Rapport Projet Java - CLEAN CODE & PRINCIPES SOLID

> 💡 Quel est le degré de conformité du code de votre projet aux principes de
> l'approche Clean Code ? En particulier sur les deux volets suivants : Nommage
> et Gestion des erreurs.
> Donnez un maximum d'exemples -pertinents- extraits de votre code

Le projet respecte globalement les principes de nommage clair et descriptif. Par exemple :

```
@Controller
@RequestMapping("/utilisateurs")
@Tag(name = "UtilisateurController", description = "Gestion
des utilisateurs")
public class UtilisateurController {
    private static final Logger logger = LoggerFactory.getL
ogger(UtilisateurController.class);

    @Autowired
    private UtilisateurService utilisateurService;
    // ...
}
```

- **Classes et Interfaces** : Les noms tels que `UtilisateurController`, `FeedbackService` sont explicites et reflètent leur responsabilité.
- **Méthodes** : Des noms comme `recupererTousLesUtilisateurs()`, `ajouterUtilisateur()` décrivent clairement l'action effectuée.

- **Variables** : Utilisation de noms significatifs comme `outilUuid`, `feedbackService` facilite la compréhension du code.

Gestion des erreurs

Le code implémente une gestion des erreurs structurée en utilisant des exceptions personnalisées et des validations :

```
@PostMapping("/api")
@ResponseBody
public ResponseEntity<Utilisateur> ajouterUtilisateur(@Requ
estBody Utilisateur utilisateur) {
    Utilisateur savedUtilisateur = utilisateurService.ajout
erUtilisateur(utilisateur);
    return ResponseEntity.status(HttpStatus.CREATED).body(s
avedUtilisateur);
}
```

- **Exceptions Personnalisées** : Utilisation de classes comme `OutilIntrouvableException`, `FeedbackQuotaExceededException` pour gérer des cas spécifiques.

```
if (!outilService.recupererOutilParId(id).isPresent()) {
    throw new OutilIntrouvableException("Outil avec l'ID "
+ id + " introuvable.");
}
```

- **Validation des Entrées** : Vérifications avant l'exécution des opérations, par exemple :

```
if (login == null || utilisateurId == null) {
    throw new UtilisateurNonTrouveException("Vous devez êtr
e connecté pour ajouter un feedback.");
}
```

- **Gestion des Erreurs dans les Scripts** : Utilisation de messages d'erreur clairs et exit codes dans mvnw.

```
die "distributionUrl is not valid, must match *-bin.zip or
maven-mvnd-*.zip, but found '${distributionUrl-}'"
```

1. Pour chacun des principes SOLID, indiquez si vous en avez tenu compte.
   Si oui, indiquez où, dans le code de votre projet, ce principe est respecté et
   comment.
   Sinon, pourquoi.
   En effet, dépendamment du langage et des stacks techniques que vous avez
   choisis dans le cadre de votre projet, votre réponse peut comporter une
   remise en question des principes SOLID : Sont-ils appropriés dans le contexte
   de votre projet ? Sont-ils pertinents ?
   La qualité et la cohérence de vos arguments ainsi que votre précision seront
   très appréciées.

Chaque classe dans le projet a une responsabilité bien définie. Par exemple, les contrôleurs comme `UtilisateurController`, `FeedbackController`, et `OutilController` sont responsables de la gestion des requêtes HTTP spécifiques à leurs domaines respectifs. De même, les services tels que `UtilisateurService`, `FeedbackService`, et `OutilService` encapsulent la logique métier.

```
@Controller
@RequestMapping("/utilisateurs")
@Tag(name = "UtilisateurController", description = "Gestion
des utilisateurs")
public class UtilisateurController {
    @Autowired
```

```
    private UtilisateurService utilisateurService;
    // ...
}
```

Le projet permet l'extension des fonctionnalités sans modifier le code existant grâce à l'utilisation de Spring Boot et de l'injection de dépendances. Par exemple, l'ajout de nouvelles opérations dans les contrôleurs ne nécessite pas de modification des classes de service existantes.

Dans le contexte actuel du projet, il n'y a pas d'héritage complexe ou de hiérarchies de classes qui nécessiteraient une vérification approfondie de ce principe. Les classes peuvent principalement utiliser la composition via l'injection de dépendances plutôt que l'héritage.

Principe de Ségrégation des Interfaces (Interface Segregation Principle)

Le projet utilise des interfaces spécifiques pour chaque service, ce qui évite de surcharger les interfaces avec des méthodes non pertinentes. Par exemple, chaque service (`UtilisateurService`, `FeedbackService`, `OutilService`) expose uniquement les méthodes nécessaires à son domaine.

```
@Autowired
private FeedbackService feedbackService;
```

Le projet suit ce principe en s'appuyant sur l'inversion de contrôle fournie par Spring Boot. Les contrôleurs dépendent des abstractions (interfaces de service) plutôt que des implémentations concrètes, facilitant ainsi le test et la maintenance.

```
@Autowired
private OutilService outilService;
```

Pertinence et Adaptation

Les principes SOLID sont indispensables dans notre projet Java en utilisant Spring Boot. Ils favorisent une architecture modulaire, maintenable et évolutive.

L'utilisation des contrôleurs et des services distincts, ainsi que l'injection de dépendances, montre une bonne application de ces principes.

Globalement, le projet adhère donc bien aux principes SOLID, ce qui contribue à une architecture saine et facilite la maintenance et l'évolution du code.

ALI IBRAHIM Abchate

DEME Kewe

SAMBA Ndeye Fatou

SALEHUDDIN Ishrat

AZLOUK issam