

TP 4A - Génie Logiciel Programme Java intégrant modélisation UML, versionning (git) et tests unitaires (JUnit)

Mathis Vaugeois - Tanguy Moriceau - Faustine Guillou

January 2023

Contents

1	Introduction	2
1.1	Contexte	2
1.2	Git	2
2	Cahier des charges	4
3	Code de départ	6
4	Développement	9
4.1	Exercice 3 - Premiers développements	9
4.2	Exercice 4 - Fusion	13
4.3	Exercice 5 - Tests	19

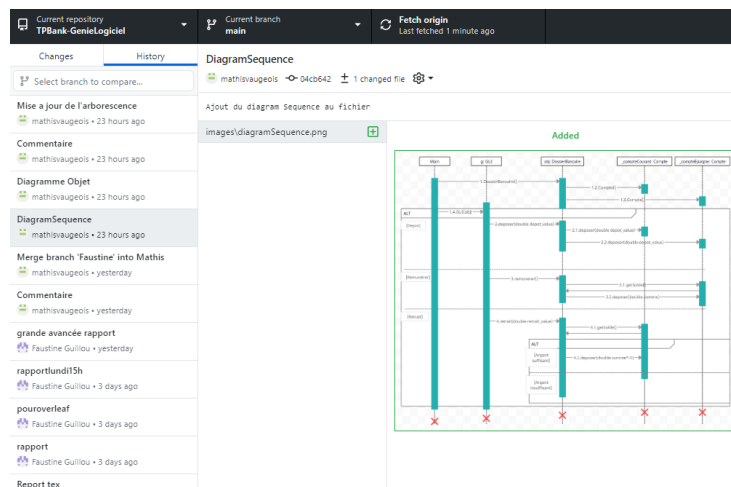
1 Introduction

1.1 Contexte

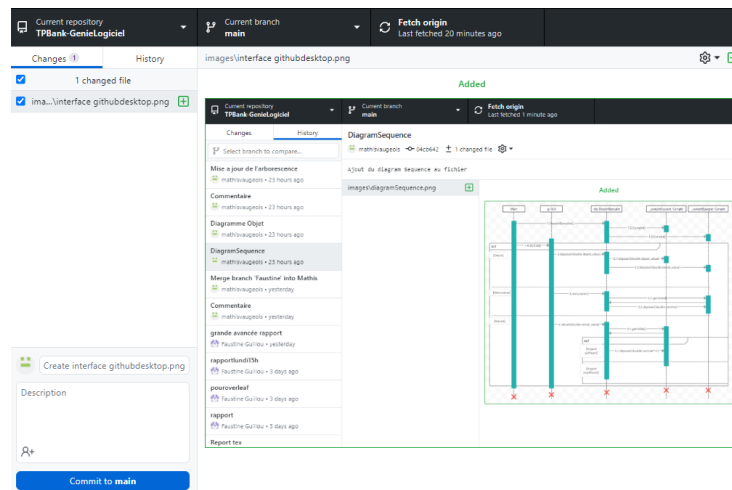
Ce TP final nous a permis de mettre en pratique tout ce que nous avons pu étudier et apprendre pendant les cours de Génie Logiciel. Nous avons créé une application de gestion d'un dossier bancaire. En premier lieu, pour bien organiser notre projet, nous avons commencé par réaliser des diagrammes. Nous avons mis en pratique ce que nous avons appris de la gestion des versions, pour travailler en équipe. De plus, nous avons utilisé les tests unitaires pour vérifier notre code.

1.2 Git

Pour faciliter les échanges de fichiers au sein du groupe, nous avons décidé d'utiliser github et plus particulièrement sont interfaces github desktop. Cela permet de facilement voir les changements des "commits" avant de les "push", l'historique, de changer de branche, ... Pour faire ceci, nous avons d'abord créé un projet sur github, qu'on s'est partagé pour pouvoir modifier directement dessus et non sur un clone. Nous avons ensuite créé différentes branches car travailler à plusieurs sur la même peut produire des conflits avec les différentes versions que l'on a push (ou pull) et donc séparer l'espace de travail permet de sauvegarder et récupérer la version à jour de la main avant de merge avec celle-ci pour limiter les bugs. Nous avons donc effectué de nombreux commit et push durant ce TP et pour mieux nous y retrouver, nous avons commenté certains commit avec un titre et une description facilitant la lecture et la compréhension des changements produits par le groupe.



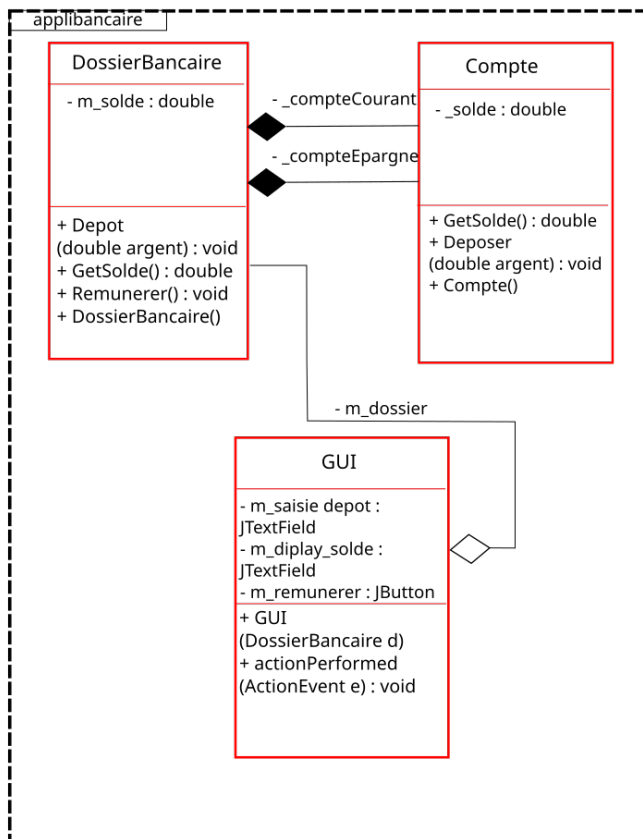
Ci-dessus, on peut voir l'interface de github desktop, on peut notamment remarquer, en haut à gauche, le projet dans lequel on travaille (ici le projet du TP), un bouton juste à sa droite où on peut choisir la branche (ici main) et un bouton avec plusieurs utilitaires : il fetch pour vérifier la version en ligne sur github, pull si le projet en ligne et différent, et push si notre projet a des commits à publier (ici on est à jour et github aussi). La barre à gauche peut afficher deux onglets. Nous avons l'onglet Changes (ci-dessous) qui nous donne les changements en cours à commit (nous permettant de mettre un titre et une description) ainsi qu'un rendu des différences des fichiers modifiés avec le précédent commits de la branche si l'application reconnaît le type du fichier (en général les images et les fichiers textes, dont les fichiers de code, sont reconnus). L'onglet History (ci-dessus) qu'en a lui, nous montre une liste défilante des commits avec la date de leur publication (ici elle est comparée à la date actuelle) et en sélectionnant un, on peut voir à droite le titre, la description ainsi que les différents fichiers qui ont été modifiés et leurs changements tout comme l'onglet Changes (c'est-à-dire que le commit sélectionné est comparé avec le précédent).



2 Cahier des charges

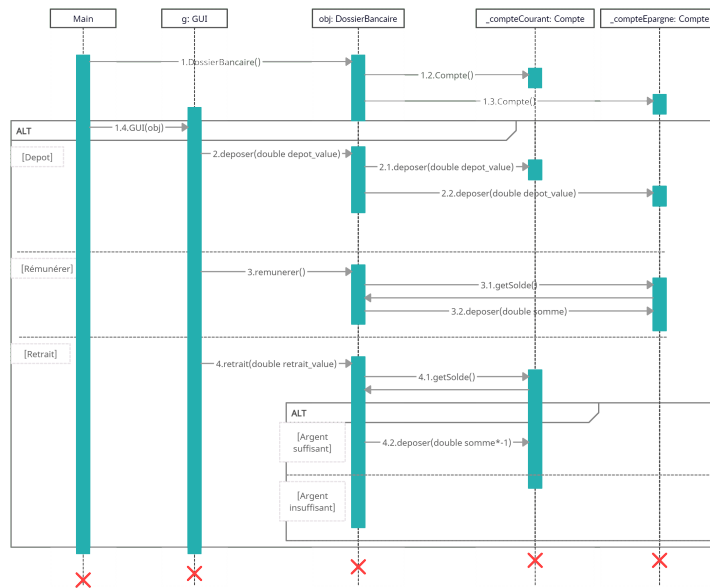
1. Proposez un diagramme UML de classes correspondant à ce cahier des charges.

Pour réaliser le diagramme de Classe, nous avons d'abord travaillé sur feuille et nous l'avons mis au propre en utilisant InkScape. Nous avons décidé de créer un seul type Compte, avec compte courant et compte épargne, deux entités différentes de Compte, car ils ont tous les deux les mêmes attributs. Compte dépend entièrement de Dossier Bancaire, donc si le dossier bancaire est supprimé, les comptes le seront aussi.



2. Donnez un diagramme UML de séquences illustrant les différents appels de méthodes et leurs relations.

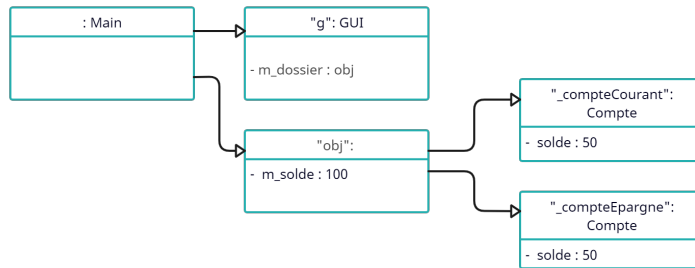
Nous avons ensuite réalisé un diagramme de Séquence. Il débute par le main, qui initialise DossierBancaire. Pour décider des actions, DossierBancaire dépend du GUI, c'est donc par ce dernier que se lance tous les ordres. Le premier ordre est déposer(). Via le GUI, le dépôt se lance, DossierBancaire dépose sur Compte Courant et Compte Epargne. Pour rémunérer, on récupère d'abord le solde du compte avant de prélever. Pour le retrait, il y a double situations. Si l'argent est suffisant, on retire, si il ne l'est pas, rien ne se passe. Pour effectuer des retraits sur le compte, nous n'avons pas créé une fonction. Nous réutilisons déposer mais en utilisant un chiffre négatif dont la valeur absolue est égale à la valeur du retrait.



3. Quel autre type de diagramme pourriez-vous proposer ?

En plus des deux diagrammes précédents, nous pourrions aussi proposer un diagramme d'objets. Le voici donc:

Il permet de rapidement voir tous les objets intervenant dans cette situation: le main, le GUI, le dossier bancaire et les deux types de compte.



3 Code de départ

3. Vérifier que vous parvenez, en ligne de commande, à compiler et exécuter le programme.)

Après avoir ouvert le projet, nous avons récupéré les commandes données dans le README, cependant cela ne fonctionnait pas.

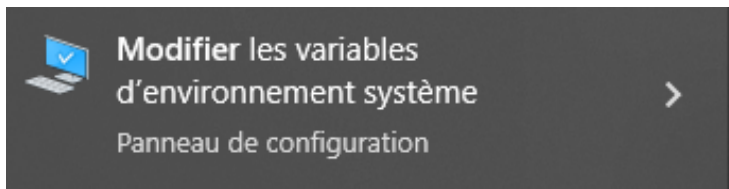
```

C:\Users\usrlocal> javac myPackage/Main.java
'javac' n'est pas reconnu en tant que commande interne
ou externe, un programme exécutable ou un fichier de commandes.

C:\Users\usrlocal>
  
```

Figure 1: Erreur de commande

Le problème était que javac n'était pas reconnu en commande interne. Pour résoudre ce problème, nous avons ajouté javac au PATH. Nous avons ouvert la page Modifier les variables d'environnement dans le panneau de configuration.



Ensuite, nous avons ouvert l'onglet variables d'environnement et avons modifié le PATH de l'utilisateur usrlocal.

4. Vérifiez que vous parvenez exécuter la suite de tests unitaires (Runner fourni, intégrant une méthode statique Main, déclenchant l'exécution de la suite de tests)

La compilation et l'exécution des tests fonctionnent de manière similaire à celle du Main. Nous avons tout d'abord récupéré la commande de compilation.

```
javac -cp "C:\Program Files (x86)\eclipse\plugins\org.junit_4.11.0.v201303080030\junit.jar";"C:\Program Files (x86)\eclipse\plugins\org.hamcrest.core_1.3.0.v201303031735.jar" tests\MyTest1.java tests\MyTest2.java ... etc...
```

Cette commande n'a pas fonctionné. Premièrement les chemins donnés étaient incorrects. De plus, il fallait ajouter tous les fichiers tests. La commande est donc devenue :

```
javac -cp "C:\eclipse\plugins\org.junit_4.13.2.v20211018-1956.jar";"C:\eclipse\plugins\org.hamcrest.core_1.3.0.v20180420-1519.jar" tests\MyTest1.java tests\MyTest2.java tests\MyTestSuite1.java tests\MyTestSuite1Runner.java myPackage\DossierBancaire.java
```

Ensuite, nous avons exécuté les tests avec la commande :

```
C:\Users\usrlocal\Desktop\JavaProgSujet\src>java -cp "C:\Program Files (x86)\eclipse\plugins\org.junit_4.11.0.v201303080030\junit.jar";"C:\Program Files (x86)\eclipse\plugins\org.hamcrest.core_1.3.0.v201303031735.jar"; tests/MyTestSuite1Runner
```

Comme la commande précédente, celle-ci n'a pas fonctionné. Les chemins étaient incorrects. La correction de la commande donne :

```
C:\Users\usrlocal\Desktop\JavaProgSujet\src>java -cp "C:\eclipse\plugins\org.junit_4.13.2.v20211018-1956.jar";"C:\eclipse\plugins\org.hamcrest.core_1.3.0.v20180420-1519.jar" tests/MyTestSuite1Runner
```


4 Développement

4.1 Exercice 3 - Premiers développements

1. Créez un dépôt git (dans le répertoire du projet). Sous éclipse, bouton droit sur le projet, Team, Share project, git, sélectionner use or create a repository in parent folder of project puis create repository, et finalement finish. Le répertoire de votre projet éclipse doit contenir un sous-répertoire .git/.

Nous avons déjà créé le dépôt git au tout début, via GitHub.

2. Ajoutez les fichiers (incluant les tests) de départ au dépôt git : add to index puis commit.

Comme pour la question précédente, cela était déjà fait.

3. Ajustez les tests unitaires pour n'avoir qu'une seule classe de test TestsDossierBancaire (on enlèvera les tests inutiles, et on (re)nommera correctement les fichiers). On implémentera un test (méthode de la classe TestsDossierBancaire) par méthode de la classe DossierBancaire : il y aura ainsi 3 tests (incluant le constructeur). La suite sera donc limitée à l'invocation des tests TestsDossierBancaire : on la renommera TestsSuite.

En premier lieu, nous avons retiré les tests inutiles. Nous avons déplacé les autres dans un fichier JUnit Test Case que nous avons nommé "TestDossierBancaire". De plus, nous avons écrit des tests supplémentaires pour tester les fonctionnalités de la classe DossierBancaire. Nous avons rajouté les tests 2,3 et 4.

Nous avons ensuite renommé le fichier TestsSuites. Voici le code du fichier de tests.

```
package tests;

import static org.junit.Assert.*;

import org.junit.Test;

import myPackage.DossierBancaire;

public class TestsSuite
{
    @Test
    public void test1()
    {
        DossierBancaire dossier = new DossierBancaire();
        dossier.deposer(100);
        assertEquals(100, dossier.get_solde(), 0);
    }
}
```

```
}
```

4. Intégrez ces modifications à git et enfin tagger cette version initiale (V1.0)

Nous avons intégré ces modifications à git via GitHubDesktop. Nous avons d'abord commit et push les changements. Pour tagger cette version, nous avons fait un clic-droit sur la version en question dans l'history, et nous avons cliqué sur create tag.

5. Ajoutez la classe compte courant seulement et intégrez la à la classe dossier bancaire : la rémunération ne change pas le solde, et le dépôt sur le dossier bancaire est intégralement affecté au compte courant. Ajouter les tests unitaires en conséquences et mettre la suite à jour (l'exécution de celle-ci doit impliquer deux nouveaux tests : constructeur et déposer). Ajouter les deux fichiers à l'index et Committer ces modifications.

Une classe Compte a été créée. Celle-ci contient un solde (privé), ainsi qu'un constructeur et deux méthodes permettant de récupérer le montant de ce solde, ainsi que de déposer de l'argent sur le compte.

```
package myPackage;

public class Compte
{
    private double solde;

    public Compte()
    {
        solde = 0;
    }

    public double getSolde()
    {
        return solde;
    }

    public void déposer(double somme)
    {
        solde += somme;
    }
}
```

Un attribut de type Compte a été ajouté au dossier bancaire (compteCourant), permettant d'accéder aux comptes.

```
private double m_solde;
private Compte _compteCourant;
```

```

public DossierBancaire()
{
    m_solde=0;
    Compte _compteCourant = new Compte();
}

```

La fonction déposer a été développée.

```

public void depoter(double argent)
{
    m_solde += argent;
    _compteCourant.depoter(argent);
}

```

Nous avons de plus rajouté une méthode avec d'accéder au compte couant.

```

public Compte getCompteCourant()
{
    return _compteCourant;
}

```

Ensuite, nous avons développé des tests pour ces changements.

```

@Test
public void test2()
{
    DossierBancaire dossier = new DossierBancaire();
    dossier.depoter(42);
    assertEquals(42, dossier.getCompteCourant().getSolde(), 0);
}

```

6. Ajoutez la classe compte épargne (CompteEpargne) et intégrez la à la classe dossier bancaire : la rémunération change le solde, et le dépôt sur le dossier bancaire est ventilé entre les deux comptes. Ajouter les tests unitaires sur CompteEpargne, ajustez ceux sur DossierBancaire et Commiter ces modifications.

Un attribut de type Compte a été ajouté au dossier bancaire (compteEpargne), permettant d'accéder aux comptes.

```

private double m_solde;
private Compte _compteCourant;
private Compte _compteEpargne;

public DossierBancaire()
{

```

```

        m_solde=0;
        Compte _compteCourant = new Compte();
        Compte _compteEpargne = new Compte();
    }

```

La fonction déposer a été modifiée, pour inclure le compte épargne.

```

    public void deposter(double somme)
    {
        m_solde += somme;
        _compteCourant.deposer(0.4 * somme);
        _compteEpargne.deposer(0.6 * somme);
    }

```

Le dossier bancaire contient aussi désormais une fonction de rémunération à taux fixe, qui dépose de l'argent sur le compte épargne.

```

    public void remunerer()
    {
        double somme = _compteEpargne.getSolde();
        somme = 0.032 * somme;
        _compteEpargne.deposer(somme);
    }
}

```

Nous avons aussi ajouté une méthode pour accéder au compte épargne.

```

public Compte getCompteEpargne()
{
    return _compteEpargne;
}

```

Ensuite, nous avons développé des tests pour ces changements.

```

@Test
    public void test3()
    {
        DossierBancaire dossier = new DossierBancaire();
        dossier.deposer(100);
        assertEquals(60, dossier.getCompteEpargne().getSolde(), 0);
        dossier.remunerer();
        assertEquals(61.92, dossier.getCompteEpargne().getSolde(), 0);
    }

```

Nous avons intégré ces modifications via GitHub Desktop en commitant.

7. Taggez cette version en 2.0.

Pour tagger cette version, nous avons fait un clic-droit sur la version en question dans l'history, et nous avons cliqué sur create tag.

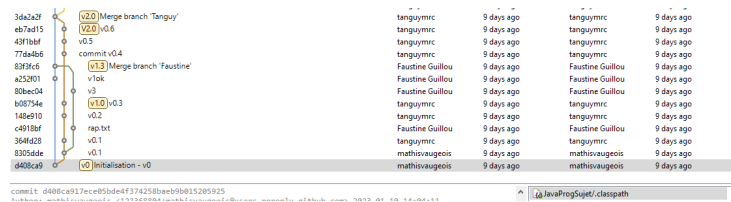
8. Testez le retour la version antérieure 1.0 (sélection du tag V1.0 dans team History, bouton droit et checkout) : consulter les fichiers, exécutez l'application (avec IHM : la rémunération est inactive), les tests unitaires.

Pour revenir à la version souhaitée, nous avons utilisé GitHub Desktop. Nous nous sommes placés sur le branche correct (main) et dans history, nous avons sélectionné le commit souhaité. Nous avons ensuite exécuté et testé la version en question.

9. Revenez finalement à la dernière version (V2.0).

Nous avons utilisé le même processus pour revenir à la dernière version.

A la fin de l'exercice, nous avons pu admirer ce graphique :



4.2 Exercice 4 - Fusion

1. Modifiez la classe DossierBancaire (e.g. documentation, renommage attribut) puis commiter.

Afin de répondre à cette question, nous avons légèrement modifié la classe DossierBancaire en ajoutant des commentaires.

```
package myPackage;

public class DossierBancaire
{
    private double m_solde;
    private Compte _compteCourant;
    private Compte _compteEpargne;

    public DossierBancaire() //constructeur du dossier bancaire
    {
        m_solde=0;
        _compteCourant = new Compte();
        _compteEpargne = new Compte();
    }

    public void depoter(double somme) //d poser de l'argent sur les deux compte
    {
        m_solde += somme;
    }
}
```

```

        _compteCourant.deposer(0.4 * somme);
        _compteEpargne.deposer(0.6 * somme);
    }

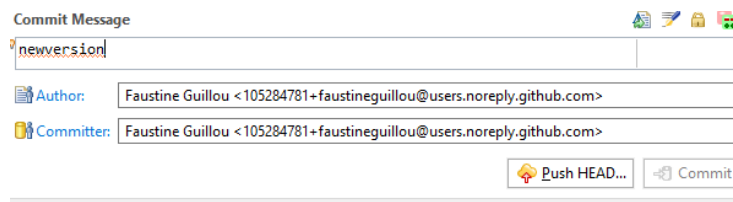
    public double get_solde() //renvoie le solde du dossier
    {
        return m_solde = _compteCourant.getSolde() + _compteEpargne.getSolde();
    }

    public Compte getCompteCourant() //renvoie le compte courant
    {
        return _compteCourant;
    }

    public Compte getCompteEpargne() //renvoie le compte   pargne
    {
        return _compteEpargne;
    }

    public void remunerer() //verser les int r ts sur le compte   pargne
    {
        double somme = _compteEpargne.getSolde();
        somme = 0.032 * somme;
        _compteEpargne.deposer(somme);
    }
}

```



2. Revenez à la version 2.0, créez une branche nommée newdev et modifiez, sur cette branche, le code : amélioration de la structure du code en intégrant l'héritage afin de factoriser des éléments des classes compte courant et compte d'épargne (faites au moins deux étapes -i.e. deux commits pour enrichir la branche).

Faisable avec GitDekstop. Amélioration de la structure du code précédemment réalisée (lors de l'initialisation du projet). Documentation enrichie.

3. Revenez sur la branche principale (master) et faites de nouvelles modifications mineures sur la classe Dossier Bancaire (sans créer de conflit : par

exemple ajoutez des commentaires).

Nous avons intégrés des commentaires afin d'effectuer des modifications.

```
public class DossierBancaire
{
    private double m_solde; //solde g  n  ral des deux comptes bancaires
    private Compte _compteCourant; //Element de la classe Compte
    private Compte _compteEpargne; //Si on en veut plusieurs on peut facilement

    public DossierBancaire() //constructeur du dossier bancaire
    {
        m_solde = 0;
        _compteCourant = new Compte(); //Creation d'un compte
        _compteEpargne = new Compte();
    }

    public void deposer(double somme) //d  poser de l'argent sur les deux comptes
    {
        m_solde += somme;
        _compteCourant.deposer(0.4 * somme); //Depose 40% de la somme
        _compteEpargne.deposer(0.6 * somme); //Depose 60% de la somme
    }

    public double get_solde() //renvoie le solde du dossier
    {
        return m_solde = _compteCourant.getSolde() + _compteEpargne.getSolde();
    }

    public Compte getCompteCourant() //renvoie le compte courant
    {
        return _compteCourant;
    }

    public Compte getCompteEpargne() //renvoie le compte   pargne
    {
        return _compteEpargne;
    }

    public void remunerer() //verser les int  r  ts sur le compte   pargne
    {
        double somme = _compteEpargne.getSolde(); //Recupere le solde actuelle
        somme = 0.032 * somme; //Fait le calcul
        _compteEpargne.deposer(somme); //Mise a jour le solde
    }

    public void retrait(double somme)
```

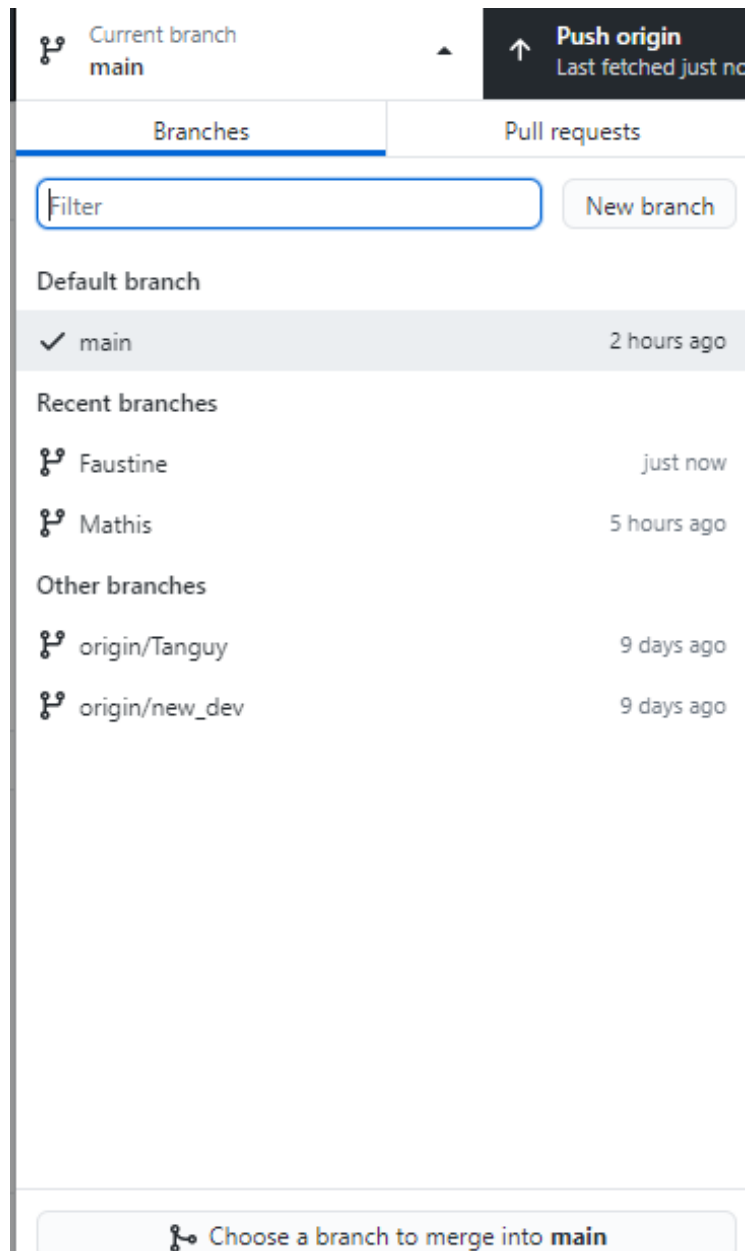
```

    {
        double current_value = _compteCourant.getSolde(); //Recupere le solde
        if (somme <= current_value)
        {
            //OK POUR LE RETRAIT
            _compteCourant.deposer(-somme);
        }
        else
        {
            //PAS OK POUR LE RETRAIT
            //SEND ERREUR
            System.out.println("Solde insuffisant");
        }
    }
}

```

4. Intégrez les modifications de la branche newdev, en revenant au préalable sur le master.

Sur GitHub Desktop, nous sommes revenus sur la branche master, puis nous avons sélectionné une branche à intégrer dans main, ici newdev.



5. Vérifiez le bon fonctionnement de l'application.

Après vérification, les tests et le GUI fonctionnent parfaitement.

Editeur dossier bancaire

Depot

Remunerer

Retrait

Solde

OK

0.0

A la fin de l'exercice, nous avons pu admirer ce graphique :

The figure displays a Git commit history graph. On the left, a vertical timeline shows the sequence of commits, with messages and commit hashes. The messages include 'Init', 'Commentaire', 'Fin de projet', 'Merge branch \'new_dev\'', 'Update DossierBancaire.java', 'Update TestsSuite.java', 'new_dev init', 'Merge branch \'Tanguy\'', 'v0.6', 'v0.5', 'commit v0.4', 'v1.0', 'Merge branch \'Faustine\'', 'v1.0', 'v3', 'v1.0', 'v0.3', 'v0.2', 'rap.txt', 'v0.1', 'v0.1', and 'Initialisation - v0'. The right side of the graph shows a table of commit details.

Id	Message	Author	Authored Date	Committer	Committed Date
a6b700	Init	Faustine Guillou	6 hours ago	Faustine Guillou	6 hours ago
42d13a5	Commentaire	mathisvaugois	22 hours ago	mathisvaugois	22 hours ago
82d830a	Commentaire	mathisvaugois	23 hours ago	mathisvaugois	23 hours ago
2134015	Report tex	tanguymrc	9 days ago	tanguymrc	9 days ago
a0044ee	Fin de projet	tanguymrc	9 days ago	tanguymrc	9 days ago
2affce6	Arbre de recherche	tanguymrc	9 days ago	tanguymrc	9 days ago
4a7c23f	Merge branch 'new_dev'	tanguymrc	9 days ago	tanguymrc	9 days ago
8c1ab0b	Update DossierBancaire.java	tanguymrc	9 days ago	tanguymrc	9 days ago
3c8968d	Update TestsSuite.java	tanguymrc	9 days ago	tanguymrc	9 days ago
2c387cc	new_dev init	tanguymrc	9 days ago	tanguymrc	9 days ago
3a62a2f	Merge branch 'Tanguy'	tanguymrc	9 days ago	tanguymrc	9 days ago
e8b7d15	v0.6	tanguymrc	9 days ago	tanguymrc	9 days ago
43f1bbf	v0.5	tanguymrc	9 days ago	tanguymrc	9 days ago
77da4b6	commit v0.4	tanguymrc	9 days ago	tanguymrc	9 days ago
83f3c6	v1.0	Faustine Guillou	9 days ago	Faustine Guillou	9 days ago
a25201	Merge branch 'Faustine'	Faustine Guillou	9 days ago	Faustine Guillou	9 days ago
80bec04	v3	Faustine Guillou	9 days ago	Faustine Guillou	9 days ago
b08754e	v1.0	tanguymrc	9 days ago	tanguymrc	9 days ago
148e910	v0.3	tanguymrc	9 days ago	tanguymrc	9 days ago
c491bbf	v0.2	Faustine Guillou	9 days ago	Faustine Guillou	9 days ago
3a6d28	rap.txt	tanguymrc	9 days ago	tanguymrc	9 days ago
8305d4e	v0.1	mathisvaugois	9 days ago	mathisvaugois	9 days ago
d408ca9	Initialisation - v0	mathisvaugois	9 days ago	mathisvaugois	9 days ago

18

4.3 Exercice 5 - Tests

1. Ajoutez la possibilité de retirer de l'argent du dossier bancaire : on considérera qu'un tel retrait n'altère que la classe Compte Courant, et qu'une exception est levée si le solde est insuffisant.

Une fonction de retrait a été ajoutée à la classe DossierBancaire et permet de retirer de l'argent du compteCourant, avec vérification du solde préalable à l'opération.

2. Faites en sorte que la levée de cette exception soit vérifiée par le test unitaire associée (voir documentation JUnit).

Nous avons mis en place un test JUnit pour vérifier que la fonction de retrait fonctionne correctement.

```
@Test
public void test4()
{
    DossierBancaire dossier = new DossierBancaire();
    dossier.deposer(100);
    dossier.retrait(30);
    assertEquals(10, dossier.getCompteCourant().getSolde(), 0);
}
```

3. Modifiez la classe GUI, pour permettre un retrait depuis l'interface graphique.

Ensuite, nous avons modifié l'interface GUI, afin de permettre d'intégrer la fonctionnalité de retrait.

```
public class GUI implements ActionListener
{
    private DossierBancaire m_dossier;
    private JTextField m_saisie_depot;
    private JTextField m_display_solde;
    private JButton m_remunerer;
    private JTextField m_saisie_retrait;

    //ajout du JTextField m_saisie_retrait

    //Element saisie retrait
    m_saisie_retrait = new JTextField (20);
    m_saisie_retrait.addActionListener(this);

    frame.getContentPane().add(m_remunerer);
    frame.getContentPane().add(new JLabel("Retrait"));
    frame.getContentPane().add(m_saisie_retrait);
}
```

```

if( e.getSource() == m_saisie_retrait )
{
    double retrait_value=Double.parseDouble
(m_saisie_retrait.getText());
    m_dossier.retrait(retrait_value);
    m_saisie_retrait.setText("");
}
m_display_solde.setText(Double.toString(m_dossier.get_solde()));

```

4. Pensez avec versionner cette nouvelle version 3.0.

Après avoir fait ces nouvelles modifications, nous avons commit sur GitHub Desktop. Ensuite, nous avons push. Dans l'interface history énumérant tous les commit, nous avons fait un clic droit sur le dernier et avons sélectionné create tag.

Tout au long de la partie concernant Eclipse, nous avons alterné entre Eclipse et GitHub Desktop pour commit nos changements. Les deux fonctionnent tout aussi bien l'un que l'autre. La seule différence entre les deux interfaces sont les fichiers commit. Eclipse ne commit que le projet lui-même. Avec Github, nous pouvons commit en même temps d'autres fichiers faisant parti du dépôt mais étant en dehors du projet Eclipse. C'est cette différence qui a déterminé l'alternance entre ces deux interfaces.

Référence

<https://github.com/mathisvaugeois/TPBank-GenieLogiciel>