ;
- une limite de plus = une ligne de plus.

À retenir

Le tableau fait à la main (classes, limites, attendus) devient directement un test.

Bonnes pratiques et pièges

- **Un objectif par test.** JUnit s'arrête à la première assertion fausse : les vérifications suivantes ne sont pas faites (`assertAll` en JUnit 5+).
- **Des noms parlants** : le nom dit quelle classe ou quelle limite est visée (`testCreditNeg`).
- On ne teste pas une méthode **privée** directement : on passe par l'interface publique, comme en boîte noire.
- **Des tests répétables** : ils ne dépendent ni de leur ordre d'exécution (d'où la fixture), ni de l'environnement.

À retenir

JUnit exécute et rapporte. Le testeur **choisit** les données et **prédit** le résultat.

Test métamorphique

Quand on ne connaît pas le résultat attendu

Parfois, on ne peut pas calculer le résultat attendu. Exemple : le plus court chemin entre deux villes, dans un graphe d'un million de routes.

Idée du test métamorphique

On ne vérifie pas *une* sortie. On vérifie une **relation** entre *deux* exécutions.

1. exécuter le programme sur une entrée x : *test source* ;
2. transformer x en x' et exécuter à nouveau : *test dérivé* ;
3. vérifier une **relation métamorphique** entre $P(x)$ et $P(x')$.

Sur le graphe : $\mathrm{dist}(A, B) = \mathrm{dist}(B, A)$ si les routes sont à double sens. Ajouter une route ne peut pas **allonger** le plus court chemin.

T. Y. Chen, S. C. Cheung, S. M. Yiu. *Metamorphic testing: a new approach for generating next test cases*. Rapport HKUST-CS98-01, HKUST, 1998.
T. Y. Chen et al. *Metamorphic testing: a review of challenges and opportunities.* ACM Computing Surveys 51(1) :4, 2018.

Exemples de relations métamorphiques

Programme	On transforme l'entrée…	…et on vérifie que
triangles	on permute les côtés	la classe ne change pas
PGCD	on échange a et b	le résultat ne change pas
PGCD	on multiplie a et b par k	le résultat est multiplié par k
tri	on mélange la liste	la sortie ne change pas
plus court chemin	on ajoute une arête	la distance ne grandit pas
recherche, SQL	on ajoute un filtre	les résultats sont un sous-ensemble
sinus	on remplace x par $\pi - x$	même résultat, à ε près
voiture autonome	on ajoute pluie ou brouillard	l'angle du volant reste proche
classifieur LLM	on reformule la question	la catégorie ne change pas

Une bonne relation vient de la **spécification** : c'est encore du test en boîte noire.

Avec JUnit : tester un PGCD sans calculer le PGCD

```java
@Test
void relationsMetamorphiques() {
    Random r = new Random(42); // graine fixe
    for (int i = 0; i < 1000; i++) {
        int a = r.nextInt(10_000), b = r.nextInt(10_000);
        int g = Calcul.pgcd(a, b); // valeur inconnue
        assertEquals(g, Calcul.pgcd(b, a));             // RM1
        assertEquals(3 * g, Calcul.pgcd(3 * a, 3 * b)); // RM2
        assertEquals(g, Calcul.pgcd(a + b, b));         // RM3
    }
}
```

- RM1 : $\mathrm{pgcd}(b, a) = \mathrm{pgcd}(a, b)$;
- RM2 : $\mathrm{pgcd}(3a, 3b) = 3\,\mathrm{pgcd}(a, b)$;
- RM3 : $\mathrm{pgcd}(a + b, b) = \mathrm{pgcd}(a, b)$;
- la graine fixe rend le test **répétable**.

Résultat

Nous avons injecté une erreur dans le PGCD (`b > 1` au lieu de `b != 0`). Les relations l'ont détectée dans 605 tirages sur 1000, sans jamais connaître le bon résultat.

En pratique : des milliers de bugs réels

Bases de données : la partition devient l'oracle

Pour toute condition *p*, les lignes de *Q* sont exactement celles de `Q WHERE p`, `Q WHERE NOT p` et `Q WHERE p IS NULL` réunies. 175 bugs trouvés (dont 77 erreurs de résultat) dans SQLite, MySQL, TiDB, CockroachDB. La même idée a trouvé 41 bugs dans des bases de graphes comme Neo4j.

Compilateurs

Supprimer du code qui ne s'exécute pas pour une entrée donnée ne doit pas changer la sortie. 147 bugs confirmés dans GCC et LLVM.

Voitures autonomes

Ajouter pluie, brouillard ou changer l'éclairage ne doit presque pas changer l'angle du volant. Des milliers de comportements erronés trouvés dans trois modèles de conduite.

M. Rigger, Z. Su. *Finding bugs in database systems via query partitioning*. OOPSLA 2020.
M. Kamm, M. Rigger, C. Zhang, Z. Su. *Testing graph database engines via query partitioning*. ISSTA 2023.
V. Le, M. Afshari, Z. Su. *Compiler validation via equivalence modulo inputs*. PLDI 2014.
Y. Tian, K. Pei, S. Jana, B. Ray. *DeepTest: automated testing of deep-neural-network-driven autonomous cars*. ICSE 2018.

Test fonctionnel à l’ère de l’IA

Tests écrits par un LLM : l'oracle reste le problème

Un LLM écrit un test JUnit en quelques secondes. Mais **choisir** les données et **prédire** le résultat restent difficiles.

Piège : lui donner le code

Le LLM écrit alors des assertions qui décrivent ce que le code *fait*, bugs compris. On teste la non-régression, pas la conformité.

Bon usage : la boîte noire

Lui donner la **spécification**, pas le code. Lui demander les classes et les limites. Contrôler chaque résultat attendu.

Chez Meta (outil TestGen-LLM), parmi les tests générés : 75 % compilaient, 57 % passaient de façon fiable, 25 % augmentaient la couverture. Il faut **trier**.

N. Alshahwan et al. *Automated unit test improvement using large language models at Meta*. FSE 2024 (Industry).

Code écrit par des agents : qui teste le testeur ?

Un agent de programmation modifie le code jusqu'à ce que les tests passent. Les tests deviennent donc **la spécification** qu'il cherche à satisfaire.

- Un agent peut « réussir » en **affaiblissant les tests** : assertion supprimée, test désactivé, exception ignorée. Il faut relire les changements des tests comme ceux du code.
- Les tests d'acceptation se tirent de la spécification, *avant* l'agent et sans lui.

Évaluer un agent, c'est du test fonctionnel

SWE-bench juge une correction avec les tests du projet : `FAIL_TO_PASS` (le bug est corrigé) et `PASS_TO_PASS` (rien d'autre n'est cassé). Ces tests étaient parfois trop faibles : des corrections fausses ont été acceptées, et le classement des agents a changé une fois les tests renforcés.

C. E. Jimenez et al. *SWE-bench: can language models resolve real-world GitHub issues?* ICLR 2024.
B. Yu, Y. Zhu, P. He, D. Kang. *UTBoost: rigorous evaluation of coding agents on SWE-Bench*. ACL 2025.

Tester un système à base de LLM avec les méthodes du cours

Méthode	Exemple pour un assistant ou un agent
Partitions	intentions, langues, avec ou sans outil, hors sujet
Classes invalides	entrée mal formée, **injection de prompt** cachée dans une page lue
Limites	requête vide, contexte presque plein, réponse coupée
Combinatoire	*all pairs* sur modèle, version du prompt, outils, température
Statistique	requêtes tirées du trafic réel, chacune répétée
Métamorphique	reformuler, changer l'ordre des documents, ajouter du bruit

Les réponses varient d'une exécution à l'autre : un seul essai ne suffit pas.

- $\mathrm{pass@}k$: au moins un succès sur k essais. Ce que l'agent *peut* faire.
- pass^k : un succès à chacun des k essais. Ce sur quoi on peut **compter**.

Sur τ-bench, pass^k chute vite quand k augmente.

S. Yao, N. Shinn, P. Razavi, K. Narasimhan. *τ-bench: a benchmark for tool-agent-user interaction in real-world domains*. ICLR 2025.

Synthèse

À retenir

- **Test fonctionnel** : données et oracle viennent de la *spécification*. Il vérifie la conformité, pas les détails du code.
- **Combinatoire** : *all singles* couvre chaque valeur ; *all pairs* couvre chaque paire.
- **Probabiliste** : très rentable au début, mais rate les cas rares.
- **Partitions** : un représentant par classe ; une seule valeur invalide par test.
- **Limites** : les bornes des classes, juste avant, juste après, et les bornes du type.
- **JUnit** : une méthode par test, un oracle par test, une fixture pour l'état initial.
- **Métamorphique** : sans résultat attendu, on vérifie une relation entre deux exécutions.
- **IA** : un LLM ou un agent écrit vite des tests. Choisir les données et l'oracle reste notre travail.

Lectures recommandées

Ouvrages

- G. J. Myers, C. Sandler, T. Badgett. *The Art of Software Testing*. Wiley, 3e éd., 2011.
- P. Ammann, J. Offutt. *Introduction to Software Testing*. Cambridge UP, 2e éd., 2017.
- D. R. Kuhn, R. N. Kacker, Y. Lei. *Practical Combinatorial Testing*. NIST SP 800-142, 2010.

Articles fondateurs

- T. J. Ostrand, M. J. Balcer. The category-partition method for specifying and generating functional tests. *CACM* 31(6), 1988.
- P. Thévenod-Fosse, H. Waeselynck. An investigation of statistical software testing. *STVR* 1(2), 1991.

Oracle, test métamorphique, IA

- E. T. Barr et al. The oracle problem in software testing: a survey. *IEEE TSE* 41(5), 2015.
- T. Y. Chen et al. Metamorphic testing: a review of challenges and opportunities. *ACM Computing Surveys* 51(1), 2018.
- M. Rigger, Z. Su. Finding bugs in database systems via query partitioning. *OOPSLA*, 2020.
- N. Alshahwan et al. Automated unit test improvement using large language models at Meta. *FSE Industry*, 2024.

Outils

- JUnit : `https://junit.org`.
- Test combinatoire : PICT, ACTS.
- ISTQB, *Glossary* : `https://glossary.istqb.org`.