UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Jour 4

Refactorer une application legacy

Refactorer une Application Legacy

Une application legacy est un logiciel hérité qui peut être difficile à maintenir ou à faire évoluer en raison de sa complexité, de son ancienneté ou de son manque de tests. Refactorer ce type d'application est essentiel pour améliorer sa qualité et faciliter son évolution.

1. Tester

Avant de commencer le refactoring, il est crucial d'avoir une suite de tests automatisés pour s'assurer que les modifications n'introduisent pas de régressions. Si l'application n'a pas de tests, il faut en écrire pour les parties critiques du code.

Refactorer une Application Legacy

2. Refactorer

Appliquez des techniques de refactoring pour améliorer la structure du code sans modifier son comportement observable. Le refactoring doit être effectué par petites étapes, en vérifiant après chaque modification que les tests passent toujours.

3. Apporter de la Valeur

Le refactoring doit faciliter l'ajout de nouvelles fonctionnalités, améliorer les performances ou rendre le code plus maintenable. Il doit toujours avoir un objectif clair qui apporte de la valeur à l'application.

Refactorer une Application Legacy

Exemple de Refactoring

Code Avant Refactoring :

```
public class ClientService {
    public void processClientData(Client client) {
        // Validation des données client
        if (client.getName() != null && client.getAge() > 0) {
            // Calcul du score de crédit
            int creditScore = calculateCreditScore(client);
            // Envoi d'un email de bienvenue
            EmailService emailService = new EmailService();
            emailService.sendWelcomeEmail(client.getEmail());
            // Mise à jour de la base de données
            Database database = new Database();
            database.updateClientRecord(client);
        } else {
            throw new IllegalArgumentException("Données client invalides");
        }
    }
}
```

Refactorer une Application Legacy

Problèmes Identifiés :

- **Long Method** : La méthode `processClientData` effectue plusieurs tâches distinctes.
- **Violation du Principe de Responsabilité Unique** : La classe gère la validation, le calcul du score, l'envoi d'emails et la mise à jour de la base de données.
- **Couplage Fort** : La classe crée directement des instances d'autres classes (`EmailService`, `Database`), ce qui rend difficile le test unitaire.

Refactorer une Application Legacy

Code Après Refactoring :

```
public class ClientService {
    private Validator validator;
    private CreditScoreCalculator creditScoreCalculator;
    private EmailService emailService;
    private Database database;
    public ClientService(Validator validator, CreditScoreCalculator creditScoreCalculator,
                         EmailService emailService, Database database) {
        this.validator = validator;
        this.creditScoreCalculator = creditScoreCalculator;
        this.emailService = emailService;
        this.database = database;
    }
    public void processClientData(Client client) {
        validateClient(client);
        int creditScore = creditScoreCalculator.calculate(client);
        emailService.sendWelcomeEmail(client.getEmail());
        database.updateClientRecord(client);
    }
    private void validateClient(Client client) {
        if (!validator.isValid(client)) {
            throw new IllegalArgumentException("Données client invalides");
        }
    }
}
```

Refactorer une Application Legacy

Améliorations Apportées :

- **Extraction de Méthodes** : Les différentes responsabilités sont maintenant dans des méthodes ou classes séparées.
- **Injection de Dépendances** : Les dépendances sont injectées via le constructeur, facilitant ainsi les tests unitaires.
- **Responsabilité Unique** : Chaque classe a une responsabilité clairement définie.

Savoir ce qu’on Appelle un Code Smell

Un code smell est un indicateur de problèmes potentiels dans le code source. Ce n'est pas nécessairement un bug, mais plutôt un symptôme de mauvaise conception ou de mauvaise implémentation qui peut rendre le code difficile à maintenir ou à faire évoluer.

Il existe plusieurs types de code smells, classés en différentes catégories. Nous allons détailler les principaux, avec des exemples concrets pour chacun.

Bloaters

Les **bloaters** sont des éléments du code qui ont "gonflé" au fil du temps, devenant trop grands ou complexes.

Savoir ce qu'on Appelle un Code Smell

Long Method (Méthode Longue)

Une méthode qui est trop longue et qui fait trop de choses, ce qui la rend difficile à comprendre et à maintenir.

Exemple de Code Avant Refactoring :

```java
public void generateReport(List<Transaction> transactions) {
    double total = 0;
    for (Transaction t : transactions) {
        // Calcul du total
        total += t.getAmount();
        // Affichage des détails de la transaction
        System.out.println("Transaction ID: " + t.getId());
        System.out.println("Amount: " + t.getAmount());
        System.out.println("Date: " + t.getDate());
        // Vérification de fraudes potentielles
        if (t.isSuspicious()) {
            alertFraud(t);
        }
    }
    // Affichage du total
    System.out.println("Total Amount: " + total);
}
```

Savoir ce qu'on Appelle un Code Smell

Problèmes Identifiés :

- La méthode effectue plusieurs tâches : calcul du total, affichage des détails, vérification de fraudes, etc.
- Difficile à tester et à maintenir.

Savoir ce qu'on Appelle un Code Smell

Exemple de Code Après Refactoring :

```
public void generateReport(List<Transaction> transactions) {
    double total = calculateTotal(transactions);
    displayTransactions(transactions);
    System.out.println("Total Amount: " + total);
}
private double calculateTotal(List<Transaction> transactions) {
    return transactions.stream()
            .mapToDouble(Transaction::getAmount)
            .sum();
}
private void displayTransactions(List<Transaction> transactions) {
    for (Transaction t : transactions) {
        displayTransactionDetails(t);
        checkForFraud(t);
    }
}
private void displayTransactionDetails(Transaction t) {
    System.out.println("Transaction ID: " + t.getId());
    System.out.println("Amount: " + t.getAmount());
    System.out.println("Date: " + t.getDate());
}
private void checkForFraud(Transaction t) {
    if (t.isSuspicious()) {
        alertFraud(t);
    }
}
```

Savoir ce qu'on Appelle un Code Smell

Améliorations Apportées :

- La méthode `generateReport` est maintenant plus concise et lisible.
- Les différentes tâches sont déléguées à des méthodes spécifiques.

Savoir ce qu'on Appelle un Code Smell

Large Class (Classe Trop Grande)

Une classe qui accumule trop de responsabilités, ce qui la rend complexe et difficile à maintenir.

Exemple de Code Avant Refactoring :

```
public class UserAccount {
    private String username;
    private String password;
    private List<Order> orders;
    private List<Message> messages;
    private Settings settings;

    public void addOrder(Order order) { /* ... */ }
    public void sendMessage(Message message) { /* ... */ }
    public void updateSettings(Settings settings) { /* ... */ }
    public void resetPassword() { /* ... */ }
    // Beaucoup d'autres méthodes
}
```

Savoir ce qu'on Appelle un Code Smell

Problèmes Identifiés :

- La classe gère les commandes, les messages, les paramètres, etc.
- Violation du principe de responsabilité unique.

Savoir ce qu'on Appelle un Code Smell

Exemple de Code Après Refactoring :

- **Création de Classes Spécifiques :**

```
public class UserAccount {
    private String username;
    private String password;
    private Settings settings;

    public void resetPassword() { /* ... */ }
    public void updateSettings(Settings settings) { /* ... */ }
}

public class OrderService {
    private List<Order> orders;

    public void addOrder(Order order) { /* ... */ }
    // Autres méthodes liées aux commandes
}

public class MessagingService {
    private List<Message> messages;

    public void sendMessage(Message message) { /* ... */ }
    // Autres méthodes liées aux messages
}
```

Savoir ce qu'on Appelle un Code Smell

Améliorations Apportées :

- Chaque classe a une responsabilité claire.
- Le code est plus modulaire et plus facile à maintenir.

Savoir ce qu'on Appelle un Code Smell

Object-Oriented Abusers: Cette catégorie concerne les abus ou les mauvais usages des principes de la programmation orientée objet.

Switch Statements (Utilisation Excessive de Switch ou If)

L'utilisation répétée de structures conditionnelles pour contrôler le flux du programme en fonction du type ou de l'état d'un objet.

Exemple de Code Avant Refactoring :

```
public double calculateDiscount(Product product) {
    if (product.getType() == ProductType.ELECTRONICS) {
        return product.getPrice() * 0.1;
    } else if (product.getType() == ProductType.CLOTHING) {
        return product.getPrice() * 0.2;
    } else if (product.getType() == ProductType.FOOD) {
        return product.getPrice() * 0.05;
    } else {
        return 0;
    }
}
```

Savoir ce qu'on Appelle un Code Smell

Problèmes Identifiés :

- Difficile à maintenir lorsque de nouveaux types de produits sont ajoutés.
- Violation du principe ouvert/fermé.

Savoir ce qu'on Appelle un Code Smell

Exemple de Code Après Refactoring :

- **Utilisation du Polymorphisme :**

```
public abstract class Product {
    protected double price;
    public abstract double calculateDiscount();
    public double getPrice() {
        return price;
    }
}

public class Electronics extends Product {
    public double calculateDiscount() {
        return price * 0.1;
    }
}

public class Clothing extends Product {
    public double calculateDiscount() {
        return price * 0.2;
    }
}

public class Food extends Product {
    public double calculateDiscount() {
        return price * 0.05;
    }
}
```

Savoir ce qu'on Appelle un Code Smell

Améliorations Apportées :

- Ajout de nouveaux types de produits sans modifier le code existant.
- Le calcul de la remise est délégué aux sous-classes.

Savoir ce qu’on Appelle un Code Smell

Temporary Field (Champ Temporaire)

Description :

Des attributs d'une classe qui ne sont initialisés que dans certaines circonstances, ce qui peut rendre le code confus.

Savoir ce qu'on Appelle un Code Smell

Exemple de Code Avant Refactoring :

```
public class ReportGenerator {
    private String title;
    private String content;
    private String footer;
    private Connection databaseConnection; // Utilisé uniquement pour certains rapports

    public ReportGenerator(String title, String content, String footer) {
        this.title = title;
        this.content = content;
        this.footer = footer;
    }

    public void generate() {
        // Génération du rapport
        if (databaseConnection != null) {
            // Génération spécifique avec accès à la base de données
        }
    }
}
```

Savoir ce qu’on Appelle un Code Smell

Problèmes Identifiés :

- L'attribut `databaseConnection` n'est pas toujours pertinent.
- Rend la classe plus complexe qu'elle ne devrait l'être.

Savoir ce qu'on Appelle un Code Smell

Exemple de Code Après Refactoring :

- **Extraction de Sous-Classes ou Utilisation de Patrons de Conception Appropriés :**

```
public class ReportGenerator {
    protected String title;
    protected String content;
    protected String footer;

    public ReportGenerator(String title, String content, String footer) {
        this.title = title;
        this.content = content;
        this.footer = footer;
    }

    public void generate() {
        // Génération du rapport standard
    }
}

public class DatabaseReportGenerator extends ReportGenerator {
    private Connection databaseConnection;

    public DatabaseReportGenerator(String title, String content, String footer, Connection databaseConnection) {
        super(title, content, footer);
        this.databaseConnection = databaseConnection;
    }

    @Override
    public void generate() {
        super.generate();
        // Génération spécifique avec accès à la base de données
    }
}
```

Savoir ce qu'on Appelle un Code Smell

Améliorations Apportées :

- Séparation des responsabilités.
- Le champ temporaire est déplacé vers une sous-classe où il est toujours pertinent.

Savoir ce qu'on Appelle un Code Smell

Change Preventers

Les **change preventers** sont des code smells qui rendent le code difficile à modifier ou à étendre.

Divergent Change (Changement Divergent)

Une classe qui doit être modifiée pour différentes raisons, à chaque fois qu'un type particulier de changement est nécessaire.

Exemple de Code Avant Refactoring :

```
public class NotificationService {
    public void sendEmailNotification(User user, String message) { /* ... */ }
    public void sendSMSNotification(User user, String message) { /* ... */ }
    public void sendPushNotification(User user, String message) { /* ... */ }
    // À chaque nouveau type de notification, la classe doit être modifiée.
}
```

Savoir ce qu'on Appelle un Code Smell

Exemple de Code Après Refactoring :

- **Utilisation du Patrons Stratégie ou Commande :**

```
public interface NotificationStrategy {
    void send(User user, String message);
}

public class EmailNotification implements NotificationStrategy {
    public void send(User user, String message) { /* ... */ }
}

public class SMSNotification implements NotificationStrategy {
    public void send(User user, String message) { /* ... */ }
}

public class PushNotification implements NotificationStrategy {
    public void send(User user, String message) { /* ... */ }
}

public class NotificationService {
    private NotificationStrategy strategy;

    public NotificationService(NotificationStrategy strategy) {
        this.strategy = strategy;
    }

    public void sendNotification(User user, String message) {
        strategy.send(user, message);
    }
}
```

Savoir ce qu’on Appelle un Code Smell

Améliorations Apportées :

- Ouvert pour extension, fermé pour modification.
- Facilite l'ajout de nouveaux types de notifications sans modifier le code existant.

Savoir ce qu'on Appelle un Code Smell

Shotgun Surgery Un changement mineur nécessite de modifier de nombreuses classes ou méthodes disséminées dans le code.

Exemple de Code Avant Refactoring :

- Supposons que le format de la date doit changer dans l'application, et que le format est utilisé directement dans plusieurs classes.

```
public class Order {
    public String getFormattedDate() {
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
        return sdf.format(orderDate);
    }
}

public class Invoice {
    public String getFormattedDate() {
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
        return sdf.format(invoiceDate);
    }
}

// Plusieurs autres classes avec le même code
```

Savoir ce qu'on Appelle un Code Smell

Problèmes Identifiés :

- Duplications de code.
- Difficulté à effectuer des changements globaux.

Exemple de Code Après Refactoring :

- **Centralisation du Code Commun :**

```
public class DateUtil {
    private static final SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");

    public static String formatDate(Date date) {
        return sdf.format(date);
    }
}

public class Order {
    public String getFormattedDate() {
        return DateUtil.formatDate(orderDate);
    }
}

public class Invoice {
    public String getFormattedDate() {
        return DateUtil.formatDate(invoiceDate);
    }
}
```

Savoir ce qu’on Appelle un Code Smell

Améliorations Apportées :

- Le format de la date est centralisé.
- Un changement du format n'affecte qu'une seule classe.

Savoir Refactorer les Principaux Code Smells

1. **Identifier le Code Smell :** Utiliser des outils d'analyse statique ou faire une revue de code pour repérer les problèmes.
2. **Comprendre le Code :** Avant de modifier, assurez-vous de bien comprendre le fonctionnement du code.
3. **Écrire des Tests :** Si des tests n'existent pas, en écrire pour couvrir les fonctionnalités existantes.
4. **Appliquer le Refactoring :** Utiliser les techniques appropriées pour améliorer le code.
5. **Vérifier les Tests :** S'assurer que les tests passent toujours après les modifications.

La Loi de Demeter

La Loi de Demeter, également connue sous le nom de "Principe du moindre savoir", stipule qu'une méthode d'un objet ne devrait interagir qu'avec :

- Ses propres méthodes.
- Ses attributs directs.
- Les objets créés par la méthode.
- Les paramètres de la méthode.

La Loi de Demeter

Violation de la Loi de Demeter :

Exemple de Code Avant Refactoring :

```
public class OrderService {
    public double getCustomerBalance(Order order) {
        return order.getCustomer().getAccount().getBalance();
    }
}
```

La Loi de Demeter

Problèmes Identifiés :

- L'accès en chaîne aux objets (`order.getCustomer().getAccount().getBalance()`) crée un couplage fort entre les classes.
- Si la structure interne change, le code doit être modifié.

La Loi de Demeter

Exemple de Code Après Refactoring :

- **Encapsulation des Détails Internes :**

```
public class Customer {
    private Account account;

    public double getBalance() {
        return account.getBalance();
    }
}

public class Order {
    private Customer customer;

    public double getCustomerBalance() {
        return customer.getBalance();
    }
}

public class OrderService {
    public double getCustomerBalance(Order order) {
        return order.getCustomerBalance();
    }
}
```

La Loi de Demeter

Améliorations Apportées :

- Réduction du couplage entre les classes.
- Les détails internes sont encapsulés, ce qui facilite les modifications futures.

La Loi de Demeter

Autre Exemple :

Avant Refactoring :

```
public class Car {
    private Engine engine;

    public void start() {
        engine.getFuelInjector().injectFuel();
        engine.getSparkPlug().ignite();
    }
}
```

La Loi de Demeter

Après Refactoring :

```
public class Engine {
    private FuelInjector fuelInjector;
    private SparkPlug sparkPlug;

    public void start() {
        fuelInjector.injectFuel();
        sparkPlug.ignite();
    }
}

public class Car {
    private Engine engine;

    public void start() {
        engine.start();
    }
}
```

La Loi de Demeter

Améliorations Apportées :

- La classe `Car` ne connaît plus les détails internes de `Engine`.
- Respect de la Loi de Demeter.

Avantages du Respect de la Loi de Demeter :

- **Faible Couplage :** Les classes sont moins dépendantes des structures internes des autres classes.
- **Facilité de Maintenance :** Les modifications internes d'une classe n'affectent pas les classes qui l'utilisent.
- **Meilleure Lisibilité :** Le code est plus clair et plus facile à comprendre.