Module 3

Refactoring

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

Objectifs du module

- Comprendre la **définition** et les principes du refactoring
- Maîtriser le **catalogue des refactorings** essentiels
- Appliquer les **stratégies pour le code legacy**
- Utiliser les **outils et automatisation** de l'IDE

> *"Le refactoring améliore la structure interne sans changer le comportement externe." — M. Fowler*

1

Fondamentaux

Définition et principes du refactoring

Qu'est-ce que le refactoring ?

Modifier un système de telle manière que le comportement externe reste inchangé, mais que la structure interne soit améliorée.

✅ Ce que c'est

- Restructurer le code existant
- Améliorer lisibilité/maintenabilité
- Réduire la complexité
- Éliminer la duplication
- Préparer de nouvelles features

❌ Ce que ce n'est PAS

- Ajouter de nouvelles fonctionnalités
- Corriger des bugs
- Optimiser les performances
- Réécrire from scratch

Règle d'or : Le comportement observable ne change PAS. f(x) = y avant ET après.

Pourquoi et quand refactorer ?

Motivations

Lisibilité
processData() → calculateMonthlyRevenue()

Maintenabilité
1000 lignes → 5 classes cohérentes

Testabilité
Injection de dépendances

Extensibilité
Switch → Polymorphisme

⚠ Quand NE PAS refactorer

- ❌ Code qu'on ne touchera plus
- ❌ Deadline imminente
- ❌ Absence de tests
- ❌ Réécriture complète nécessaire

Règle des Trois

```
1ère fois → Faites-le simplement
2ème fois → Dupliquez (à contrecœur)
3ème fois → REFACTOREZ !
```

Moments opportuns

- Avant d'ajouter une feature
- Phase Refactor du TDD
- Pendant une code review
- Quand on corrige un bug

Prérequis absolu — Les tests

🚨 Pas de tests = Pas de refactoring sécurisé

Sans tests

```
Refactoring = Réorganisation + Prière 🙏
```

Avec tests

```
Refactoring = Réorganisation + Vérification
✅
```

Couverture minimum recommandée

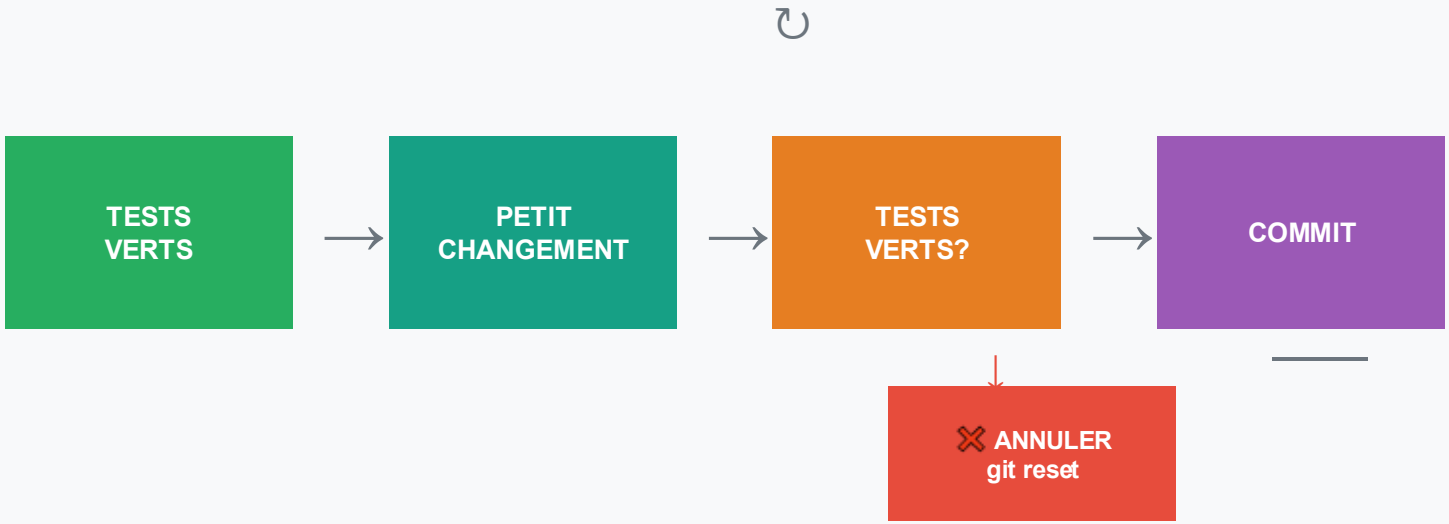
Renommage simple	Tests existants suffisants
Extract Method	Tests de la méthode parente
Extract Class	Tests complets de la classe
Changement d'architecture	Tests E2E + intégration

Si pas de tests ?

1. Écrire des tests de caractérisation
2. Seulement ensuite, refactorer

"Legacy code is code without tests."
— Michael Feathers

Le cycle du refactoring sécurisé



Règles d'or

1. Un seul refactoring à la fois
2. Tester après chaque changement
3. Commiter souvent
4. Jamais de changement fonctionnel pendant un refactoring

2

Catalogue des Refactorings

Les techniques essentielles

Vue d'ensemble du catalogue

Composing Methods	Extract/Inline Method	Méthodes claires
Moving Features	Move Method, Extract Class	Responsabilités
Organizing Data	Encapsulate Field	Données structurées
Simplifying Conditionals	Decompose, Polymorphism	Logique claire
Method Calls	Rename, Add Parameter	API intuitive

Raccourcis IDE essentiels (IntelliJ)

`Shift+F6`	Rename
`Ctrl+Alt+M`	Extract Method
`Ctrl+Alt+N`	Inline
`F6`	Move

Extract Method

Quand utiliser ?

- Code dupliqué
- Méthode > 20 lignes
- Commentaire = signe
- Niveaux mélangés

```
Ctrl + Alt + M
```

❌ **AVANT**

```
void printOwing() {
  printBanner();
  // Calculate outstanding
  double o = 0.0;
  for (Order ord : orders)
    o += ord.getAmount();
  // Print details
  println("Name: " + name);
  println("Owing: " + o);
}
```

✓ **APRÈS**

```
void printOwing() {
  printBanner();
  double o = calcOutstanding();
  printDetails(o);
}

private double calcOutstanding() {
  return orders.stream()
    .mapToDouble(Order::getAmount)
    .sum();
}
```

💡 Le nom doit révéler l'intention (QUOI, pas COMMENT). Trop de paramètres → Extract Class

Rename — Le refactoring le plus important !

"There are only two hard things: cache invalidation and naming things." — Phil Karlton

`Shift + F6` Renomme TOUTES les occurrences

Exemples de renommage

```
int d;              → int daysSinceModification;
void calc()         → void calculateMonthlyFee();
class Proc          → class PaymentProcessor;
List<int[]> list1 → List<Cell> flaggedCells;
String strName      → String customerName;
boolean flag        → boolean isOverdue;
```

Conventions de nommage

Variable locale	camelCase	`loan`
Champ	camelCase	`activeLoans`
Constante	UPPER_SNAKE	`MAX_LOAN`
Méthode	camelCase + verbe	`calculateFee()`
Classe	PascalCase	`LoanService`

📞 Test du téléphone : Pouvez-vous expliquer ce code AU TÉLÉPHONE sans voir l'écran ? Si non → améliorer le nommage.

Extract Class

Quand utiliser ?

- Trop de responsabilités (SRP)
- Champs qui vont ensemble
- Plusieurs centaines de lignes
- Nom avec "et"

✗ AVANT

```
class Person {
  String name;
  String streetAddress;
  String city;
  String zipCode;
  String areaCode;
  String phoneNumber;

  String getFullAddress()...
  String getFormattedPhone()...
}
```

✓ APRÈS

```
class Person {
  String name;
  Address address;
  PhoneNumber phone;
}

class Address {
  String street, city, zip;
  String getFormatted()...
}

class PhoneNumber {...}
```

Bénéfices

- Responsabilité unique
- Réutilisation
- Tests ciblés
- Value Objects

Replace Conditional with Polymorphism

Quand utiliser ?

- Switch/if sur un type ou catégorie
- Même switch à plusieurs endroits
- Violation Open/Closed

❌ Switch viole OCP

```
Money calculate(Loan loan) {
  switch (loan.getMember().getType()) {
    case STUDENT:
      return Money.of(days * 0.25);
    case TEACHER:
      return Money.ZERO;
    case EXTERNAL:
      return Money.of(days * 0.50);
  }
}
// Ajouter VIP = MODIFIER cette classe
```

✓ Polymorphisme (OCP respecté)

```
interface PenaltyPolicy {
  Money calculate(int days);
}
class StudentPolicy implements... {
  return Money.of(days * 0.25);
}
class TeacherPolicy implements... {
  return Money.ZERO;
}
// Ajouter VIP = AJOUTER une classe
```

Utiliser une Factory ou un Map<Type, Policy> pour créer les implémentations

Guard Clauses — Éliminer l'imbrication

Problème : Arrow code (imbrication profonde)

❌ **Imbrication**

```
double getPayAmount() {
  double result;
  if (isDead) {
    result = deadAmount();
  } else {
    if (isSeparated) {
      result = separatedAmount();
    } else {
      if (isRetired) {
        result = retiredAmount();
      } else {
        result = normalPayAmount();
      }
    }
  }
  return result;
}
```

✓ **Guard clauses (early return)**

```
double getPayAmount() {
  if (isDead) return deadAmount();
  if (isSeparated) return separatedAmount();
  if (isRetired) return retiredAmount();

  return normalPayAmount();
  // Happy path clairement visible
}
```

✓ Réduction de l'indentation ✓ Happy path visible ✓ Cas d'erreur explicites

3

Code Legacy

Stratégies pour le code existant

Définition du code legacy

"Legacy code is simply code without tests." — Michael Feathers

Mais aussi :

- Code dont l'auteur est parti
- Code qu'on a peur de modifier
- Documentation obsolète/inexistante
- Code critique mal compris

Le cercle vicieux

```
Code sans tests
    ↓
Peur de modifier
    ↓
Pas de refactoring
    ↓
Code plus complexe
    ↓
Encore plus dur à tester
    ↻ Retour au début
```

🎯 Objectif : Briser le cercle — ajouter des tests progressivement pour permettre le refactoring

Tests de caractérisation

Concept (M. Feathers)

Capture le comportement ACTUEL du code, même si ce comportement est incorrect. Préserver pendant le refactoring, corriger après.

Méthode

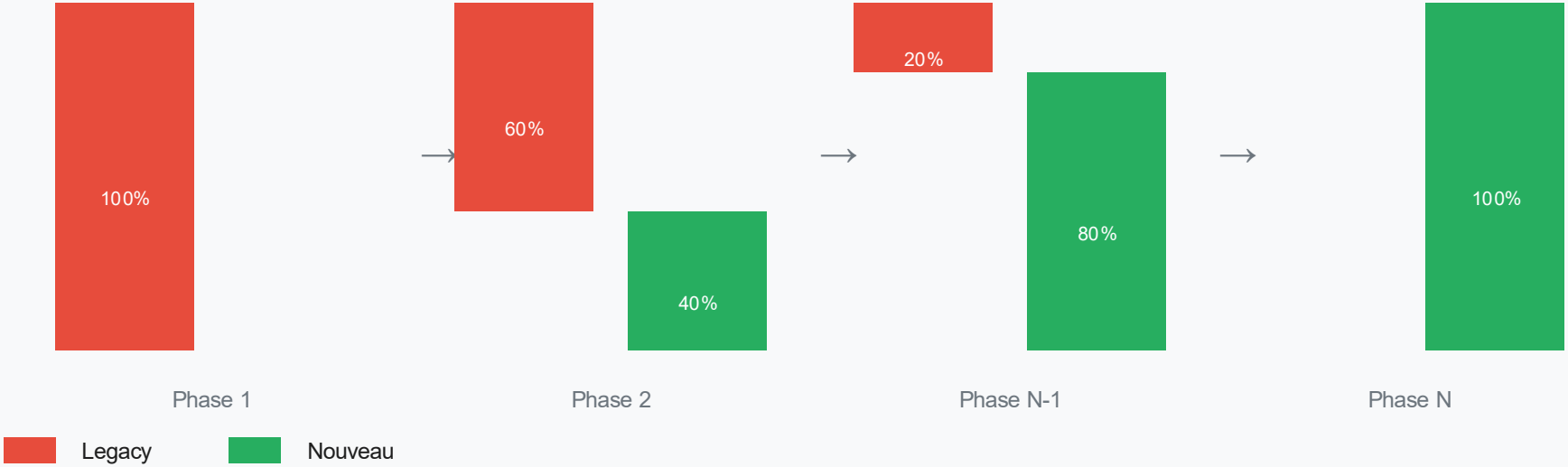
1. Identifier code à refactorer
2. Écrire test qui appelle ce code
3. Laisser échouer, noter la valeur
4. Copier dans l'assertion

```
@Test
void characterization_of_fee_calculation() {
    var calculator = new LegacyFeeCalculator();
    var loan = new Loan(...);  // Configuration réelle

    double result = calculator.calculate(loan);

    // Même si 42.37 n'est pas la valeur "correcte" attendue !
    assertThat(result).isEqualTo(42.37);
}
```

⚠ On capture le comportement ACTUEL, pas le comportement SOUHAITÉ. Correction des bugs = après refactoring.

Strangler Fig Pattern

Concept : Remplacer progressivement un système legacy fonctionnalité par fonctionnalité, sans big bang.



✓ Risque minimal (rollback via feature flag) ✓ Valeur continue ✓ Comparaison possible ✓ Pas de big bang

Sprout Method / Sprout Class

Concept (M. Feathers) : Créer une nouvelle méthode/classe TESTÉE qu'on appelle depuis le legacy non testé.

```
// Code legacy non testé
public void processOrder(Order order) {
    // ... 200 lignes de code legacy ...

    // NOUVELLE FONCTIONNALITÉ : faire "pousser" une méthode
    validateShippingAddress(order.getAddress());

    // ... suite du code legacy ...
}

// Nouvelle méthode testable indépendamment
boolean validateShippingAddress(Address address) {
    // Code propre, testé
    return address != null
        && isValidZipCode(address.getZipCode());
}
```

Avantages

✓ Pas de modification du legacy

✓ Nouveau code testé dès le départ

✓ Îlot de qualité qui grandit

✓ Base pour futur refactoring

4

Outils & Automatisation

Support IDE et métriques

Support IDE — IntelliJ IDEA

Refactorings automatiques

`Shift + F6`	**Rename**	Partout (fichiers, refs, Javadoc)
`Ctrl + Alt + M`	**Extract Method**	Détecte paramètres/retours
`Ctrl + Alt + V`	**Extract Variable**	Crée variable locale
`Ctrl + Alt + N`	**Inline**	Supprime indirection
`Ctrl + F6`	**Change Signature**	Propage les changements
`F6`	**Move**	Déplace classe/méthode
`Alt + Delete`	**Safe Delete**	Vérifie les usages

Alt + Enter

Actions contextuelles :

- Sur paramètre → to field
- Sur condition → invert if
- Sur switch → enhanced
- Sur boucle → Stream API

Analyse : Ctrl+Alt+Shift+I (Run Inspection) | Analyze > Inspect Code | Analyze > Data Flow

Métriques de qualité

Métrique	Avant ❌	Après ✓	Outil
Complexité cyclomatique	> 10	< 5	SonarQube
Lignes/méthode	> 50	< 20	Checkstyle
Paramètres/méthode	> 4	≤ 3	PMD
Couverture tests	< 50%	> 80%	JaCoCo
Duplication	> 5%	< 3%	SonarQube

Complexité cyclomatique — Seuils

1-5 : Simple	6-10 : Modéré	11-20 : Complexe	21+ : Refactoring urgent!

Git et refactoring

✗ Mauvais

```
git commit -m "Refactoring
et ajout de la feature X"
```

✓ Bon (atomique)

```
"Refactor: Extract PenaltyCalc"
"Refactor: Rename calculateFee"
"Feature: Add progressive penalty"
```

Préfixes de commit

`Refactor:`	Restructuration	`Rename:`	Renommage	`Extract:`	Extraction
`Move:`	Déplacement	`Feature:`	Nouvelle fonctionnalité	`Fix:`	Correction de bug

Workflow sécurisé

```
git checkout -b refactor/extract-penalty-calc  # Branche dédiée
mvn test                                       # Tests verts ?
# ... UN refactoring ...
mvn test                                       # Toujours verts ?
git commit -m "Refactor: Extract..."          # Commit atomique
```

Récapitulatif

1. Fondamentaux

- Définition précise
- Tests = prérequis
- Petits pas, verts

2. Catalogue

- Extract Method
- Rename (le + important)
- Replace with Poly.

3. Legacy

- Tests caractérisation
- Strangler Fig
- Sprout Method

4. Outils

- Raccourcis IDE
- Métriques qualité
- Git atomique

🎯 **Règle d'or : Le comportement observable ne change JAMAIS pendant un refactoring**

Prochaine étape : Module 4 — Design Patterns