UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

JAVA CLEAN CODE Partie 2

Métriques

- Définition et importance des métriques de qualimétrie.
- Présentation de certaines métriques de qualimétrie clés.

Définition et importance des métriques de qualimétrie

- **Définition** : Les métriques de qualimétrie sont des indicateurs quantitatifs utilisés pour mesurer la qualité d'un produit ou d'un service, dans notre cas, la qualité du code.
- **Importance** : Les métriques de qualimétrie sont essentielles pour identifier les zones du code qui nécessitent des améliorations, pour garantir la maintenabilité, la fiabilité, la testabilité, et l'efficacité du code.

Définition et importance des métriques de qualimétrie

Demo : Imaginez que nous essayons de comprendre la complexité d'une méthode spécifique dans notre code.

Présentation de certaines métriques de qualimétrie clés

- **Complexité cyclomatique** : Comme mentionné précédemment, cette métrique mesure le nombre de chemins d'exécution à travers le code. Une valeur élevée peut indiquer que le code est trop complexe et difficile à maintenir ou à tester.
- **LOC (Lines of Code)** : Mesure le nombre de lignes dans votre code. C'est une métrique simple, mais elle peut donner une idée de la taille du projet.
- **Couplage** : Il existe plusieurs métriques de couplage, mais en général, elles mesurent à quel point les classes dépendent les unes des autres. Un couplage élevé peut rendre le code difficile à modifier et à tester.
- **Cohésion** : Mesure à quel point les responsabilités d'une classe sont liées. Une classe doit avoir une seule responsabilité, et donc une cohésion élevée.
- **Profondeur de l'héritage** : Mesure le nombre de niveaux de la hiérarchie d'héritage d'une classe. Une profondeur élevée peut rendre le code plus difficile à comprendre et à maintenir.

Exercice Complexité cyclomatique

- Supposons que vous développiez une application de gestion des tâches où vous avez une classe TaskService qui contient une méthode calculatePriority() pour calculer la priorité d'une tâche en fonction de certains critères. Cependant, cette méthode a une complexité cyclomatique élevée, ce qui peut rendre le code difficile à comprendre et à maintenir.

1. Analysez la méthode calculatePriority() pour identifier les points où la complexité cyclomatique est élevée.
2. Réduisez la complexité cyclomatique en refactorisant le code pour améliorer sa lisibilité et sa maintenabilité.

Exercice LOC

- On souhaite réduire la loc de méthode createOrder du code suivant, proposez une nouvelle méthode :

```
@RestController
public class OrderController {
    @Autowired
    private OrderRepository orderRepository;

    @Autowired
    private CustomerRepository customerRepository;

    @PostMapping("/orders")
    public ResponseEntity<Order> createOrder(@RequestBody Order order, @RequestParam Long customerId) {
        Optional<Customer> customerOptional = customerRepository.findById(customerId);
        if (customerOptional.isPresent()) {
            Customer customer = customerOptional.get();
            order.setCustomer(customer);
            order.setDate(new Date());
            order.setStatus("CREATED");
            Order savedOrder = orderRepository.save(order);
            return ResponseEntity.ok(savedOrder);
        } else {
            return ResponseEntity.notFound().build();
        }
    }
}
```

Exercice Couplage

- Prenons l'exemple d'un contrôleur BookController qui utilise un service BookService. Le service lui-même utilise un dépôt BookRepository.

```
@RestController
public class BookController {
    @Autowired
    private BookService bookService;

    @GetMapping("/books")
    public List<Book> getAllBooks() {
        return bookService.findAllBooks();
    }
}
@Service
public class BookService {
    @Autowired
    private BookRepository bookRepository;

    public List<Book> findAllBooks() {
        return bookRepository.findAll();
    }
}
public interface BookRepository extends JpaRepository<Book, Long> {}
```

- Refactorisez ce code pour réduire le couplage

Exercice Cohésion

- Supposons que vous ayez une classe CustomerService qui est responsable de la gestion des clients dans votre application, mais aussi du calcul de certains indicateurs financiers

```
@Service
public class CustomerService {
    @Autowired
    private CustomerRepository customerRepository;

    public Customer createCustomer(Customer customer) {
        // Logique de création de client
        return customerRepository.save(customer);
    }

    public double calculateAverageRevenue(List<Customer> customers) {
        // Logique pour calculer le revenu moyen à partir d'une liste de clients
    }

    public double calculateProfitMargin(List<Customer> customers) {
        // Logique pour calculer la marge bénéficiaire à partir d'une liste de clients
    }
}
```

- Refactorisez le code ci-dessus pour améliorer sa cohésion