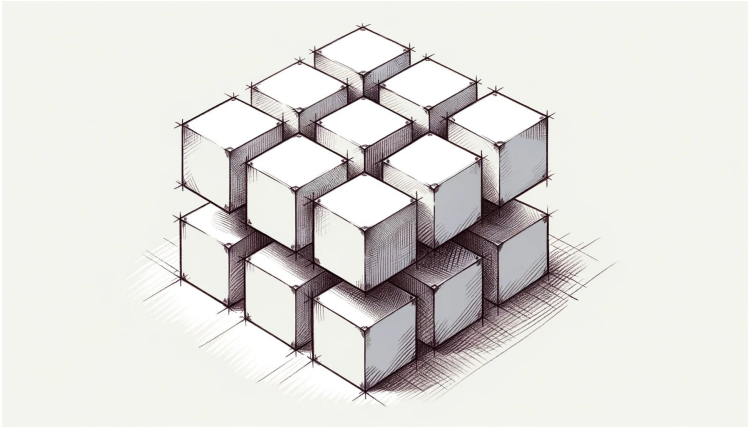
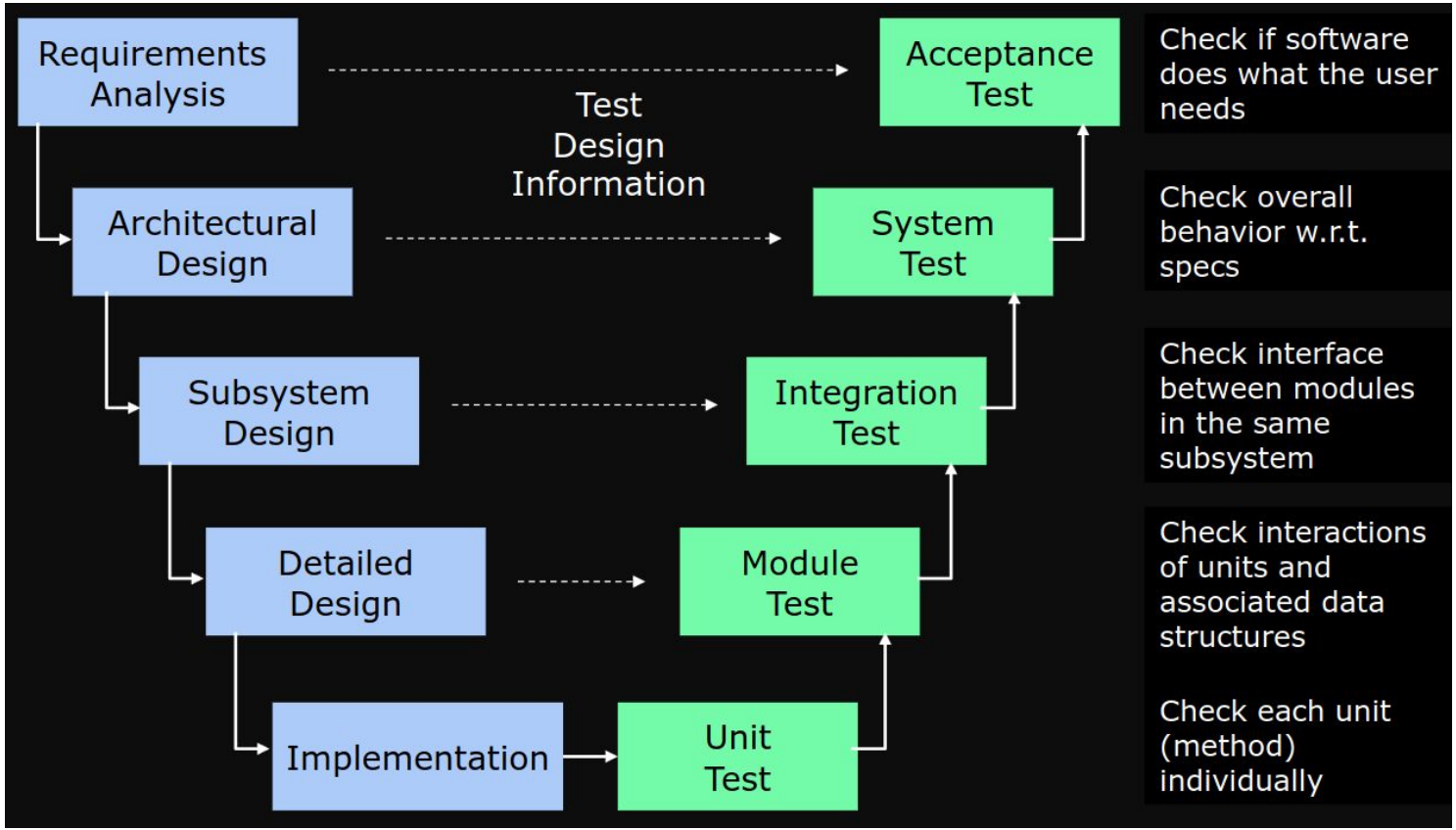
Test Unitaire et JUnit



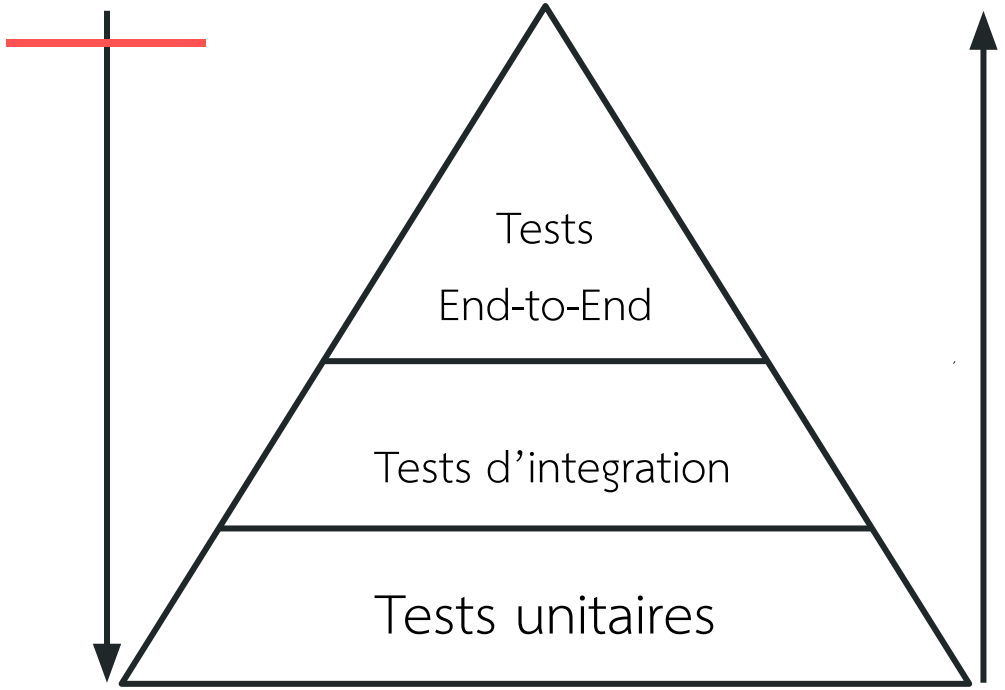
ENSEIRB-MATMECA · I3 GL
Test Logiciel
2023–2024

Granularités et niveaux de test



La Pyramide des Tests (Mike Cohn)

- Plus l'on monte dans la pyramide, plus les tests sont **chers** en couts de developpement et d'execution
- Plus l'on monte dans la pyramide, **moins** l'on a de tests
- Plus l'on monte dans la pyramide, plus les tests sont **fragiles**
- On lance les tests unitaires aussi souvent que possible



Qu'est ce qu'une "unité"?

- Chaque developpeuse et testeur possede une definition *legerement* differente
- En regle generale:
  - En programmation procedurale : une **procedure**
  - En programmation fonctionnelle : une **fonction**
  - En programmation orientee objet : une methode, une **classe**
- On teste une unite en interagissant avec son **interface**
- Ici, on ecrira *des* tests pour `BankAccount()`, `deposit()`, et `withdraw()`

```
class BankAccount {
  private double balance;
  public BankAccount(double initial) { /* ... */ }
  public void deposit(double amount) { /* ... */ }
  public void withdraw(double amount) { /* ... */ }
}
```

Quelques scénarios de test : application bancaire

- Unitaire (par exemple, pour la classe `BankAccount`)
  - “Si je depose 5€ dans un compte contenant 5€, sa balance doit alors etre de 10€”
  - “Si je retire 5€ d’un compte contenant 5€, sa balance doit etre nulle”
  - “Si je retire 15€ d’un compte contenant 5€, une exception doit etre levee”
- Integration (par exemple, le back-end incluant une base de donnees et un serveur mail)
  - “Lorsque je depose de l’argent dans un compte, le resultat doit etre visible en base de donnees”
  - “Si je retire de l’argent au-dela des limites, une alerte est emise par e-mail”
  - “Si la base de donnees est hors service, les depots sont bloques”
- End-to-End tests (par exemple, l’application Android)
  - “Si je me connecte a l’application et depose une somme d’argent, alors je reçois une notification de confirmation”
  - “Si je rentre une devise incorrecte, alors un message d’erreur s’affiche”

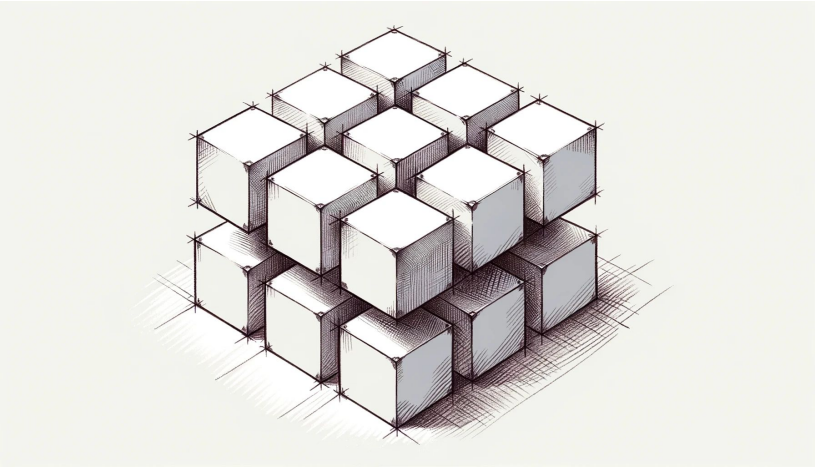
Bénéfices du test unitaire

- Les tests unitaires renforcent la **confiance** que l'on a en une partie du code
- Ils permettent de trouver des fautes **tôt** dans le developpement
- Ils sont **legers** et **rapides** et peuvent donc etre souvent lances (integration continue)
- Les suites de test deviennent *de facto* des tests de non-regression
- Cela rend les **evolutions** et **re-usinages** plus confortables
- Les tests permettent de symboliser l'**avancement** d'un projet, et de le **documenter**

Peut-on toujours isoler une unité ?

- Une unite a le plus souvent des **collaborateurs externes** (dependances)
- On souhaite trouver des fautes dans l'unite, non dans ses collaborateurs
- On cherche donc a **isoler** l'unite de ses collaborateurs
- Ce n'est pas si simple ! (couplage, injection de dependances, etc.)
- On le fait a l'aide de **doubles de test** (e.g., *dummies, fakes, stubs, mocks, spies)*
- Plus de details dans les prochaines sessions

```
class BankAccount {
  private DbConnector db;
  private Logger log;
}
```



Que tester ?

- Tout (sous-)programme est une fonction mappant *n* entrees a (*n*) sorties, e.g.:

$$\text{withdraw} : \text{balance } (\mathbb{R}) \times \text{amount } (\mathbb{R}+) \longrightarrow \text{new balance } (\mathbb{R})$$

- Un *vecteur de test* est un vecteur de valeurs que l'on soumet a l'unite sous test
  - <15, 5> : retirer 5€ d'un compte dont la balance est a 15€
  - <-5, 5> : retirer 5€ d'un compte dont la balance est a -5€

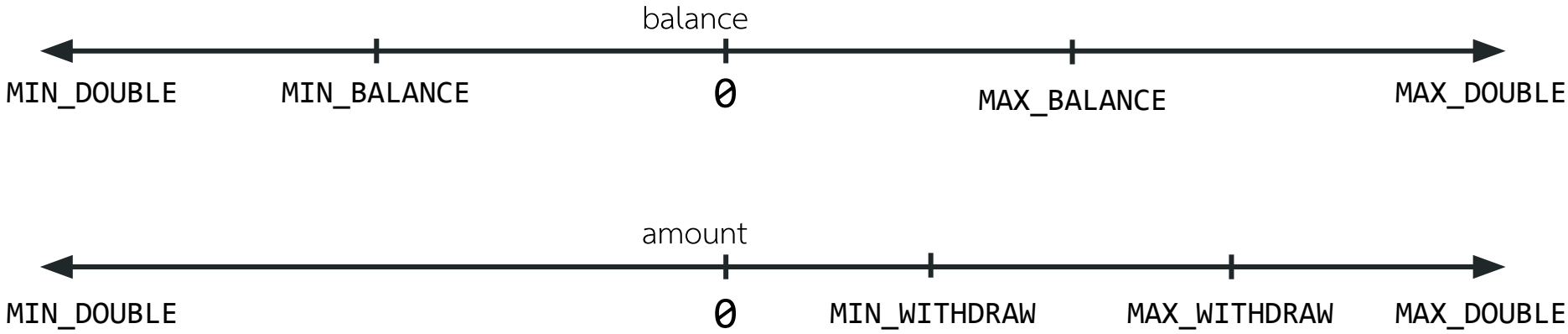
- Trouver des valeurs de test pour les classes *valides* (~= cas nominaux) et *invalides*
- Y associer les oracles appropries (valeur attendue, succes, echec, exception, etc.)

Que tester : partitions/classes d'équivalence

- Idee : on divise les valeurs d'entrees en *classes* qui doivent mettre en valeur le meme comportement (valide ou invalide)
- On conçoit les tests de sorte a choisir une valeur par classe
- Technique "boıte noire" qui s'applique a tous les niveaux de la pyramide des tests

Que tester : partitions/classes d'équivalence

- Idee : on divise les valeurs d'entrees en *classes* qui doivent mettre en valeur le meme comportement (valide ou invalide)
- On conçoit les tests de sorte a choisir une valeur par classe
- Technique "boite noire" qui s'applique a tous les niveaux de la pyramide des tests



Que tester : partitions/classes d'équivalence

- Idee : on divise les valeurs d'entrees en *classes* qui doivent mettre en valeur le meme comportement (valide ou invalide)
- On conçoit les tests de sorte a choisir une valeur par classe
- Technique "boite noire" qui s'applique a tous les niveaux de la pyramide des tests
- En supposant **une balance entre -300 et 10,000 et des retraits entre 5 et 1,000** :
- Vecteurs de test :

```
Balance : -500, -150, 500, 15000
Amount  : -500, 3,    500, 1500
```

```
<-500, -500> <-500, 3> <-500, 500> <-500, 1500>
<-150, -500> <-150, 3> <-150, 500> <-150, 1500>
  <500, -500> <500, 3> <500, 500> <500, 1500>
<15000, -500> <15000, 3> <15000, 500> <15000, 1500>
```

Que tester: analyse des valeurs limites

- L'implementation reflete les contraintes du domaine !
- Les fautes se glissent souvent aux *valeurs limites* des classes/partitions
  - Que se passe-t-il a balance = 0 ; a balance = 1 ; a amount = 0 ; a amount = -1 ? …

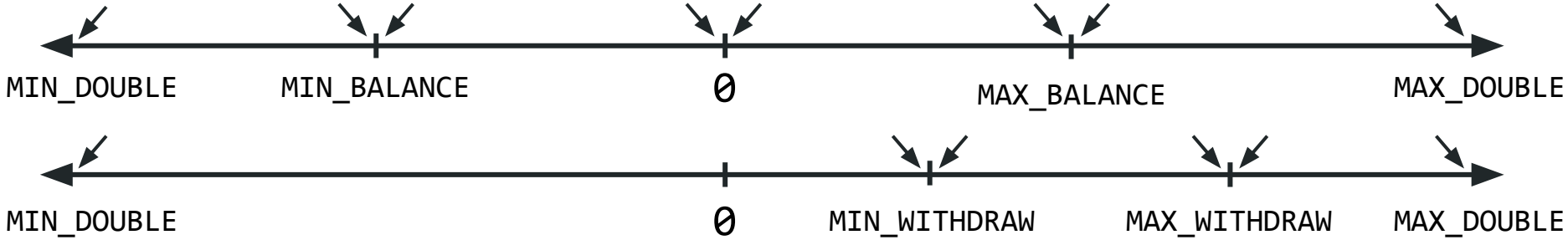
```
public void withdraw(double amount) {
  double current = this.getBalance();
  if (current ⩽ 0)
    throw new BalanceException("Can't withdraw");
  this.setBalance(current - amount);
}
```

Que tester: analyse des valeurs limites

- L'implementation reflete les contraintes du domaine !
- Les fautes se glissent souvent aux *valeurs limites* des classes/partitions
  - Que se passe-t-il a balance = 0 ; a balance = 1 ; a amount = 0 ; a amount = -1 ? …
-

```
public void withdraw(double amount) {
  double current = this.getBalance();
  if (current ⩽ 0)
    throw new BalanceException("Can't withdraw");
  this.setBalance(current - amount);
}
```

- On teste les valeurs limites (minimum et maximum) de chaque classe



Le test unitaire avec JUnit

JUnit

- La reference pour le test unitaire en Java
- Version courante >= 5.0 (2017)
- Initie par Kent Beck et Erich Gamma (lors d'un vol en avion ;)
- Proche des idees de l'eXtreme Programming (XP) et du Test-Driven Development (TDD)
- JUnit n'est cependant pas *limite* au test unitaire, et est souvent utilise pour d'autres niveaux
- Un ecosysteme tres riche (metriques, couverture, *build*, integration continue, etc.)
- On en trouve l'equivalent dans beaucoup de langages (SUnit, CppUnit, etc.)

Cas de test et suites de tests

- En JUnit, les tests d'une classe (unite) sont regroupes dans une classe de test
- Si la classe sous test est situee dans le package `pkg` (`src/main/java`), alors la classe de test est situee dans le package `pkg` (`src/test/java`)
- Chaque methode `@Test` definit un cas de test (`void`, non-`abstract`, no parameters)

```
class BankAccount {
  private double balance;
  public BankAccount(double initial)
  public void deposit(double amount)
  public void withdraw(double amount)
}
```

```
class BankAccountTest {
  @Test
  void withdraw_positive_amount() {
    BankAccount acc = new BankAccount(100);
    acc.withdraw(50);
    assertEquals(50, acc.getBalance());
  }

  @Test
  void withdraw_from_zero_should_fail() {
    BankAccount atZero = new BankAccount(0);
    assertThrows(IllegalWithdraw.class,
      () → atZero.withdraw(1));
  }
}
```

Assertions courantes

- Le package `org.junit.jupiter.api.Assertions.*` regorge de types d'assertions
- On doit parfois envelopper nos appels dans des *lambdas*

```
BankAccount legitAccount      = new BankAccount(5_000);
BankAccount fraudulentAccount = new BankAccount(5_000_000);

assertEquals(5_000, legitAccount.getBalance(), "The balance should be $5.000");
assertFalse(legitAccount.getBalance() > 10_000, "Too high");
assertNotSame(fraudulentAccount, legitAccount);
assertTimeout(Duration.ofSeconds(1), () → legitAccount.withdraw(5));
assertThrows(SecurityException.class, () → fraudulentAccount.withdraw(1));
fail("Should not be there!");
```

Test Fixtures

- Souvent, on reutilise des donnees et objets d'un test a l'autre (e.g., `legitAccount`)
- Les donnees et objets ne doivent pas etre impactes par les executions de test
  - Pas de garantie sur l'ordre d'execution des tests
  - Il faut re-initialiser les donnees pour eviter les effets de bord !
- On peut voir les *fixtures* comme le *contexte commun* a un ensemble de tests, stocke dans les attributs de la classe de test
- Trois phases : **initialisation**, execution, **nettoyage** (pour *chaque* test ou pour un *ensemble*)
- On annote des methodes speciales de la classe de test pour indiquer leur role
  - `@BeforeAll` : execute avant *tous* les tests d'une classe
  - `@BeforeEach`: execute avant *chaque* test d'une classe
  - `@AfterEach`: execute apres *chaque* test d'une classe
  - `@AfterAll`: execute apres *tous* les tests d'une classe
-

Test Fixtures

```
@BeforeAll
    @BeforeEach
        @Test 1
    @AfterEach
    @BeforeEach
        @Test 2
    @AfterEach
@AfterAll
```

```
class BankAccountTest {
  static private DbConnector db;
  private BankAccount fraudulentAccount;
  private BankAccount legitAccount;

  @BeforeAll
  static void setUpClass() {
    db = DbConnector.initialize("https://host:port");
  }

  @BeforeEach
  void setUpTest() {
    legitAccount      = new BankAccount(5_000);
    fraudulentAccount = new BankAccount(5_000_000);
  }

  @AfterAll
  static void tearDownClass() {
    db.close();
  }
}
```

Cas de test : quelques bonnes pratiques

- Un cas de test verifie *un* comportement
  - Idealement, un cas de test invoque *une* methode de la classe, pour *un* vecteur de test et *un* oracle
- Un cas de test est le plus concis possible
- Un cas de test suit une structure *AAA* : *Arrange* (setup), *Act* (exercise), *Assert* (verify)
- Chaque test doit etre independant des autres
- Les tests doivent clairement indiquer le scenario teste

```
@DisplayName("Verify that negative deposits are disallowed")
@Test
void deposit_negative_amount() { /* ... */ }
```

- Il est inutile (et impossible en JUnit) de tester les elements *prives* de vos classes
- Testez *toute* votre interface
  - Une exception cependant : inutile de tester vos *getters* ;)

Runners et sémantique d'exécution

- JUnit "decouvre" les suites et cas de test par *introspection*
- Identifie chaque classe de test, puis instancie le *runner* (harnais) approprie
- Le *runner* identifie les methodes `@Test` ainsi que les *fixtures*
- Puis les execute en suivant la semantique appropriee
- Trois issues possibles : le test *passe*, le test *echoue*, le test s'*execute anormalement*
- Affiche un message par defaut, ou le message personnalise
- JUnit regorge de points d'extension pour personnaliser tout point de son fonctionnement

```
org.opentest4j.AssertionFailedError: The balance should be $5.000 ⟹
Expected :5000.0
Actual   :4841.0
<Click to see difference>
```

EOF