Module 2

Clean Code & Analyse Statique

Génie Logiciel et Qualité — M1 MIAGE | 2h CM | 1h TD | 1h30 TP

Objectifs du module

- Comprendre les principes du **Clean Code** pour écrire du code lisible
- Maîtriser les **principes SOLID** pour une conception robuste
- Utiliser les outils d'**analyse statique** pour le contrôle qualité
- Écrire des **tests d'architecture** avec ArchUnit

"Good programmers write code that humans can understand." — Martin Fowler

1

Clean Code

Fondamentaux

Pourquoi le Clean Code ?

LE RATIO

10:1

Le code est lu 10× plus qu'il n'est écrit

Un code clean est :

✓ Lisible

✓ Simple

✓ Testable

✓ Expressif

Coûts du code "sale"

Noms cryptiques

+30% temps compréhension

Fonctions 200+ lignes

Bugs cachés, tests impossibles

Couplage fort

Cascade de changements

Duplication

Bugs non corrigés partout

Investir dans la lisibilité paie des dividendes à chaque lecture.

Règles de nommage

❌ **MAUVAIS**

```
// Qu'est-ce que "d" ?
int d;

// "list1" ne dit rien
List<int[]> list1;

// Trop vague
void process() { }

// Bruit sans sens
class DataManager { }
```

✓ **BON**

```
// Intention claire
int elapsedTimeInDays;

// Contenu explicite
List<Book> overdueBooks;

// Action précise
void calculateMonthlyFee() { }

// Responsabilité évidente
class PenaltyCalculator { }
```

Principe : Le nom doit révéler l'**intention**, pas l'implémentation.

Fonctions — Règles d'or

RÈGLE 1 : PETITES

✓ Idéal : 5-15 lignes
⚠ Acceptable : 15-30 lignes
✗ Problème : 30+ lignes

RÈGLE 2 : UNE SEULE CHOSE

Une fonction = une responsabilité

RÈGLE 3 : PEU D'ARGUMENTS

0-2 ✓ | 3 ⚠ | 4+ ✗

Exemple : Décomposition

```
// ❌ Fait trop de choses
void processOrder(Order o) {
  // validation...
  // calcul total...
  // réductions...
  // sauvegarde...
}
```

```
// ✓ Chaque fonction = 1 chose
void processOrder(Order o) {
  validateOrder(o);
  var total = calculateTotal(o);
  total = applyDiscounts(o, total);
  saveOrder(o, total);
}
```

2

Principes SOLID

Conception orientée objet robuste

SOLID — Vue d'ensemble

S — **Single Responsibility**

Une classe = une seule raison de changer

O — **Open/Closed**

Ouvert à l'extension, fermé à la modification

L — **Liskov Substitution**

Les sous-types substituables aux types de base

I — **Interface Segregation**

Interfaces spécifiques > interface générale

D — **Dependency Inversion**

Dépendre des abstractions, pas des implémentations

S Single Responsibility Principle

✗ GOD CLASS

```
class Employee {
  // Données
  String name; BigDecimal salary;
  // Calcul paie
  calculatePay() { ... }
  // Rapports
  generatePaySlip() { ... }
  // Persistance
  save() { ... }
  // Notification
  sendEmail() { ... }
}
```

4 acteurs différents touchent la même classe !

✓ RESPONSABILITÉS SÉPARÉES

```
class Employee { /* Données */ }

class PayCalculator {
  calculate(emp) { ... }
}

class PaySlipGenerator {
  generate(emp) { ... }
}

class EmployeeRepository {
  save(emp) { ... }
}
```

Chaque classe = 1 responsabilité

"Une classe ne devrait avoir qu'une seule raison de changer." — Robert C. Martin

O Open/Closed Principle

✗ SWITCH SUR TYPE

```
double calculatePenalty(Member m) {
  switch (m.getType()) {
    case STUDENT:
      return days * 0.25;
    case TEACHER:
      return 0;
    case EXTERNAL:
      return days * 0.50;
  }
}
```

Ajouter VIP = modifier cette classe

✓ POLYMORPHISME

```
interface PenaltyPolicy {
  Money calculate(int days);
}

class StudentPolicy implements... {
  return Money.of(days * 0.25);
}

// Nouveau type ? Nouvelle classe !
class VipPolicy implements... { }
```

Extension sans modification !

Ouvert à l'extension (nouveau comportement) • Fermé à la modification (code existant)

D Dependency Inversion Principle

✗ COUPLAGE FORT

```
class OrderService {
  // Dépendances en dur !
  private repo =
    new MySqlOrderRepo();
  private mailer =
    new SmtpEmailSender();
}
```

Impossible à tester sans MySQL/SMTP

✓ INJECTION

```
class OrderService {
  // Abstractions injectées
  private final OrderRepository repo;
  private final EmailSender mailer;

  OrderService(OrderRepository r,
               EmailSender m) { ... }
}
```

Mock/Fake injectable en test !

Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau.
Les deux doivent dépendre d'abstractions.

3

Analyse Statique

Automatiser le contrôle qualité

SonarQube — Plateforme de qualité

Types de problèmes détectés

Bug

Vulnérabilité

Code Smell

Métriques clés

Coverage
> 80%

Duplications
< 3%

Complexity
< 10/méthode

Tech Debt
< 5%

Quality Gate

✅ Passe si :

- Coverage > 80%
- Duplications < 3%
- 0 Blocker / 0 Critical
- Tech Debt Ratio < 5%

❌ Échoue sinon → Build bloqué

```
# Lancer l'analyse
mvn sonar:sonar -Dsonar.host.url=...
```

Écosystème d'outils

Checkstyle

Conventions de codage

- Indentation
- Nommage
- Imports
- Javadoc

PMD

Anti-patterns

- Code mort
- Complexité
- Empty catch
- Mauvaises pratiques

SpotBugs

Analyse bytecode

- Null pointer
- Ressources
- Concurrence
- Sérialisation

ArchUnit

Tests architecture

- Dépendances
- Conventions
- Annotations
- Couches

ArchUnit — Tests d'architecture

Écrire des tests JUnit pour vérifier les règles d'architecture

```
@AnalyzeClasses(packages = "com.bibliotech")
class ArchitectureTest {

  // Le domaine ne dépend pas de l'infrastructure
  @ArchTest
  static final ArchRule domain_independent =
    noClasses().that().resideInAPackage("..domain..")
    .should().dependOnClassesThat()
    .resideInAPackage("..infrastructure..");

  // Controllers annotés @RestController
  @ArchTest
  static final ArchRule controllers_annotated =
    classes().that().haveSimpleNameEndingWith("Controller")
    .should().beAnnotatedWith(RestController.class);
}
```

Les règles d'architecture deviennent des **tests automatisés** exécutés à chaque build

Récapitulatif

1. Clean Code

- Nommage — Révéler l'intention
- Fonctions — Petites, une seule chose
- Commentaires — Éviter si possible

2. Principes SOLID

S — Une responsabilité
O — Extension sans modification
L — Substitution
I — Interfaces spécifiques
D — Abstractions

3. Analyse Statique

- SonarQube
- Checkstyle
- PMD / SpotBugs
- ArchUnit

Quality Gate

Seuils de qualité automatisés dans la CI/CD

Prochaine étape : Module 3 — Refactoring