Synthèse du Projet Java - Analyse et Améliorations

Introduction

Ce projet Java a été développé et amélioré sur plusieurs sprints afin d'optimiser la qualité du code, la correction des bugs et l'intégration d'outils de développement. Il repose sur JDK 11, Maven, ainsi que divers outils de CI/CD tels que Jenkins et SonarQube. Voici une synthèse des travaux réalisés.

🔹 Sprint 1 : Mise en place et Premiers Correctifs

1.1 Mise en place de l'environnement de développement

Installation de JDK 11, configuration de Maven et installation des outils de développement pour assurer un environnement stable.

1.2 Compréhension du Code et Diagnostic des Erreurs

Analyse du code source, identification des erreurs de compilation, de logique et d'exécution.

1.3 Réalisation du Diagramme de Classe

Création d'un diagramme UML pour visualiser la structure du projet et identifier les améliorations possibles.

1.4 Corrections et Sécurisation du Stockage des Mots de Passe

Mise en place d'un cryptage sécurisé pour stocker les mots de passe dans la base de données.

1.5 Création de Tests Unitaires pour la Sécurité

Tests unitaires pour vérifier la bonne implémentation du chiffrement des mots de passe.

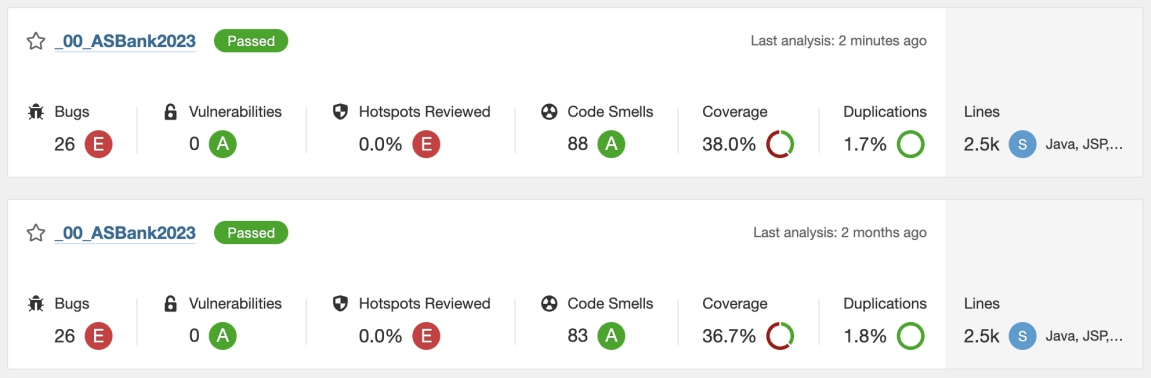
🔹 Sprint 2 : Améliorations de la Qualité et Correction de Bugs

2.1 Intégration de Jenkins et SonarQube

Mise en place de Jenkins pour l'intégration continue et SonarQube pour l'analyse de la qualité du code.

2.2 Comparaison des Analyses SonarQube

Comparaison des résultats entre une analyse réalisée il y a deux mois et celle d'aujourd'hui, montrant une amélioration de la couverture de code et une augmentation des Code Smells.



2.3 Correction de Bugs Bloquants

🔴 *Bug 1 : Erreur dans l'action "débiter"*

Correction de l'opération où le solde était augmenté au lieu d'être diminué.

Explication du bug :

Lors de la connexion à un compte client, l'utilisateur accède à son tableau de bord où sont affichés tous ses comptes. Lorsqu'un client clique sur le numéro de son compte, il est redirigé vers une page où il peut soit créditer (ajouter de l'argent) son compte, soit le débiter (retirer de l'argent). Le problème survenait lorsque l'utilisateur tentait de débiter de l'argent, car au lieu de retirer de l'argent, le système lui ajoutait de l'argent sur son compte.

Résolution :

Pour résoudre ce problème, il a été nécessaire de modifier la partie front-end de l'application qui gérait les envois des formulaires. Initialement, il y avait un seul formulaire pour les deux actions (créditer et débiter). Ce choix posait un problème, car Struts ne savait pas comment traiter ces deux actions différentes dans un même formulaire. En soumettant le formulaire, Struts interprétait la soumission du bouton "Débiter" comme une soumission d'un formulaire commun, ce qui provoquait des conflits dans le traitement des données.

```
<s:form name="formOperation" action="creditAction" method="post">
    <s:textfield label="Montant" name="montant" />
    <input type="hidden" name="compte"
        value="<s:property value='compte' />">
    <s:submit value="Crediter" />
    <s:submit value="Debiter" action="debitAction" />
</s:form>
```

Pour résoudre ce problème, j'ai séparé les formulaires pour les actions de crédit et de débit. Désormais, chaque formulaire possède son propre bouton et son action spécifique (debitAction pour le débit et creditAction pour le crédit).

Cela permet à Struts de savoir exactement quelle action exécuter lorsque l'utilisateur clique sur l'un ou l'autre des boutons, éliminant ainsi toute confusion.

```
<s:form name="formOperationDebit" action="debitAction" method="post">
    <s:textfield label="Montant" name="montant" />
    <input type="hidden" name="compte" value="<s:property value='compte' />">
    <s:submit value="Débiter" />
</s:form>

<s:form name="formOperationCredit" action="creditAction" method="post">
    <s:textfield label="Montant" name="montant" />
    <input type="hidden" name="compte" value="<s:property value='compte' />">
    <s:submit value="Créditer" />
</s:form>
```

🔴 *Bug 2 : Message d'erreur inadapté lors de la création d'un client*

Amélioration des messages d'erreur pour les rendre plus clairs et compréhensibles

Explication du bug :

Lorsqu'un utilisateur saisissait un code utilisateur (userId) incorrect tout en respectant le format du numéro de client (numClient), l'application affichait systématiquement le message : "Format du numéro de client incorrect". Ce message induisait en erreur, car il ne reflétait pas essentiellement la nature réelle du problème, qui pouvait concerner le userIdet non le numClient.

Impact :

- Confusion pour l'utilisateur qui croit que son « numéro de client » est faux alors qu'il a mal saisi le « code utilisateur ».
- Impossibilité de deviner le vrai format attendu pour le code utilisateur.
- Impossibilité de créer un nouvel utilisateur même si toutes les conditions sont respectées

Analyse du bug :

- On a découvert que, dans la classe Client, les méthodes **setUserId** et **setNumeroClient()** lèvent toutes deux une **IllegalFormatException** avec des messages différents (“L'identifiant n'est pas au bon format” et “Le numéro de client n'est pas au bon format“).
- Dans la classe **CreerUtilisateur,** dans le bloc **catch(IllegalFormatException e)**, la variable **this.message** était toujours affectée à **“Format du numéro de client incorrect“** en dur .
- **Les modifications apportées**
    - Dans **CreerUtlisateur** , méthode **creationUtilisateur()**, on a remplacé:

```
catch (IllegalFormatException e) {
            this.message = e.getMessage();
            this.result = "ERROR";
            return "ERROR";
```

Dans la classe **Client**, j’ai rendu les messages plus explicites , par exemple : throw new IllegalFormatException("L'identifiant n'est pas au bon format." + "Format attendu : ex. d.dupont123"); et throw new IllegalFormatException("Le numéro de client n'est pas au bon format." + "Format attendu : exactement 10 chiffres (ex. 1234567890)");

1. Confirmation par des tests:

a) Tests Unitaires (TU)

TU sur la logique de format

On a ajouté de nouveaux tests dans la classe TestsClientpour vérifier les méthodes :

- checkFormatUserIdClient(...)et checkFormatNumeroClient(...), qui contrôlent la validité des chaînes saisies pour le code utilisateur et le numéro de client.
- setUserId(...)et setNumeroClient(...), qui lèvent l'exception IllegalFormatExceptionen cas de format invalide.

```
@Test
public void testSetUserIdThrowsExceptionForInvalidFormat() {
    Client c = new Client();

    try {
       c.setUserId("JA");
       fail("Devrait lever IllegalFormatException pour userId
invalide !");
    } catch (IllegalFormatException e) {

       System.out.println("Exception message: " +
e.getMessage());
        assertTrue(e.getMessage().contains("n'est pas au bon
format"));
    }
}
```

b) Tests d'Intégration (TI)

- J'ai effectué un **test d'intégration** qui simule l'appel à la façade BanqueFacade.createClient(...)avec différents paramètres (un userIdet un numClientinvalides/valides), pour vérifier :
    a. Que la façade relais l'exception IllegalFormatExceptionquand le format est incorrect.
    b. Que le client est correctement créé dans la base de données quand tout est bon.

c) Tests de Recette (TR)

- Côté **interface utilisateur** , j'ai effectué un test manuel (ou automatisé) consistant à:
    a. Saisir un userIdinvalide (par ex. JA) et un numClientvalide (10 chiffres).
    b. Constater que l'application affiche désormais « L'identifiant n'est pas au bon format » au lieu de « Format du numéro de client incorrect ».
    c. Saisir un userIdvalide et un numClientinvalide (9 chiffres, par exemple) et vérifier que le message « Le numéro de client n'est pas au bon format » s'affiche.
    d. Saisir un userIdet un numClientvalide puis vérifier que la création du client se fait sans erreur.

```
[INFO] Tests run: 11, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.999 s -- in com.iut.banque.test.facade.TestsBanqueManager
[INFO] Running com.iut.banque.test.modele.TestsClient
Exception message: L'identifiant n'est pas au bon format.Format attendu : e. d.dupont123
Exception message: Le numéro de client n'est pas au bon format.Format attendu : exactement 10 chiffres (ex. 1234567890)
[INFO] Tests run: 27, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.009 s -- in com.iut.banque.test.modele.TestsClient
[INFO] Running com.iut.banque.test.modele.TestsCompte
[INFO] Tests run: 11, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.001 s -- in com.iut.banque.test.modele.TestsCompte
[INFO] Running com.iut.banque.test.modele.TestsCompteAvecDecouvert
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.003 s -- in com.iut.banque.test.modele.TestsCompteAvecDecouvert
[INFO] Running com.iut.banque.test.modele.TestsCompteSansDecouvert
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.002 s -- in com.iut.banque.test.modele.TestsCompteSansDecouvert
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 85, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  20.464 s
[INFO] Finished at: 2025-01-11T14:49:22+01:00
[INFO] ------------------------------------------------------------------------
```

2.4 Correction de Bugs Non Bloquants

🟡 *Bug 1 : Correction du CSS sur la page d'accueil*

Correction de l'affichage dégradé de la page d'accueil en appliquant correctement le CSS.

Explication du bug :

Dans l'ensemble des pages de l'application, le style CSS défini ne s'appliquait pas correctement, ce qui entraînait l'absence totale de mise en forme visuelle. La cause de ce problème était liée à une erreur dans le chemin spécifié pour accéder au fichier CSS : le lien <link rel="stylesheet" href="..."> pointait vers un emplacement incorrect.

Résolution :

Pour corriger ce bug, il a été nécessaire de modifier le chemin dans l'attribut href pour qu'il cible correctement le fichier CSS. Le nouveau chemin utilise la syntaxe suivante :

```
<link rel="stylesheet" href="${pageContext.request.contextPath}/style/style.css" />
```

Explication de la syntaxe ${pageContext.request.contextPath} **:**

- La variable ${pageContext.request.contextPath} est une expression JSP qui va permettre de récupérer dynamiquement le contexte de l'application.
- **Contexte de l'application :** C'est la partie de l'URL qui suit le domaine et représente le chemin de base où l'application est déployée sur le serveur. Par exemple :
  Si l'application est déployée à l'adresse http://localhost:8080/monApplication, alors ${pageContext.request.contextPath} renverra /monApplication.
  Si l'application est accessible directement à la racine, comme http://localhost:8080, alors ${pageContext.request.contextPath} renverra /.

En utilisant ${pageContext.request.contextPath}, le lien vers le fichier CSS devient dynamique et fonctionne correctement, quel que soit le contexte dans lequel l'application est déployée. Cela garantit que le fichier CSS sera toujours accessible sans avoir à modifier manuellement les chemins lors du déploiement.

🟡 *Bug 2 : Affichage du logo*

Correction du chemin d'accès au fichier image pour assurer l'affichage du logo.

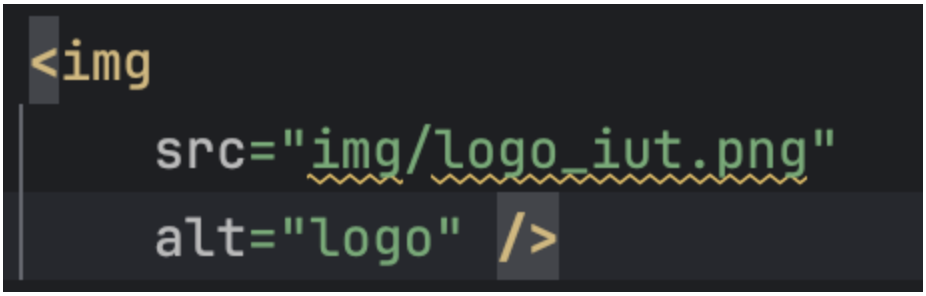
Explication du bug :

Lors du lancement de l'application, l'image du logo de l'Université de Lorraine ne s'affichait pas. Ce problème était dû à une URL incorrecte dans la balise <img>, qui était invalide.

```
<img
    src="https://www.iut-metz.univ-lorraine.fr/images/AdminSite/Logos/Logo_IUT_Metz.UL.small.png"
    alt="logo" />
```

Résolution du bug :

Pour résoudre ce problème, il a suffi de modifier le lien de l'image. J'ai téléchargé l'image et l'ai placée dans un nouveau dossier appelé img, que j'ai créé dans le projet. Ensuite, j'ai mis à jour le chemin de l'image dans la balise <img> pour pointer vers ce nouveau dossier.



🔹 Conclusion et Perspectives

Grâce aux différentes actions menées au cours de ces deux sprints, le projet a connu une nette amélioration sur plusieurs aspects :

1. Stabilisation du projet avec la correction des bugs majeurs et mineurs.

2. Sécurisation des données utilisateurs via le cryptage des mots de passe.

3. Automatisation et suivi de la qualité avec SonarQube et Jenkins.

4. Amélioration de l'interface utilisateur avec des corrections CSS et d'affichage.

Prochaines étapes possibles

• Améliorer la couverture des tests (atteindre 80% minimum).

• Optimiser les performances en analysant les temps de réponse.

• Réduire le nombre de Code Smells en refactorisant certaines parties du code.

• Automatiser le déploiement via un pipeline CI/CD complet.

En conclusion, ce projet a bénéficié d'une amélioration significative en termes de qualité, de sécurité et de maintenance.