

# TP 4A - Génie Logiciel

## Programme Java intégrant modélisation UML, versionning (git) et tests unitaires (JUnit)

Nicolas Delanoue


L'objectif de ce TP est de combiner les connaissances et compétences acquises autour d'un unique projet. Plus précisément, vous allez créer une application minimaliste de gestion d'un dossier bancaire, en intégrant une modélisation UML, ainsi que deux autres pratiques fondamentales en génie logiciel : les tests unitaires et la gestion des versions.

### 1 Cahier des charges

La section suivante présente le cahier des charges de l'application tel qu'elle doit fonctionner à la fin de ce projet :

- Un **dossier bancaire** comprend deux comptes : un **compte courant** et un **compte d'épargne**. On peut **déposer de l'argent** sur chacun des comptes, afin d'augmenter le **solde**.
- Le dépôt se fait par l'intermédiaire du dossier bancaire, qui **ventile automatiquement une somme versée** entre les deux comptes : 60% sur le compte épargne et le reste sur le compte courant.
- Le **solde du dossier bancaire** correspond à la somme des soldes des deux comptes. Seul le compte d'épargne **peut être rémunéré en fonction de son taux d'intérêt** (fixé à 3.2% dans notre cas).
- Un utilisateur n'interagit qu'avec un dossier bancaire via les actions suivantes :
  - il peut déposer de l'argent (automatiquement ventilée entre les deux comptes),
  - il peut le rémunérer (i.e. rémunérer le compte épargne),
  - il peut consulter le solde (somme des soldes des deux comptes).

Le fonctionnement attendu est illustré par la figure 1.



Editeur dossier bancaire		Editeur dossier bancaire		Editeur dossier bancaire	
Depot	<input type="text"/>	Depot	<input type="text"/>	Depot	<input type="text"/>
Remunerer	<input type="button" value="OK"/>	Remunerer	<input type="button" value="OK"/>	Remunerer	<input type="button" value="OK"/>
Solde	0.0	Solde	100.0	Solde	101.92

FIGURE 1 – Exemple de comportement attendu.

### Exercice 1

1. Proposez un diagramme UML de classes correspondant à ce cahier des charges.
2. Donnez un diagramme UML de séquences illustrant les différents appels de méthodes et leurs relations.

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

## 2 Code de départ

Un code élémentaire de départ est fourni, intégrant une IHM minimaliste permettant de déposer de l'argent et de rémunérer le dossier bancaire. L'API de la classe «Dossier bancaire» est fournie : ceci permettra de laisser l'IHM inchangée au cours des développements, celle-ci interagissant seulement avec l'API de la classe dossier bancaire. La classe GUI se sera donc pas modifiée durant les développements.

Ce code élémentaire comprend également des tests unitaires. Typiquement, une classe de tests a pour vocation de tester les méthodes d'une classe donnée (voir l'exemple fourni). Une «suite» permet d'invoquer une série de tests (i.e. classes de tests). Enfin, on peut également avoir un programme «d'amorce» dédié à l'exécution de la suite de tests.

### Exercice 2 (Compilation en ligne de commande)

1. Téléchargez le code disponible via [http://perso-laris.univ-angers.fr/~delanoue/polytech/genie\\_logiciel/](http://perso-laris.univ-angers.fr/~delanoue/polytech/genie_logiciel/)
2. Appropriiez-vous le code fourni et lisez les fichiers README.
3. Vérifier que vous parvenez, en ligne de commande, à compiler et exécuter le programme.
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).

## 3 Développement

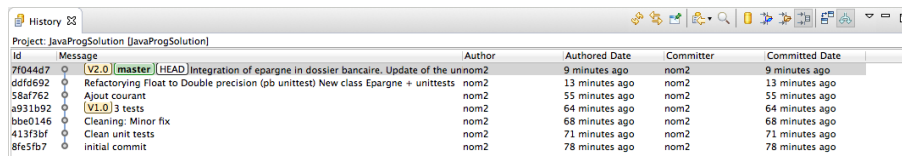
### Exercice 3 (Premiers développements)

On respectera les étapes suivantes (voir tableau ci-après pour une vue des fichiers et d'un exemple d'historique des versions GIT) :

1. Créez un dépôt git (dans le répertoire du projet). Sous eclipse, 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 eclipse doit contenir un sous-répertoire «.git/».
2. Ajoutez les fichiers (incluant les tests) de départ au dépôt git : «add to index» puis «commit».
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».
4. Intégrez ces modifications à git et enfin «tagger» cette version initiale (V1.0)
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 «deposer»). Ajouter les deux fichiers à l'«index» et «Committer» ces modifications.

- 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.
- Taggez cette version en 2.0.
- Testez le retour la version antérieure 1.0 (sélection du «tagV1.0» dans «team History», bouton droit et «checkout») : consulter les fichiers, exécutez l'application (avec IHM : la rémunération est «inactive»), les tests unitaires.
- Revenez finalement à la dernière version (V2.0).

A la fin de cet exercice, vous devriez pouvoir admirer ce genre de graphique dans Eclipse (Voir figure 2).



Id	Message	Author	Authored Date	Committer	Committed Date
7f044d7	[V2.0] [master] HEAD integration of epargne in dossier bancaire, Update of the unnom2	nom2	9 minutes ago	nom2	9 minutes ago
ddfd692	Refactoring Float to Double precision (pb unittest) New class Epargne + unittests	nom2	13 minutes ago	nom2	13 minutes ago
58af762	Ajout courant	nom2	55 minutes ago	nom2	55 minutes ago
a931b92	[V1.0] 3 tests	nom2	64 minutes ago	nom2	64 minutes ago
b4e0146	Cleaning: Minor fix	nom2	68 minutes ago	nom2	68 minutes ago
413f3bf	Clean unit tests	nom2	71 minutes ago	nom2	71 minutes ago
8fe5fb7	Initial commit	nom2	78 minutes ago	nom2	78 minutes ago

FIGURE 2 – Exemple d'historique «git» (intégrant quelques modifications complémentaires).

#### Exercice 4 (Fusion)

L'objectif est d'améliorer la conception et d'exploiter la notion de branche avec Git.

- Modifiez la classe «DossierBancaire» (e.g. documentation, renommage attribut) puis «commiter».
- Revenez à la version «2.0», créez une branche nommée «new\_dev» 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).
- 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).
- Intégrez les modifications de la branche «new\_dev», en revenant au préalable sur le «master».
- Vérifiez le bon fonctionnement de l'application.

A la fin de cet exercice, vous devriez pouvoir admirer ce genre de graphique dans Eclipse (Voir figure 3).

Project: JavaProgSolution [JavaProgSolution]							
Id	Message	Author	Authored Date	Committer	Committed		
4b7151b	<b>master</b> (HEAD) Merge branch 'new_dev'	nom2	5 minutes ago	nom2	5 minutes a		
bf32e10	new documentation	nom2	5 minutes ago	nom2	5 minutes a		
2b0e4d6	<b>new_dev</b> inheritance finished	nom2	6 minutes ago	nom2	6 minutes a		
30e3afc	new classe "compte"	nom2	11 minutes ago	nom2	11 minutes		
fb006da	documentation	nom2	13 minutes ago	nom2	13 minutes		
7f044d7	<b>V2.0</b> Integration of epargne in dossier bancaire. Updanom2	nom2	3 months ago	nom2	3 months a		
ddfd692	Refactoring Float to Double precision (pb unittest) New nom2	nom2	3 months ago	nom2	3 months a		
58af762	Ajout courant	nom2	3 months ago	nom2	3 months a		
a931b92	<b>V1.0</b> 3 tests	nom2	3 months ago	nom2	3 months a		
bbe0146	Cleaning: Minor fix	nom2	3 months ago	nom2	3 months a		
413f3bf	Clean unit tests	nom2	3 months ago	nom2	3 months a		
8fe5fb7	initial commit	nom2	3 months ago	nom2	3 months a		

commit 4b7151b754a7f11f0a11c0e0287f1952a1406f10

Author: nom2 <nom2@email.com> 2016-03-07 17:30:53

Committer: nom2 <nom2@email.com> 2016-03-07 17:30:53

Parent: [bf32e10866dbe0a7d808f31a9df705c4ade3af4c](#) (new documentation)

Parent: [2b0e4d69eff1376ed8ee2c337c0b4b9f76e82950](#) (inheritance finished)

Branches: [master](#)

Merge branch 'new\_dev'

FIGURE 3 – Exemple d'historique «git» (intégrant quelques modifications complémentaires).

### Exercice 5 (Tests)

L'objectif est d'ajouter une nouvelle fonctionnalité et de tester (test unitaire) la levée d'une exception.

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.
2. Faites en sorte que la levée de cette exception soit vérifiée par le test unitaire associée (voir documentation JUnit).
3. Modifiez la classe «GUI», pour permettre un retrait depuis l'interface graphique.
4. Pensez avec «versionner» cette nouvelle version «3.0».