

S-INFO-015 Projet d'analyse et de conception  
et S-INFO-106 Projet de développement logiciel

## **Gestion de portefeuille financier**

Une application client-serveur en Java

Enseignant: Professeur Tom Mens

Année Académique 2021-2022  
Sciences Informatiques

Faculté des Sciences, Université de Mons

*Date de ce document: 28 septembre 2021*

### CHANGE LOG

v1.1.0: Changement de la taille des groupes, précisions sur la pondération du travail.  
v1.0.0: La première version stable de l'énoncé, rendu aux étudiants.

# Table des matières

<b>1</b>	<b>Énoncé</b>	<b>3</b>
1.1	Préambule . . . . .	3
1.2	Introduction et contexte du projet . . . . .	3
1.3	Terminologie spécifique au domaine financier . . . . .	5
1.4	Exigences technologiques . . . . .	7
1.4.1	Architecture REST . . . . .	7
1.4.2	Testabilité de l'application . . . . .	10
1.5	Fonctionnalités de base . . . . .	12
1.5.1	Internationalisation et gestion des langues . . . . .	12
1.5.2	Application titulaire pour gérer ses “portefeuilles financiers” . . . . .	12
1.5.3	Application pour une institution financière . . . . .	14
1.6	Fonctionnalité des extensions . . . . .	15
1.6.1	Gestion de cartes . . . . .	15
1.6.2	Gestion de devises et virements internationaux . . . . .	17
1.6.3	Gestion de placements . . . . .	18
1.6.4	Gestion du budget financier . . . . .	21
1.6.5	Gestion des contrats d'assurance . . . . .	22
1.6.6	Paievements et gestion de fraudes . . . . .	24
<b>2</b>	<b>Exigences</b>	<b>26</b>
2.1	Étapes clés et livrables . . . . .	26
2.2	Organisation et conditions du travail . . . . .	29
2.3	Outils . . . . .	29
<b>3</b>	<b>Livrables du projet</b>	<b>32</b>
3.1	Démarrage . . . . .	32
3.2	Phase d'analyse et de conception (S-INFO-015) . . . . .	32
3.3	Phase de développement logiciel (S-INFO-106) . . . . .	33
<b>4</b>	<b>Critères de recevabilité</b>	<b>35</b>
<b>5</b>	<b>Critères d'évaluation</b>	<b>36</b>
5.1	Pour la phase d'analyse de de conception . . . . .	36
5.1.1	Maquette de l'interface utilisateur . . . . .	36
5.1.2	Rapport de modélisation . . . . .	36
5.2	Pour la phase d'implémentation . . . . .	37
5.2.1	Qualité du code source . . . . .	37
5.2.2	Testabilité . . . . .	37
5.2.3	Exécutabilité et fonctionnalité . . . . .	38
5.2.4	Documentation . . . . .	38
<b>6</b>	<b>Dates importantes</b>	<b>39</b>

# 1 Énoncé

## 1.1 Préambule

Le projet est composé de deux parties (phase d'analyse et de conception et phase de développement logiciel), chacune correspondant à une AA différente (S-INFO-015 en Q1 et S-INFO-106 en Q2). Selon l'UE qui se trouve dans votre PAE, vous devez soit suivre les deux AA ou uniquement la deuxième. Dans ce dernier cas, vous pouvez ignorer la partie de ce document qui concerne la phase d'analyse et de conception.

Le but du projet consiste à modéliser (au moyen d'une maquette de l'interface utilisateur et de la conception en UML) un système logiciel non-trivial dans le cadre de l'AA S-INFO-015, et développer ce système (c'est à dire implémenter en Java, tester avec JUnit et documenter avec JavaDoc) dans le cadre de l'AA S-INFO-106. La **qualité** des modèles UML et du code source, ainsi que la **testabilité** du produit logiciel est considérée comme *au moins aussi importante* que la fonctionnalité, robustesse et convivialité de l'application développée.

Ce projet consiste à réaliser en groupe une **application logicielle de gestion de portefeuille financier**. L'application sera de type client-serveur. Le serveur devra contenir essentiellement les données à gérer, et offrir accès aux données par le biais d'une architecture REST.

Bien que le projet fasse partie d'un cours universitaire, son contenu correspond à des besoins réels, il ne s'agit donc pas juste d'un sujet artificiel. La qualité interne du projet (à la fois au niveau du code source, des tests unitaires et de la documentation du code) et la qualité externe (la fonctionnalité, l'ergonomie et la fluidité) sont donc primordiales.

Les enseignants du cours figureront comme clients et utilisateurs externes de votre application. Si vous trouvez des incohérences, ambiguïtés ou incomplétudes dans l'énoncé, veuillez donc contacter les enseignants pour clarifier les choses. Cela permet d'éviter beaucoup de problèmes ou malentendus pendant la modélisation et l'implémentation du projet, qui pourraient impacter négativement votre note finale.

## 1.2 Introduction et contexte du projet

Chaque groupe sera composé de **quatre personnes**<sup>1</sup> et proposera des services de gestion d'un portefeuille financier. Chaque groupe peut être vu comme une entreprise qui commercialise des applications bancaires. L'une des applications doit permettre à une institution financière (par exemple, une banque ou une compagnie d'assurance) de gérer les produits financiers offerts à ses clients (particuliers ou entreprises). L'autre application doit permettre à un client d'une ou plusieurs institutions financières de gérer le suivi de ses portefeuilles financiers. Les données nécessaires pour le bon fonctionnement des deux applications seront hébergées dans une base de données sur un serveur, qui se synchronisera avec les applications qui s'y connectent à travers une API REST.

*Par exemple, supposons que Jeanne est cliente de deux banques B1 et B2. Elle utilisera une instance de l'application cliente pour visualiser et manipuler ses deux portefeuilles financiers (un par banque). Pour leur part, les deux banques utiliseront (de manière continue), une instance de l'application qui leur permet de gérer les produits financiers de tous leurs clients (dont Jeanne). Les applications ne communiquent pas directement entre elles, toute interaction passe par l'API du serveur qui héberge la base de données. Par exemple, si Jeanne souhaite créer un nouveau produit financier (par exemple, un compte épargne) chez B1, elle peut introduire la demande par son application cliente, qui enverra une requête par HTTPS au serveur. L'application utilisée par B1 sera notifiée par le serveur (par une autre requête HTTPS) de la demande de Jeanne et un employé de la banque pourra alors traiter la demande, après quoi sa décision sera envoyée, par une autre requête HTTPS, au serveur. En cas d'acceptation, le serveur contactera (par une requête) l'application cliente utilisée par Jeanne afin de mettre à jour son portefeuille financier en incluant le nouveau produit financier. En cas de refus, le serveur avertira l'application cliente de celui-ci.*

La terminologie nécessaire pour la compréhension de l'énoncé est décrite en section 1.3. Les exigences technologiques sont détaillées en section 1.4. Chaque groupe doit implémenter les mêmes fonctionnalités de

---

1. Des plus petits groupes seront uniquement permis dans des cas exceptionnels.

base décrites en section 1.5. De plus, chaque étudiant devra individuellement réaliser une des extensions des fonctionnalités de base (décrites en section 1.6) et les intégrer dans le projet de son groupe. Le choix des extensions se fera en accord commun entre les étudiants et les enseignants.

### 1.3 Terminologie spécifique au domaine financier

Cette section décrit la terminologie du domaine financier qui est nécessaire pour comprendre l'énoncé de la fonctionnalité de base. La terminologie qui est nécessaire pour comprendre les extensions spécifique sera décrite plus tard.

À cause de la complexité du domaine financier, l'information fournie est volontairement lacunaire. Si vous désirez plus de renseignements sur le paysage financier en Belgique, vous les trouverez sur les sites suivants. Nous vous conseillons vivement de les consulter avant et pendant le développement du projet, car ils fournissent toute une série d'informations utiles :

- Instruments et rendements. Le Vif Extra, 5 mai 2021 (une copie de cet article se trouve sur Moodle)  
<https://www.levif.be/actualite/magazine/instruments-rendements/article-normal-1421443.html>
- Guides pratiques banque  
<https://www.killmybill.be/fr/guides/banque/>
- Comptes en banque : lesquels sont à votre disposition ?  
<https://www.killmybill.be/fr/comptes-banque/>
- Les différents types de comptes bancaire en Belgique  
<http://www.blog-notes-finances.com/presentation-comptes-bancaires-belge/>

N'hésitez pas à chercher et à consulter d'autres sources d'information par vous-mêmes pour vous documenter davantage afin de mener à bien ce projet.

#### Les comptes bancaires

Type de compte	Caractéristiques
Compte courant (compte à vue)	Rendement : aucun Ce type de compte est fréquemment sujet à des frais annuels.
Compte jeune	Compte courant limité jusqu'à un certain âge (typiquement 24 ans) et offrant plus d'avantages et moins de frais qu'un compte courant normal.
Compte épargne	Rendement annuel : 0,11% rendement minimal garanti (0,01% intérêt + 0,10% prime de fidélité)
Compte à terme	Argent placé pour une durée fixe. Le rendement annuel sur 5 ans varie typiquement entre 0,1% et 0,5%

TABLE 1 – Types de comptes bancaires.

Les comptes bancaires sont des produits financiers avec un risque très faible, mais le rendement annuel sera également faible. On peut distinguer différents types de comptes bancaires. La Table 1 présente un résumé. De manière générale, on reçoit un rendement annuel sur l'argent disponible sur son compte. Si le solde sur son compte est négatif, le rendement sera négatif pendant toute la durée du solde négatif. Le rendement (positif ou négatif) peut varier selon le type de compte.

Tout individu majeur peut être titulaire d'un **compte courant** (aussi appelé **compte à vue**) chez une ou plusieurs banques pour gérer son budget. Ce compte est utilisé pour les transactions bancaires quotidiennes telles que les achats, le paiement des factures, la réception des salaires mensuels et des allocations familiales, les virements, les versements, etc...

Des individus jusqu'à un certain âge (souvent 24 ans) peuvent être titulaire d'un **compte jeune**. Si la personne n'est pas encore adulte (donc  $< 18$  ans), un adulte (typiquement un des parents) doit être co-titulaire du compte. Certaines transactions dépassant une certaine limite seront uniquement possibles par le titulaire adulte.

Un individu peut être titulaire d'un ou plusieurs **comptes épargne** qui produisent plus d'intérêts qu'un compte courant. Ce type de compte permet de mettre de côté de l'argent dont on n'a pas besoin immédiatement. Un compte épargne doit toujours être lié à un compte courant du même titulaire. Si on laisse son argent sur ce compte pendant une certaine durée (typiquement 1 an), on reçoit une prime de fidélité.

Un individu peut créer un **compte à terme** s'il souhaite mettre de côté une certaine somme d'argent durant une période prédéterminée. En échange, il obtient un intérêt plus important que sur un compte

épargne. Toutefois, un retrait (partiel ou complet) de l'argent déposé avant échéance peut engendrer des pénalités financières.

Par défaut, à l'exception du compte jeune, une seule personne est titulaire d'un compte bancaire. On parle alors d'un **compte individuel**. Il est aussi possible d'avoir des **comptes joints**, partagés par plusieurs personnes qui seront toutes co-titulaires et solidairement responsables du compte. Chaque co-titulaire peut effectuer des transactions sur un compte joint sans avoir besoin de la signature des autres co-titulaires. Les comptes joints sont typiquement utilisés par des couples mariés ou par des co-habitants légaux. Un **compte indivis (ou compte en indivision)** est un compte joint à partir duquel les transactions nécessitent la signature de tous les co-titulaires.

L'argent sur un compte s'exprime dans une certaine **devise**, par défaut en € (EUR). Cependant, il est possible d'avoir des comptes dans une autre devise, comme un compte courant en dollars (USD) ou un compte d'épargne en livres sterling (GBP). Un **compte bancaire multidevises** est un compte qui est sous-divisé en plusieurs compartiments. Chaque compartiment sera dans une devise différente. Ce type de compte permet de transférer de l'argent d'un compartiment à l'autre avec des frais de conversion assez limités.

Un compte est identifié de manière unique par son **code IBAN**<sup>2</sup>, c'est un format standard en Europe pour identifier les comptes bancaires. Les banques en Europe sont identifiées de manière unique par leur **code SWIFT ou BIC**<sup>3</sup>. Pour effectuer des virements internationaux, le code IBAN et le code SWIFT/BIC sont obligatoires.

- Voir par exemple <https://bank-codes.fr/iban/structure/belgium/> pour connaître la structure du format IBAN.
- Voir par exemple <https://bank-codes.fr/swift-code/> pour la structure du code SWIFT.

## Les opérations bancaires

Un **retrait** est une opération permettant de retirer de l'argent liquide de son compte bancaire. Un **versement** est l'opération inverse, permettant de mettre de l'argent liquide sur son compte bancaire. Les deux opérations nécessitent de passer par un **guichet automatique bancaire** ou par un employé de la banque. Dans les deux cas, il est obligatoire d'utiliser sa carte bancaire comme moyen d'identification et d'autorisation.

Un **virement** est une opération permettant de transférer de l'argent d'un compte source à un compte cible. Selon le type de compte, le titulaire du compte, l'institution financière, le pays, le devise du compte, et de nombreux autres facteurs, des frais bancaires peuvent s'appliquer sur un virement. Pour des virements réalisés entre comptes en euros dans la **zone SEPA**<sup>4</sup>, aucun frais bancaires ne sont appliqués. Un virement entre comptes en différentes devises (EUR, USD, GBP, CAD, ...) est sujet à un taux de change et parfois à une commission supplémentaire. Un virement hors zone SEPA est sujet à des frais d'émission et de réception, qui peuvent être à la charge entière du destinataire, de l'émetteur, ou répartis entre les deux selon l'option choisie lors de la réalisation du virement.

---

2. IBAN = International Bank Account Number

3. BIC = Bank Identification Code

4. SEPA = Single Euro Payments Area, couvrant actuellement les états membres de l'Union Européenne ainsi que la Suisse, le Liechtenstein, Monaco, la Norvège et l'Islande.

## 1.4 Exigences technologiques

Un des défis du projet consiste en un choix judicieux des technologies, frameworks et librairies Java qui vous aideront dans votre travail. C'est votre choix, et c'est à vous de vous documenter sur le meilleur choix, et justifier ce choix dans le rapport d'implémentation.

Pensez-bien à la facilité d'installer et d'évaluer votre application par les enseignants. L'application doit être multi-plateformes, donc il faut éviter les technologies qui ne fonctionnent que sur un système d'exploitation spécifique (par exemple Windows), ou qui nécessitent des outils ou licences particulières. L'application devrait être installable et exécutable sans aucun soucis sur des machines tournant sous Windows, Linux et Mac OS X.

### 1.4.1 Architecture REST

Au niveau technologique, il est imposé d'utiliser une architecture "RESTful" de type **client-serveur**. REST correspond à un ensemble de contraintes architecturales, et il y a de nombreuses façons de réaliser une telle architecture.

Toutes les données concernant les institutions financières, les titulaires des produits financiers offerts, et l'historique des produits financiers associés aux titulaires seront hébergés dans une base de données sur un serveur. Les applications clientes (qui seront décrites dans la section 1.5) ne stockeront pas de données en local, sauf les données qui ne pourront pas être partagées avec d'autres applications pour des raisons de confidentialité, sécurité ou autres. Les applications clientes seront donc principalement de simples "consommables" de données hébergées sur le serveur. Cependant, rien n'empêche l'utilisation d'un cache local sur les applications si cela permet d'augmenter la performance des applications.

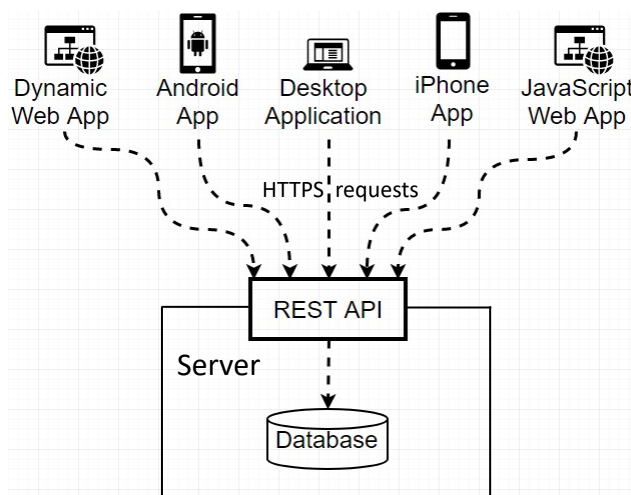


FIGURE 1 – Architecture "RESTful" de type client-serveur.

Au niveau de l'architecture du système, nous distinguons les composants visualisés dans la figure 1 :

- Un composant de type **serveur** en **Java** qui s'occupera de l'aspect gestion de données, permettant de stocker ou récupérer des données d'une **base de données**. À part de la gestion des données, il est probable que certaines fonctionnalités du système à réaliser doivent être implémentées sur le serveur. Vous êtes libre dans les choix technologiques pour gérer les données (par exemple, base de données SQL, NoSQL, ou n'importe quel autre format), étant donné que cette gestion sera totalement transparente pour les applications clientes qui communiqueront avec le serveur.
- Une **API** (Application Programming Interface) de type **REST** (REpresentational State Transfer) cachera les détails du serveur, et permettra à n'importe quelle application cliente de manipuler les données par l'API qui lui est fournie. L'interaction entre l'application cliente et le serveur se fera par

le protocole **HTTPS** (secure hypertext transfer protocol), en utilisant des méthodes paramétrées de type GET, POST, PUT et DELETE.

La communication entre les clients et le serveur doit être “sans état” (anglais : *stateless*), impliquant que le serveur ne stockera aucune information cliente entre plusieurs requêtes : chaque requête sera indépendante et totalement déconnectée des requêtes précédentes.

- Les données structurées qui seront transférées entre clients et serveur devront être encodées de préférence en format **JSON** ou **YAML** à cause de leur format qui est à la fois structuré et lisible par des humains.<sup>5</sup>. Des exemples de fichiers YAML sont disponibles aux figures 2, 3 et 4. Notez qu’il s’agit bien d’exemples purement illustratifs : nous ne donnons aucune garantie sur la pertinence des formats présentés ici, ni de leur contenu ; c’est aux membres du groupe de réfléchir à la manière d’organiser et de structurer les données.
- Plusieurs composants de type **client**, qui seront des applications desktop en **Java** pour réaliser les fonctionnalités de base décrites en section 1.5. Les applications ne sont pas autorisées à communiquer directement entre elles. Toute communication se fait par l’intermédiaire du serveur par le biais de son API, en respectant l’architecture REST.
- Pour réaliser certaines extensions de la section 1.6, d’autres composants de type application client sont envisageables. Chacune de ces applications interagira de la même manière avec le serveur.

Puisque le système à réaliser nécessite plusieurs applications qui communiqueront toutes avec le même serveur via une API, il convient de décider quelle partie de la logique devrait se trouver du côté serveur (pour éviter que chaque application doive implémenter la même logique) ou plutôt du côté client (pour éviter que le serveur ait une charge de travail trop importante).

```
portefeuille:
  client-ID: 00732643 02
  client-name: Tom Mens
  portefeuille-ID: P101
  language: English
  BIC: BBRUBEBB
  financial-institution: ING Belgium
  accounts:
    #compte courant
    - type: current-account
      IBAN: BE61 3101 2698 5517
      currency: EUR
    #compte épargne
    - type: savings-account
      IBAN: BE12 1234 1234 1234
      currency: EUR
    #compte titre
    - type: securities-account
      IBAN: BE97 3630 2113 3865
      currency: USD
  cards:
    #carte de débit
    - type: debit
      card-ID: 6703 3032 00302301 6
    #carte de crédit de type VISA
    - type: VISA credit
      card-ID: 4801 2411 2222 3333
```

FIGURE 2 – Exemple d’un fichier YAML représentant un portefeuille financier d’un client particulier, incluant l’ensemble de ses comptes et cartes liés à une institution financière particulière.

5. Avec l’accord des enseignants, et sur demande, bien justifiée, d’autres formats de données comme le **XML** pourront également être acceptés



```

transaction-history:
  client-ID: 00732643 02
  IBAN: BE61 3101 2698 5517
  extraction-date: 2021-09-01
  balance: 3120,24 EUR
  extraction-period: 30 days
  transactions:
    - type: outgoing-transfer
      amount: 200,00 EUR
      destination: BE97 3630 2113 3865
      date: 2021-08-31
    - type: withdrawal
      amount: 300,00 EUR
      ATM-ID: 23678
      date: 2021-08-27
    - type: Bancontact payment
      amount: 150,00 EUR
      destination: FNAC Bruxelles
      date: 2021-08-23
    - type: Maestro payment
      amount: 99,95 EUR
      destination: Proximus
      date: 2021-08-15
    - type: domiciliation
      amount: 123,00 EUR
      destination: Electrabel
      date: 2021-08-06
    - type: incoming-transfer
      amount: 1000,00 EUR
      source: BE12 1234 1234 1234
      date: 2021-08-02

```

FIGURE 3 – Exemple d’un fichier YAML représentant un extrait des transactions historiques liées à un compte particulier d’un client particulier.

Documentez-vous donc bien afin de faire un choix technologique le plus approprié. Voici quelques pointeurs potentiellement utiles :

- Representational state transfer  
[https://en.wikipedia.org/wiki/Representational\\_state\\_transfer](https://en.wikipedia.org/wiki/Representational_state_transfer)
- REST API Tutorial  
<https://restfulapi.net/rest-api-design-tutorial-with-example/>
- How to create a (basic) REST API in Java  
<https://happycoding.io/tutorials/java-server/rest-api>
- Top 10 Best Java REST and Microservice Frameworks  
<https://rapidapi.com/blog/top-java-rest-frameworks/>
- Building REST services with Spring  
<https://spring.io/guides/tutorials/rest/>
- Developing Restful APIs in Java Tutorial  
<https://www.youtube.com/watch?v=5jQSat1cKMo>
- JSON Tutorial <https://www.tutorialspoint.com/json/>
- Framework MVC pour des sites web Java (avec support pour REST/JSON)  
<https://struts.apache.org/>
- Hébergeur gratuit qui gère Java  
<https://www.alwaysdata.com/fr/>

```

institution:
  nom: ING Belgium
  BIC: BBRUEBBB
  siege:
    rue: Avenue Marnix
    numero: 24
    ville: Bruxelles
    code-postal: 1000
  clients:
    # a first bank client identified by his unique client ID
    - client-ID: 00732643 02
      IBAN-1: BE61 3101 2698 5517
      card-ID: 6703 3032 00302301 6
    # a second client without card but with 2 accounts
    - client-ID: 00301213 30
      IBAN-1: BE97 3630 2113 3865
      IBAN-2: BE12 1234 1234 1234

```

FIGURE 4 – Exemple d’un fichier YAML représentant les clients et comptes d’une institution financière.

### 1.4.2 Testabilité de l’application

Lors de la réalisation des fonctionnalités demandées, il est **obligatoire** d’offrir un moyen automatique aux enseignants (et à vous même!) pour tester et simuler toute fonctionnalité implémentée. Nous insistons sur l’utilisation d’un processus de **test-driven development**<sup>6</sup>. Tout au long du développement vous devez mettre en place des **tests unitaires**, avant et après chaque ajout de fonctionnalité ou chaque refactoring, ainsi que l’ajout des tests de régression chaque fois qu’un bug ou défaut est découvert. Les tests serviront comme spécification et documentation active et continue de votre application. Si un test échoue, alors cela implique qu’il y a un problème avec la fonctionnalité développée. Il est donc essentiel que des tests unitaires non triviaux et automatisés soient présents pour tester la fonctionnalité et le comportement de votre application.

Dans la programmation orientée objet, des **mock objects** sont des objets simulés qui imitent le comportement d’objets réels de manière contrôlée, le plus souvent dans le cadre de tests unitaires. La principale raison de créer de tels mock objects est de pouvoir tester une unité du système logiciel sans avoir à se soucier des modules dépendants. La fonctionnalité de ces dépendances est “simulée” par les mock objects. Ceci est particulièrement important si les fonctions simulées sont difficiles ou longues à obtenir (par exemple parce qu’elles impliquent des calculs complexes), si on désire tester l’interaction avec des composants auxquels on a pas accès lors du développement de l’application (par exemple un serveur distant), si le résultat est non déterministe, ou s’il est trop dangereux d’accéder à une base de données en production lors des test. Les mock objects sont aussi utilisés souvent pour simuler l’interaction avec une interface graphique.

Un autre aspect de la testabilité de votre application est que votre solution technique proposée doit être totalement transparente pour l’utilisateur du point de vue interaction avec la base de données (qui pourrait se trouver en local ou en ligne). Les enseignants doivent être capables d’exécuter et de tester l’application sans avoir à interagir directement avec la base de données. Par exemple, les enseignants ne devront pas installer un client SQL pour exécuter des commandes ou des scripts fournis pour configurer ou peupler la base de données. Les livrables fournis lors de la phase d’implémentation (commande gradle et fichier jar auto-exécutable) doivent suffire pour exécuter et tester l’application.

☛ Pour faciliter les tests de certaines fonctionnalités, il se peut que vous devriez mettre en place des fonctionnalités auxiliaires (accessibles aux tests unitaires) qui ne sont pas directement disponibles dans l’interface utilisateur (ou peut-etre uniquement par le biais d’une sorte de “mode debug”). Par exemple, si l’on veut tester qu’une certaine fonctionnalité se déclenche à une date précise (par exemple calculer la consommation

6. <http://www.agiledata.org/essays/tdd.html>

mensuelle à la fin du mois, ou calculer le rendement au début de l'année), on pourra mettre en place soit un système permettant d'accélérer le temps (par exemple une seconde correspondra à un jour), ou un système permettant de modifier la date interne de l'application.

## 1.5 Fonctionnalités de base

Les fonctionnalités décrites dans cette section doivent obligatoirement être implémentées par chaque groupe d'étudiants. Cependant, la collaboration entre différents groupes est strictement proscrite et sera assimilée à un plagiat !

Deux applications clientes différentes doivent être réalisées, chacune visant un type d'utilisateur distinct. La section 1.5.2 décrit une application pour un titulaire de comptes chez une (ou plusieurs) institutions financières. La section 1.5.3 décrit une application pour permettre à une institution financière de gérer tous les comptes (ou autres produits financiers) émis par cette institution pour ses clients.

Les deux types d'application partagent une même base de données hébergée sur un serveur. Dans cette base de données, tous les titulaires, comptes et institutions financières sont représentés de manière unique, et l'historique des transactions financières et de la gestion des comptes (y compris leur création et fermeture) est capturé. Les comptes seront identifiés par leur code IBAN, les institutions financières par leur code SWIFT/BIC, et les individus titulaires de compte par leur numéro de registre national.<sup>7</sup>

### 1.5.1 Internationalisation et gestion des langues

Les applications doivent supporter différentes langues, suivant les bonnes pratiques de développement Java<sup>8</sup>. Au minimum deux langues doivent être supportées (l'anglais et le français) avec la possibilité de facilement ajouter d'autres langues.

Lors de la création d'un login, chaque utilisateur doit choisir sa langue préférée. Il peut toujours changer ce choix. L'application utilisera alors cette langue après chaque connexion et utilisation de l'application par l'utilisateur.

### 1.5.2 Application titulaire pour gérer ses “portefeuilles financiers”

Pour l'application décrite ici, on suppose que l'utilisateur est déjà client chez une institution financière. L'application décrite dans la section 1.5.3 permet à l'institution financière d'ajouter de nouveaux clients.

**Portefeuille financier.** La première application permet à un client, titulaire de produits financiers chez une ou plusieurs institutions financières (typiquement des banques) de créer et de gérer son **portefeuille financier**. On entend par « portefeuille » l'ensemble des produits financiers émis par l'institution financière (par exemple, les comptes bancaires) pour lesquels le client est (co-)titulaire.

Dans la fonctionnalité de base, nous supposons que le client possède *un portefeuille différent pour chaque institution financière dans laquelle il a des produits financiers*. Nous supposons aussi qu'*un même portefeuille ne peut pas être partagé par plusieurs clients*. Dans les cas où un produit financier (par exemple un compte joint ou un compte indivis) possède plusieurs co-titulaires, chaque co-titulaire y aura accès par son propre portefeuille dans l'application, lui permettant d'accéder aux données des produits financiers dont il est (co-)titulaire.

Si le client possède plusieurs portefeuilles (chacun lié à une autre institution financière), l'application doit lui permettre de facilement passer d'un portefeuille à un autre, ou même d'avoir une vue d'ensemble sur l'entièreté de ses portefeuilles.

**Attention !** La notion de portefeuille est un concept qui est spécifique à l'application de gestion de portefeuilles financiers. Cette information est considérée confidentielle, et doit donc être invisible pour l'institution financière ou toute autre application client.

**Accès et authentification.** Cette application sera accessible à plusieurs titulaires. Chacun aura uniquement accès (par ses portefeuilles) aux produits financiers pour lesquels il est (co-)titulaire. Au lancement de l'application, l'utilisateur doit choisir entre se connecter avec un login existant ou en créer un nouveau. Le

7. Ce numéro est composé de 11 chiffres, dont les 6 premiers correspondent à la date de naissance de la personne.

8. Par exemple : <https://docs.oracle.com/en/java/javase/11/intl/> et <https://developer.android.com/training/basics/supporting-devices/languages>

login se base sur le nom complet et l'identifiant unique (numéro de registre national) de l'utilisateur, ainsi que d'un mot de passe alphanumérique choisi par l'utilisateur. Une fonction de réinitialisation de mot de passe doit également être présente. À tout moment pendant l'utilisation de l'application l'utilisateur peut décider de se déconnecter. Un autre client peut alors utiliser la même application pour gérer ses portefeuilles à lui.

**Création de portefeuille.** Au moment de la première connexion d'un utilisateur (après la création d'un nouveau login), aucun portefeuille ne lui sera accessible. Pour ajouter un portefeuille lié à une institution financière, l'utilisateur doit introduire une demande, en précisant l'institution financière où il est titulaire de certains produits financiers. Cette demande sera envoyée au serveur, et après vérification que l'utilisateur est bien client de cette institution financière, l'institution pourra approuver ou refuser la demande à l'aide de l'application décrite dans la section 1.5.3. Si la demande est approuvée, une permission sera activée dans la base de données sur le serveur, donnant ainsi à l'utilisateur accès à tous les produits financiers pour lesquels il est titulaire dans cette institution financière. À ce moment, le client verra apparaître dans son nouveau portefeuille tous les produits auxquels il a accès. Au minimum, tout portefeuille créé doit inclure le compte courant du client.

**Modification de portefeuille.** Le client peut toujours choisir de désactiver certains produits dans son portefeuille, rendant ainsi invisible les produits qu'il ne désire pas gérer. Il doit aussi avoir la possibilité de réactiver ses produits non-activés. Si le client souscrit à d'autres produits financiers auprès de l'institution financière ou si cette institution décide de supprimer certains produits pour lesquels le client est titulaire, alors le client doit être automatiquement averti au moment où il ouvre son portefeuille. Les nouveaux produits deviendront automatiquement disponibles, et les produits supprimés seront "archivés" (afin de permettre au client d'avoir une trace de l'historique, même pour des produits qui n'existent plus).

La souscription à de nouveaux produits financiers doit être possible à partir de l'application de gestion de portefeuille. Par exemple, l'application doit permettre à l'utilisateur de demander l'ouverture et la fermeture de comptes financiers pour lesquels il est titulaire.

**Virements.** En sélectionnant un compte dans son portefeuille, l'application doit permettre au titulaire du compte d'effectuer des virements sans nécessiter une carte bancaire.

- Pour avoir accès à cette fonctionnalité de virement, une demande doit être introduite (par l'application) à l'institution financière qui gère le(s) compte(s) du titulaire. La demande sera envoyée au serveur, et l'institution financière pourra approuver ou refuser la demande en se servant de l'application décrite dans la section 1.5.3. Si la demande est approuvée, la permission sera activée dans la base de données, donnant au client accès à la fonctionnalité de virement.
- Pour effectuer un virement, l'utilisateur doit sélectionner un de ses comptes, préciser le compte du destinataire et préciser le montant à transférer. Optionnellement, l'utilisateur peut fournir une communication en texte libre ou structurée<sup>9</sup>, et/ou une date mémo si le virement doit être effectué à une date ultérieure.
- Uniquement des virements entre comptes en euros dans la zone SEPA doivent être supportés dans la fonctionnalité de base. Chaque virement sera soumis à une vérification de sécurité (par exemple, mot de passe, code PIN, ou autre moyen de sécurité). Le niveau de vérification de sécurité peut différer selon si le client souhaite effectuer des virements au sein d'un même portefeuille (donc entre comptes du même titulaire), entre comptes de la même institution financière mais appartenants à des titulaires différents, entre comptes dans le même pays, etc...

**Historique et visualisation.** Pour un portefeuille donné, le client doit avoir la possibilité de visualiser les données historiques pour n'importe quel sous-ensemble de produits financiers contenus dans son portefeuille, et ceci à différentes granularités temporelles : journalière, hebdomadaire, mensuelle ou annuelle. La

9. Voir [https://www.monastucesetconseils.be/2004-06/un-virement-et-une-communication-structuree-WAACACAR\\_EU112002](https://www.monastucesetconseils.be/2004-06/un-virement-et-une-communication-structuree-WAACACAR_EU112002)

visualisation doit se faire sous forme de listes et tableaux, ainsi que sous forme de graphes (par exemple : line charts, pie charts, bar charts, box plots, spider charts, violin plots, ...).

Le client doit aussi avoir la possibilité de réaliser les mêmes types de visualisation pour l'ensemble de ses portefeuilles. Cette visualisation doit être flexible, permettant au client de sélectionner les produits qu'il souhaite visualiser, ainsi que la période de visualisation. (Par exemple, le client pourra décider de visualiser l'historique de l'ensemble de ses comptes d'épargne pour la période allant de janvier 2020 jusqu'à décembre 2021.)

L'utilisateur doit être capable d'exporter les données ainsi visualisées vers des fichiers en format CSV (comma-separated values), ainsi qu'au moins un des formats suivants : JSON ou YAML.<sup>10</sup>

■ **Du point de vue technique, plusieurs librairies vous permettent de gérer les différents formats de fichiers avec des parseurs dédiés. De la même façon, plusieurs librairies graphiques vous permettent de réaliser les visualisations. Le choix de la librairie la plus appropriée est laissé à la discrétion de chaque groupe. Il est autorisé de discuter entre groupes d'étudiants concernant le choix des librairies les plus appropriées.**

### 1.5.3 Application pour une institution financière

L'application décrite ici fournira une interface pour les institutions financières. L'application doit permettre à une institution d'ajouter de nouveaux clients, de supprimer de clients existants, et de gérer l'ensemble des produits financiers de ses clients.

Si une personne est client chez plusieurs institutions financières, chaque institution aura uniquement accès à ses propres produits financiers liés à ses clients pour des raisons de confidentialité. Une institution n'aura pas accès non plus à la notion de "portefeuille", car les portefeuilles font partie uniquement de l'application de gestion de portefeuilles décrite en section 1.5.2.

Par simplicité, nous supposons qu'il y a un seul login unique par institution financière dans l'application. Ceci évite la création d'un compte individuel pour chaque employé de l'institution financière. Tout comme dans l'application de la section 1.5.2, le login est protégé par un mot de passe, qui peut être changé dans l'application.

Un employé d'une institution financière qui est connecté à l'application aura l'autorité de créer (ou supprimer) des clients, d'ouvrir de nouveaux produits financiers pour un des clients de l'institution (par exemple, la création d'un nouveau compte courant pour un client) et de clôturer des produits existants (par exemple, fermer un compte épargne d'un client). Cette création ou clôture de produits pourra se faire soit directement, soit suite à une demande venant du serveur, déclenchée par un client ayant introduit la demande par le biais de l'application de la section 1.5.2. De la même façon, l'application doit permettre à l'institution financière d'approuver ou de refuser des demandes d'activation de la fonctionnalité "virement" pour un client (voir section 1.5.2).

**Historique et visualisation.** Un utilisateur de l'application peut lister l'ensemble des clients de l'institution financière, et l'ensemble des produits financiers associés à ses clients. Il peut aussi filtrer ces données selon plusieurs critères, comme le nom partiel du client, le type du produit financier, le nom partiel ou code unique du produit financier, la date de création du produit financier, etc...

**Import/Export.** Les données ainsi listées pourront être exportées vers des fichiers en format CSV (comma-separated values), ainsi qu'au moins un des formats suivants : JSON ou YAML. Il sera également possible de peupler la base de données (hébergée sur le serveur) à partir de l'application, en important des fichiers de données, toujours en s'assurant que les fichiers ne contiennent que de l'information appartenant à l'institution financière qui est actuellement connectée à l'application.

---


10. Une exportation en format XML pourra être envisagée également, mais c'est optionnel.

## 1.6 Fonctionnalité des extensions

Chaque groupe d'étudiants doit choisir un ensemble d'extensions différentes, en accord commun avec les enseignants. Chaque étudiant doit s'occuper d'implémenter son extension choisie, et réaliser une application exécutable intégrant la fonctionnalité de l'extension à la fonctionnalité de base. Un exécutable indépendant sera donc produit par chaque étudiant pour l'extension qui lui a été attribuée.

### 1.6.1 Gestion de cartes

L'objectif de cette extension consiste à ajouter du support pour les **cartes bancaires** dans les deux applications réalisées pour la fonctionnalité de base. La liste de fonctionnalités proposées ici est volontairement incomplète et peu détaillée, afin de laisser de la flexibilité lors de la réalisation de cette extension.

**Terminologie.**  La terminologie introduite ici est nécessaire pour une meilleure compréhension de l'énoncé de cette extension. Pour en savoir plus, voir [https://fr.wikipedia.org/wiki/Carte\\_de\\_paiement](https://fr.wikipedia.org/wiki/Carte_de_paiement) ou de nombreuses autres sources sur Internet.

Une **carte bancaire** est émise par une institution financière (typiquement, une banque) et appartient à un et un seul titulaire. Une même personne peut être titulaire de plusieurs cartes bancaires. La carte bancaire est liée à un compte courant appartenant à son titulaire. En cas d'un compte joint, chaque co-titulaire peut avoir sa propre carte bancaire liée au même compte.

Pour régler des achats avec son compte courant, le titulaire peut disposer d'une **carte de débit** (typiquement une carte de type Bancontact ou Maestro en Belgique) qui sera liée à un de ses comptes courant. Lors d'un achat le montant sera directement déduit de son compte.

Une personne peut aussi être titulaire d'une **carte de crédit** (par exemple Visa, American Express ou Mastercard) liée à son compte courant. Avec cette carte, le titulaire reçoit une certaine somme de crédit (disons 1,500 € par défaut) dont il peut disposer chaque mois en utilisant la carte. A la fin du mois, la somme dépensée avec la carte de crédit sera retirée du compte courant.

Chaque carte bancaire possède une date de fin de validité et un numéro unique, différent de toute autre carte bancaire. Ce numéro contient typiquement entre 14 et 19 chiffres, selon l'émetteur de la carte. Il y a également un code de sécurité (CSC) ou de vérification (CVC) de typiquement trois chiffres au dos de la carte.

**Énoncé.** Une carte bancaire est toujours liée à un seul titulaire, et est identifiée par un numéro unique. La carte est associée à un et un seul compte courant qui sera débité lors de son utilisation. L'application de gestion de portefeuille doit permettre de :

- associer les cartes bancaires que l'utilisateur possède à ses portefeuilles
- demander une carte pour la première fois à son institution financière ; ou demander l'arrêt d'utilisation d'une carte
- gérer au moins 2 types de cartes de débit différents (au minimum Bancontact et Maestro) et au moins 3 types de cartes de crédit différents (au minimum Visa, American Express et Mastercard) ; ceci inclut la possibilité d'effectuer des achats par carte de crédit
- tenir compte des frais d'utilisation de la carte (frais mensuel ou annuel payé à la banque ; frais de commission quand on utilise la carte pour des transactions à l'étranger ; ...)
- utiliser la carte bancaire comme moyen de se connecter à l'application, en simulant un lecteur de carte ; il suffit de proposer une interface utilisateur différente pour l'authentification, faisant croire à l'utilisateur de l'application qu'il se connecte par carte (par exemple, en choisissant le numéro de carte pour simuler l'insertion d'une carte bancaire, et en introduisant le code PIN associé à la carte pour se connecter)
- permettre de visualiser et d'exporter l'historique d'utilisation des cartes bancaires. La visualisation doit être disponible sous forme de listes et tableaux, ainsi que sous forme de graphes (similaire à ce qui est possible pour visualiser l'historique des autres produits financiers dans un portefeuille)
- vérifier la date de fin de validité et interdire que la carte soit utilisée après cette date

- gérer la procédure de renouvellement automatique : si une carte est proche de sa date de fin de validité, l'institution financière créera automatiquement une nouvelle carte qui remplacera la carte actuelle dans les deux applications. Au moment de la première utilisation de cette nouvelle carte dans l'application client, celui-ci doit explicitement valider son utilisation. L'ancienne carte est automatiquement invalidée à ce moment.
- bloquer et demander une nouvelle carte en cas de perte ou de vol : dans les deux cas, l'institution financière en sera informée (en passant par le serveur) dans l'application pour l'institution financière, et un employé de l'institution financière désactivera toute utilisation de la carte perdue ou volée, et lancera la procédure de renouvellement de carte (cf. point précédent)
- modifier les paramètres de la carte : L'application doit permettre au titulaire d'augmenter ou diminuer le seuil mensuel du crédit accordé, soit de manière temporaire ou permanente. Elle doit également permettre d'activer/désactiver l'utilisation de la carte à l'étranger et d'autoriser/interdire des transactions qui donneront lieu à un solde négatif sur le compte (pour éviter des rendements négatifs). Certaines de ces configurations doivent nécessiter une validation de la part de l'institution financière, par le biais de l'application correspondante.



### 1.6.2 Gestion de devises et virements internationaux

L'application de base ne prévoit pas d'autres devises que euros (EUR). Cette extension se charge d'étendre la fonctionnalité de base avec la gestion de différents types de devises. Au moins 5 devises différents doivent être supportées, dont euros (EUR), dollars (USD), et livres sterling (GBP). La gestion de devises impliquera des changements dans plusieurs aspects de la fonctionnalité de base :

- Il doit être possible par l'application de gestion de portefeuilles de suivre l'historique des taux des devises, et de calculer le taux de conversion entre devises. En effet la valeur relative d'une devise par rapport à une autre peut varier de jour en jour.<sup>11</sup>
- Tous les types de comptes de la section 1.3 doivent être supportés, et il doit être possible de créer et d'utiliser des comptes dans d'autres devises. Il doit être possible de gérer des comptes bancaires multidevises.
- La fonctionnalité de virement (faisant partie de la fonctionnalité de base) doit être étendue en tenant compte de plusieurs facteurs pouvant engendrer un coût supplémentaire ou une durée de traitement plus importante. Si le compte source et le compte destination utilisent une devise différente, alors un taux de conversion sera comptabilisé. Ce taux de conversion peut varier de jour en jour. Les frais de conversion resteront cependant moins élevé quand il s'agit d'un virement entre différents compartiments d'un compte multidevises.

Si on effectue un virement entre comptes de différents pays, tout en restant en zone SEPA, le virement ne sera pas instantané (le traitement peut prendre plusieurs jours).

Si le virement est vers un compte hors zone SEPA, il est obligatoire de préciser l'adresse complète du destinataire, ainsi que le numéro SWIFT de la banque du destinataire (qui doit être située hors zone SEPA).<sup>12</sup> L'utilisateur doit être bien informé du fait que des frais de commission sur la transaction seront facturés.<sup>13</sup> Il pourra préciser si les frais seront payés par lui même, ou par le destinataire, ou partagés entre les deux parties.

L'application pour institution financière devra également être adaptée. Chaque institution sera liée à un pays et une devise principale (qui sera utilisée par défaut). Les comptes dans une institution hors zone SEPA seront considérés comme des comptes hors zone SEPA pour les virements. Les fonctionnalités autres que les virements ne changent pas pour ces institutions.

---

11. Il existe des API pour obtenir les taux de conversion de devises, par exemple <https://www.exchangerate-api.com/docs/java-currency-api>.

12. Pour un virement en SEPA, uniquement le numéro IBAN du compte doit être précisé, le numéro BIC/SWIFT de la banque n'est pas obligatoire.

13. Les frais dépendront des institutions financières et des pays.

### 1.6.3 Gestion de placements

L'objectif de cette extension consiste à ajouter du support pour les **placements** dans les deux applications réalisées pour la fonctionnalité de base.

**Terminologie.** ☞ La terminologie introduite ici est nécessaire pour une meilleure compréhension de l'énoncé de cette extension.

Un client souhaitant investir dans des produits boursiers, doit obligatoirement souscrire à un **compte titre**. C'est un type de compte permettant de gérer les **produits boursiers**, aussi appelés des **placements** financiers. Les placements auxquels on a accès dépendent de la devise du compte titre. Avec un compte titre en EUR on peut souscrire à des placements en €. Avec un compte titre en USD on peut souscrire à des placements en dollars. Un **compte titre multidevises** combine les possibilités en créant un compartiment par devise.

Le titulaire du compte titre peut passer des ordres de placements sur le marché financier. Cette section décrit les différents types de placements que le titulaire peut associer à son compte titre.

Les types de placement auxquels le titulaire peut avoir accès dépendent de sa **perception du risque**, de son **horizon d'investissement** et de sa situation financière. Ceci est défini par un **profil d'investisseur** qui doit obligatoirement être établi (en remplissant un formulaire) pour chaque titulaire d'un compte titre désirant effectuer des placements. Les cinq profils sont listés dans la table 2.<sup>14</sup>

Profil	Description
Conservateur	Perception du risque : Vous ne voulez prendre que très peu de risques. Horizon d'investissement : au moins 3 ans. Rendement : En protégeant au maximum votre capital, vous acceptez un rendement minimal. Votre portefeuille ne grandira que de façon très limitée.
Défensif	Perception du risque : Vous souhaitez prendre un peu de risques. Vous acceptez que la valeur de votre investissement puisse fluctuer, mais de façon limitée. Horizon d'investissement : au moins 4 ans. Rendement : En prenant un peu de risques vous augmentez le rendement potentiel de votre portefeuille.
Neutre	Perception du risque : Vous comprenez que vous courez le risque de perte éventuelle du capital investi et vous acceptez que la valeur de votre portefeuille puisse fluctuer fortement. Horizon d'investissement : au moins 5 ans. Rendement : Vous visez un bon équilibre entre risque et rendement.
Dynamique	Perception du risque : Vous ne craignez pas de prendre de vrais risques. Vous comprenez que votre portefeuille peut être sujet à de fortes fluctuations de valeur et subir des pertes considérables. Horizon d'investissement : au moins 6 ans. Rendement : En prenant plus de risque, vous souhaitez réaliser un très haut rendement.
Agressif	Perception du risque : Vous osez prendre des risques importants. Vous comprenez que votre portefeuille peut être sujet à de très fortes fluctuations de valeur et qu'il peut subir des pertes majeures. Horizon d'investissement : au moins 7 ans. Rendement : Vous souhaitez réaliser un rendement maximal.

TABLE 2 – Types et caractéristiques des profils d'investisseur.

On peut toujours **acheter ou vendre des placements** correspondant à son profil d'investisseur. Un coût sera associé à chaque achat et vente. Ce coût dépendra du produit financier, de l'émetteur du produit financier, de l'institution financière qui gère vos placements, des règles fiscales, et de nombreux autres facteurs.

Les types de placements sont résumés dans la Table 3. À chaque type de placement est lié un **code ISIN** unique, composé de 12 caractères alphanumériques. La page Wikipedia donne les détails sur la composition d'un numéro ISIN [https://fr.wikipedia.org/wiki/ISO\\_6166](https://fr.wikipedia.org/wiki/ISO_6166). Si vous souhaitez obtenir davantage de détails, veuillez consulter d'autres sources sur Internet.

Un **bon de caisse** est un type de placement à durée fixe (maximum 5 ans), généralement établi par un établissement financier. Il se présente sous le forme d'un bon représentatif d'un dépôt productif d'intérêts. Il est comparable à un compte à terme. Juridiquement l'établissement émetteur du bon de caisse devient le débiteur du titulaire du bon, lequel est remboursé à son échéance. Les intérêts convenus sont soit payés

14. Ceci n'est qu'un exemple, les noms des profils peuvent varier selon l'institution financière.

Type de placement	Caractéristiques
Bon de caisse	Un placement à durée fixe de maximum 5 ans, comparable à un compte à terme. Risque : très faible Durée : Fixe avec un maximum de 5 ans Rendement annuel sur 5 ans : fixe, typiquement entre 0,1 et 0,5% Fiscalité : précompte de 30% sur les intérêts
Obligation	Un titre de créance correspondant à une fraction de la dette d'une entreprise. Risque : moyen, dépend du risque de défaillance de l'entreprise. Durée : fixe, mais vous pouvez toujours décider de vendre vos obligations avant la fin de la durée, selon les taux du marché. Rendement : fixe, et typiquement plus faible que le rendement sur les actions.
Action	Un droit de propriété d'une partie d'une entreprise. Risque : (très) élevé Durée : variable, mais un investissement à long terme réduit les risques liés à la volatilité de la bourse financière. Rendement : variable. À part du dividende annuel versé par l'entreprise à ses actionnaires, le rendement n'est pas borné car cela dépend de la volatilité de la bourse, et peut même être négatif.
Fonds de placement	Un <b>fonds d'investissement</b> correspondant à une gestion collective de produits financiers. Risque : dépend du contenu du fonds Durée : variable, comme les actions. Rendement : variable, comme les actions.
Fonds d'épargne pension	Un investissement dans un fond de placement, mais offrant des avantages fiscaux spécifiques.
Tracker (ETF)	Un fonds de placement qui réplique une indice boursier en suivant fidèlement son évolution.

TABLE 3 – Types de produits boursiers.

d'avance, auquel cas ils sont déduits du capital placé en tenant compte du temps restant jusqu'à la date du remboursement, soit payés à l'échéance en même temps que le capital placé.

Une **obligation** est un titre de créance correspondant à une fraction de la dette d'une entreprise ou d'un Etat. En contrepartie, le détenteur de l'obligation va percevoir des intérêts. Ces intérêts vont déterminer un **coupon** périodique sur une durée préétablie, après quoi il sera remboursé à échéance du capital prêté initialement.

Une **action** est un placement par lequel vous devenez actionnaire et donc propriétaire d'une partie de l'entreprise. L'actionnaire a droit à une partie des profits réalisés par l'entreprise, par le biais d'un **dividende** par action. Ce dividende varie chaque année en fonction de la capacité de l'entreprise à générer des profits. Vous pouvez toujours acheter et vendre vos actions sur la bourse financière, la valeur dépendra de la volatilité de la bourse, ce qui peut constituer un risque assez élevé. Il s'agit donc plutôt d'un investissement à long terme.

Un **fonds de placement** ou **fonds d'investissement** correspond à une gestion collective de produits financiers. L'intérêt de la gestion collective est de bénéficier de gestionnaires de placements professionnels et d'économies d'échelle (coûts de transaction moins élevés). Un actionnaire d'un fonds de placement deviendra propriétaire d'une partie de ce portefeuille de produits financiers. Un fonds de placement peut être composé d'obligations uniquement (**fonds d'obligations**), d'actions uniquement (**fonds d'actions**), ou même combiner les obligations, les actions et de l'argent liquide selon une certaine répartition qui peut varier au cours du temps (**fonds mixte**). Le risque financier associé varie selon le type de fonds. À chaque fonds est également lié des coûts d'entrée (frais d'achat), de sortie (frais de vente) et des frais de gestion. Des fonds de placement sont disponibles pour les divers profils d'investisseur.

Un **fonds d'épargne pension** est un fonds de placement permettant d'optimiser vos impôts en récupérant jusqu'à 315 € par an. En effet, si vous épargnez jusqu'à 980 € par an, vous profiterez d'un avantage fiscal de 30% sur les versements effectués. Vous pouvez aussi choisir d'épargner annuellement jusqu'à 1260 €. Vous aurez alors droit à un avantage fiscal de 25%.

Un **tracker** ou **ETF** (pour Exchange Traded Fund) est un fonds de placement qui vise à répliquer un indice boursier en suivant fidèlement son évolution. En investissant dans un ETF, vous bénéficiez alors d'un investissement diversifié à moindres frais que pour un fonds de placement traditionnel qui nécessite un suivi plus actif par les gestionnaires du fonds.

**Énoncé.** Cette extension implique l’ajout des **comptes titre** comme nouveau type de compte bancaire au sein des portefeuilles financiers. Un compte titre permet à son titulaire de gérer un ou plusieurs produits boursiers émis par la même institution financière. Le compte titre doit toujours être lié à un compte courant associé au même titulaire dans la même institution financière.

Lors de la création d’un compte titre, le client doit préciser son **profil d’investisseur** (voir table 2), en remplissant un formulaire de questions. Ce formulaire sera envoyé au serveur, afin d’être validé par l’institution financière. En fonction des réponses aux questions, l’institution attribuera un des 5 profils au client. Le client peut toujours changer son profil d’investisseur en répétant cette procédure.

Sur base de son profil d’investisseur, le client peut sélectionner et acheter de nouveaux placements afin de les ajouter dans son compte titre. Uniquement les produits financiers compatibles avec le profil d’investisseur seront accessible et visible au client. Par exemple, dans un profil neutre, il sera pas possible d’inclure des placements dynamiques ou agressifs, mais il est possible d’inclure des placements conservateurs ou défensifs. Sur base du profil d’investisseur, le contenu du compte titre peut être comparé avec un “portefeuille de référence” c.à.d. un compte titre fictif correspondant à ce profil d’utilisateur. (Par exemple, un portefeuille dynamique contiendra surtout des actions et autres placements à risque élevé ; alors qu’un portefeuille défensif contiendra surtout des obligations et bon de caisse avec un risque relativement faible.)

Les placements dans le compte titre du client doivent être classifiés par type de placement. Pour chaque placement, il faut stocker au minimum le nom, le code ISIN, la date d’achat, la date d’expiration (si applicable), la quantité, le prix d’achat, la valeur actuelle, le rendement, le poids relatif de ce placement dans le compte titre et dans le portefeuille du client.

Le client peut décider de vendre des placements qui se trouvent dans son compte titre. L’argent reçu après vente sera versé sur un des comptes du client.

Cette extension de gestion de placements doit permettre le suivi de l’évolution du rendement et des coûts des placements se trouvant dans le compte titre du client. Le client doit aussi avoir la possibilité de voir l’historique des placements qui ne se trouvent pas encore dans son portefeuille, mais qu’il souhaite acheter. Pour obtenir l’historique de chaque placement, vous pouvez mettre en place une des deux solutions suivantes :

- un système aléatoire de génération de données pourra être implémenté au niveau du serveur, tenant compte des paramètres définis pour chaque placement.
- un système d’accès à des services en ligne (par le biais d’un API) fournissant des données historiques sur les produits financiers. Plusieurs tels services existent (souvent payants), l’étudiant devraient chercher et choisir le service le plus approprié.

Peu importe la solution choisie, les données historiques doivent être stockées dans la BDD, et mises à jour à intervalle régulier.

Similaire à la fonctionnalité de base, cette extension doit permettre la visualisation du contenu et de l’évolution de la totalité ou d’une partie de son compte titre sous la forme de tableaux et graphes. Cette visualisation doit aussi être exportable sous plusieurs formats (comme c’est le cas pour la fonctionnalité de base).

#### 1.6.4 Gestion du budget financier

Cette extension vise à étendre l'application de gestion de portefeuille d'un client avec la possibilité de mieux suivre et gérer son budget. Cela inclut, entre autre, les fonctionnalités suivantes :

**Transactions récurrentes.** Le détenteur d'un portefeuille devrait être capable de programmer flexiblement des virements automatiques récurrents. Voici quelques exemples typiques :

- Le client pourra transférer automatiquement une certaine somme fixe de son compte courant vers son compte épargne à la fin du mois. Ce transfert automatique pourrait être conditionné, par exemple uniquement s'il y a encore suffisamment d'argent disponible sur le compte courant.
- Le client pourra automatiquement transférer de l'argent vers son compte courant chaque fois que celui-ci risque d'aller en négatif.

**Classification des transactions financières.** Le détenteur d'un portefeuille doit pouvoir classifier les transactions financières par catégorie. Chaque paiement effectué ou reçu pourra être classifié selon des catégories prédéfinies. Par exemple : santé, travail, vacances, sport, énergie, études, autres, ... L'utilisateur doit avoir la possibilité de modifier ces catégories et d'en définir d'autres. Chaque transaction doit être associée à une catégorie, soit de manière manuelle par l'utilisateur, ou sur base de règles simples et configurables. Chaque catégorie aura son propre code couleur et icône associé. Sur base de cette classification, la fonctionnalité de base de *historique et visualisation* doit être étendue en tenant compte de ces différents catégories de transactions.

**Analyse du budget.** L'application devrait fournir une interface visuelle permettant d'analyser les dépenses budgétaires du client. Cette fonctionnalité doit inclure :

- Le calcul des statistiques de base pour chaque séquence de données temporelle. Par exemple : le minimum et le maximum ; la moyenne et l'écart type ; la médiane, les quartiles et l'écart interquartile. Pour le calcul de ces statistiques, l'utilisateur doit pouvoir sélectionner les catégories de transactions (définies ci-haut) à prendre en compte ou exclure. Par défaut, l'ensemble des transactions est considéré.
- Des tendances temporelles du budget. Ceci inclut : (a) la saisonnalité (les dépenses par catégorie seront différentes pendant l'été que pendant l'hiver par exemple) ; (b) des tendances de croissance (par exemple, en comparant les dépenses de l'année précédente avec celles de l'année en cours ;
- La détection d'anomalies. Par exemple, est-ce que l'utilisateur dépense trop pour sa consommation d'énergie (eau, électricité et gaz) par rapport à un foyer moyen ? Est-ce que certains types de dépenses deviennent trop importantes par rapport à l'évolution historique du budget ? Par exemple, l'application pourra notifier le client si celui-ci commence à dépenser beaucoup plus d'argent que d'habitude pour sa facture d'électricité.

### 1.6.5 Gestion des contrats d'assurance

L'objectif de cette extension consiste à ajouter du support pour les **contrats d'assurance** dans les deux applications réalisées pour la fonctionnalité de base.

**Terminologie.** ☞ La terminologie introduite ici est nécessaire pour une meilleure compréhension de l'énoncé de cette extension.

La table 4 résume les principaux types d'assurance. Certains seront détaillés par la suite. Veuillez consulter Internet pour vous renseigner sur les détails des différents types d'assurance offerts sur le marché belge par les différents assureurs.

Type d'assurance	Caractéristiques
Assurance véhicule	Permet d'assurer votre véhicule.
Assurance familiale	Couvre les dommages que vous, vos enfants ou vos animaux domestiques pourriez causer à une tierce personne.
Assurance assistance	En cas de panne de voiture, ou pour couvrir les soins ou rapatriement en cas de maladie ou d'accident à l'étranger.
Assurance habitation	Pour protéger votre bâtiment/maison et/ou son contenu en tant que propriétaire, par exemple en cas d'incendie, inondation ou autres types de calamités.
Assurance hospitalisation	Pour prendre en charge vos frais médicaux et paramédicaux en cas d'hospitalisation ou maladie grave.
Assurance voyage	Couvre les frais en cas d'empêchement qui vous oblige à annuler votre voyage.
Assurance vie de type "branche 21"	Un placement avec rendement garanti et avec couverture décès. Risque : très faible Frais d'entrée entre 0 et 6% ; taxe d'assurance de 2% sur chaque versement Rendement annuel sur 5 ans : 1,6% brut Protégée par une garantie de dépôt jusqu'à 100.000 euros par personne et par assureur Durée : au moins 8 ans si on veut éviter le précompte mobilier de 30% sur les intérêts.
Assurance vie de type "branche 23"	Une assurance vie couplée à un fonds de placement. Rendement : dépend des performances du fonds de placement. Risque : élevé (dépend du fonds de placement) Frais d'entrée entre 0 et 6% ; taxe d'assurance de 2% sur chaque versement Durée : variable, et exonérée du précompte mobilier de 30% sur les intérêts.
Assurance épargne pension	Un contrat d'assurance vie qui est lié à un fonds d'épargne pension. Fiscalement avantageux et pas de taxe d'assurance de 2%.

TABLE 4 – Types d'assurance

Une **assurance véhicule** permet d'assurer votre véhicule motorisé (auto ou moto). Une assurance de type **responsabilité civile** est obligatoire en Belgique lors de l'utilisation d'un véhicule motorisé. Cette assurance peut être complétée par une **assurance omnium** pour couvrir les dégâts à votre véhicule, une **assurance conducteur** pour couvrir les dommages corporels, et une **assurance de protection juridique** en cas de litige. Il existe aussi des assurances véhicules non obligatoires (pour les vélos par exemple), notamment mais pas exclusivement contre le vol.

Une **assurance vie de type "branche 21"** (parfois appelé "assurance épargne") offre un rendement garanti, éventuellement complété d'une participation bénéficiaire. Cette dernière n'est pas garantie, et dépend des résultats de l'assureur. Les avoirs des assurances épargne de droit belge sont protégés par le Fonds de garantie à concurrence de 100,000 EUR par personne et par assureur. Les assurances épargne sont exonérées de précompte mobilier (30%) à partir de la huitième année. Pour cette raison fiscale, une assurance épargne est donc intéressante si vous pouvez vous passer de votre argent pendant au moins huit ans et si vous ne voulez prendre aucun risque avec votre capital.

Une **assurance vie de type "branche 23"** est un type de placement (produit boursier) qui couple une assurance à un fonds de placement. Contrairement à la branche 21, son rendement n'est pas garanti, mais dépend des performances du fonds de placement. Lorsque les marchés financiers se portent bien, le capital augmente. Et lorsqu'ils vont moins bien, le capital diminue.

Une **assurance épargne pension** combine un fonds d'épargne pension (voir sous-section 1.6.3) avec une assurance vie. Cela vous permet d'optimiser votre épargne fiscale et d'échapper à la taxe d'assurance

de 2%. Ce type d'assurance existe en "branche 21" ou "branche 23" ou encore une combinaison des deux formules (parfois appelée "branche 44").

**Énoncé.** Certaines institutions financières proposent des produits d'assurance à leur clients. Ici on suppose qu'un utilisateur est client d'une telle institution financière.

Le portefeuille financier du client doit contenir un compartiment spécifique listant toutes ses assurances. Ce compartiment sera vide si le client ne possède aucun produit d'assurance. À partir de ce compartiment, l'utilisateur doit être capable de :

- Lister toutes les assurances dont il est titulaire, en incluant la prime annuelle et la date de renouvellement annuel.
- Visualiser l'historique de toutes les assurances présentes dans son portefeuille. (Y inclus ses anciennes assurances qui ne sont plus actives.)
- Payer la prime annuelle des assurances auxquels il est inscrit, soit de manière automatique ou de manière manuelle.
- Annuler une assurance, à condition que la demande d'annulation soit faite au minimum 3 mois et 1 jour avant la date de renouvellement annuel de l'assurance.
- Modifier les paramètres d'une assurance. (Par exemple, changer son assurance auto en omnium, ou ajouter un conducteur supplémentaire ; augmenter le plafond assuré de sa maison ; ajouter une option d'assistance juridique à son assurance familiale ; etc...)
- Dans le cas d'un produit de type "assurance vie" ou "assurance épargne" : verser de l'argent supplémentaire dans ce produit (de manière automatique ou manuelle), ou retirer une partie de son argent de ce produit.
- Obtenir de l'information sur tous les types d'assurance fournis par l'institution financière (cf. tableau 4)
- S'inscrire à une nouvelle assurance, en payant la prime de l'assurance.
- Demander un devis pour un certain type d'assurance à plusieurs institutions financières. Cette demande sera introduite par l'utilisateur dans l'application client qui envoie la demande au serveur. L'application pour institutions financières, qui donne accès à tous les produits financiers, recevra la demande, récoltera tous les produits similaires offerts par différents institutions, et renverra l'information (via le serveur) à l'application client. Le client recevra alors une liste des différents produits correspondant à sa demande. Le client doit pouvoir consulter les détails de chaque produit et, s'il trouve l'un des produits intéressant, pourra décider de s'inscrire au produit (cf. point précédent)

### 1.6.6 Paiements et gestion de fraudes

L'objectif de cette extension consiste à ajouter du support pour simuler des paiements (avec ou sans contact, ainsi que par QR code), et un système de détection et de gestion de fraude.

**Paiement par QR code.** L'application client doit permettre de recevoir ou d'effectuer des paiements par le moyen de la génération et la lecture d'un QR code :

- L'utilisateur peut créer une demande de paiement par QR code, dans laquelle il précise le montant à payer, le numéro de compte sur lequel ce montant doit être versé, et un message précisant la nature du paiement. L'application génère alors une image contenant un QR code, et cette image peut être sauvegardée sur la machine sur laquelle l'application tourne. Chaque QR code doit également encoder sa date et heure de création.
- Pour simuler un utilisateur qui reçoit une demande de paiement par QR code, cet utilisateur peut télécharger un fichier contenant l'image d'un QR code. L'application client peut alors lire ce QR code et vérifier sa validité. Si le QR code est lu plus d'une heure après sa création, on considère que le code est expiré et ne peut plus être utilisé. De la même manière, le même QR code ne peut être utilisé qu'une seule fois. Après son utilisation, il devient automatiquement invalide. Si le QR code a été validé, l'utilisateur connecté à l'application client peut indiquer de quel compte le montant précisé dans le QR code doit être retiré. Après cela, un transfert d'argent (virement) sera effectué automatiquement vers le compte d'origine précisé dans le QR code.

**Simulation de paiement sans contact ou avec contact.** Au sein de l'application client, une interface graphique doit être ajoutée pour simuler des paiements avec ou sans contact. Pour faciliter cette simulation, on suppose que l'application dispose d'une liste de paiements à réaliser (par exemple, l'achat d'un livre dans un magasin en ligne, l'achat d'un pain dans une boulangerie, un paiement dans une station-service, l'achat d'un billet de train, la réservation dans une musée, ...). Cette liste de paiements peut être stockée dans un fichier externe (dans un format json ou yaml). L'utilisateur peut choisir un des paiements de cette liste, et ensuite préciser s'il désire payer avec ou sans contact. Dans le premier cas, un virement automatique sera effectué à partir de son compte courant. Dans le deuxième cas, l'utilisateur doit d'abord confirmer le paiement par son code PIN. Pour tous les paiements (avec ou sans contact) venant du même compte courant il faut tenir compte d'une limite par transaction, ainsi qu'une limite journalière, hebdomadaire et mensuelle qui ne peut pas être dépassée. Ces limites doivent être configurables dans les paramètres d'un client d'une institution financière, disponible uniquement dans l'application pour les institutions financières. Les limites sont typiquement plus strictes pour les paiements sans contact que pour les paiements avec contact.

**Gestion de fraude.** L'application doit mettre en place un système de détection de fraude. (Par exemple, en supposant qu'une personne tierce à eu accès à vos données bancaires, et essaie d'abuser votre compte pour effectuer des transactions suspectes.) La détection de fraude peut se baser sur des règles simples, mais un système d'apprentissage automatique peut être envisagé également si l'étudiant le désire. Pour mettre en place un système de détection de fraude, il faut d'abord étendre les métadonnées de chaque virement bancaire, afin de stocker également le mode de paiement (sans contact, par carte bancaire, par ordinateur), le lieu de paiement (dans un magasin réel, vente en ligne), le pays de paiement (en Belgique, en zone SEPA, à l'extérieur de la zone SEPA), la date et l'heure de paiement, ... Sur base de ces données, des règles simples pourront être mises en place pour détecter des suspicions de fraude :

- Une série rapide de plusieurs paiements successifs venant du même compte courant.
- Des paiements venant des pays différents dans un intervalle de temps limité.
- Des paiements correspondant à des valeurs inhabituelles (par exemple des montants très élevés).
- Des paiements vers des destinataires correspondant à une "liste noire" de clients frauduleux.
- ...

A chaque règle de suspicion de fraude sera associée un poids (certaines règles seront plus importantes que d'autres) et une probabilité (par exemple faible, moyenne ou forte suspicion). En cas de faible suspicion, le



client sera demandé d'utiliser son code PIN ou mot de passe pour toute nouvelle transaction, même pour les paiements sans contact. En cas de moyenne suspicion, la transaction sera bloquée. En cas de forte suspicion, le compte du client sera bloqué. Dans tous les cas de suspicion de fraude, l'application client notifiera le client de la suspicion, en demandant au client de confirmer ou d'infirmier l'utilisation abusive de son compte. (En cas de "fausse suspicion", les blocages éventuels seront retirés.)

Le système de détection de fraude devrait avoir un historique, afin de permettre de garder une trace de toutes les transactions suspectes, y inclus celles qui ont été confirmées ou infirmées par les clients. Cette historique doit être visualisable, et l'information historique devrait permettre au système de détection de s'améliorer en tenant compte de fausses ou réelles suspicions. (Par exemple, en diminuant le poids d'une règle qui a donné lieu à beaucoup de fausses suspicions par le passé.) L'historique peut aussi être utilisé pour augmenter le niveau de suspicion de fraude : si plusieurs transactions de faible suspicion sur un compte ce succèdent, alors la probabilité de suspicion devrait augmenter par un effet d'accumulation.

## 2 Exigences

Chaque groupe d'étudiants doit *conjointement* réaliser les **fonctionnalités de base** (section 1.5) du système logiciel à réaliser. Chaque membre du groupe doit *individuellement* réaliser une **extension** choisie parmi les extensions proposées dans la section 1.6. Les enseignants veilleront à une répartition équilibrée des extensions entre les étudiants et les groupes.

Les différentes phases du projet doivent toutes inclure la fonctionnalité de base et les extensions choisies. Le système logiciel contenant uniquement les fonctionnalités de base devrait être rendu par le groupe comme un exécutable à part entière. Chaque membre du groupe doit réaliser un autre exécutable correspondant à son extension choisie : cet exécutable doit alors inclure toutes les fonctionnalités de base, étendues avec les fonctionnalités qui font partie de l'extension.

☞ Lors de l'évaluation de chaque phase du projet, en supposant un groupe composé de 4 étudiants, la fonctionnalité de base comptera pour 60% de la note finale, et les extensions individuelles réalisées par chaque étudiant compteront chacune pour 40% de la note finale. (Pour un groupe de 3 étudiants, la pondération sera 70% / 30% et pour un groupe de 2 étudiants 80% / 20%.)

### 2.1 Étapes clés et livrables

Le travail sera décomposé en deux phases. Au terme de chaque phase, des livrables précis seront rendus. Chaque livrable sera déposé sur la plateforme Moodle<sup>15</sup> selon les critères de recevabilité imposés dans la section 4.

#### Phase 1 (en Q1) : Projet d'analyse et de conception (S-INFO-015)

Cette phase concerne la maquette et la modélisation du projet logiciel.

**Diagrammes de conception UML.** En se basant sur le cahier des charges, un modèle de conception sera réalisé en UML. Les diagrammes suivants doivent **obligatoirement** être utilisés :

- un diagramme de cas d'utilisation, en incluant des descriptions semi-structurées des scénarios pour chaque cas d'utilisation
- un interaction overview diagram précisant les interactions entre les cas d'utilisation
- les diagrammes de classes (en précisant la structuration en paquetage)
- les diagrammes de séquences décrivant des scénarios de comportement non-trivial.

Les diagrammes d'état comportementaux (statechart) et les diagrammes d'activités sont optionnels.

**Modèle de données.** Un modèle de données relationnelles doit être rendu, en utilisant les diagrammes d'entité-relation (ERD). Ce schéma doit préciser les différents types de données à manipuler par le système logiciel, ainsi que les attributs, les relations, et les clés primaires et étrangères. Le(s) modèle(s) de données doit(vent) couvrir les fonctionnalités de base ainsi que les fonctionnalités des extensions choisies. Un texte explicatif doit accompagner votre modèle, pour expliquer l'utilité des tables et la logique sous-jacente.

☞ Comment réaliser un modèle de données relationnelles fait partie des prérequis pour ce projet. À titre d'illustration, la figure 2.1 montre un exemple d'un diagramme d'entité-relation, créé avec l'outil Visual Paradigm, pour une application de location de vidéo.

**Design du REST API.** Étant donné que le système à réaliser se base sur une API REST, il est également demandé de fournir un schéma ou modèle de votre API rest. Vous avez la liberté de choisir comment réaliser ce schéma. Certains outils permettent de faire un design visuel de l'API REST. Un exemple est illustré à la figure 2.1, en utilisant l'outil de modélisation Visual Paradigm.

**Maquette de l'interface utilisateur.** Un rapport présentant les **maquettes de l'interface utilisateur** doit être rendu. Il s'agit d'un prototype non-fonctionnel de l'interface utilisateur illustrant de manière visuelle le design de la fonctionnalité de base et de chaque extension à réaliser. Cette maquette devrait présenter plusieurs visualisations illustrant les aspects importants

---

15. <https://moodle.umons.ac.be>

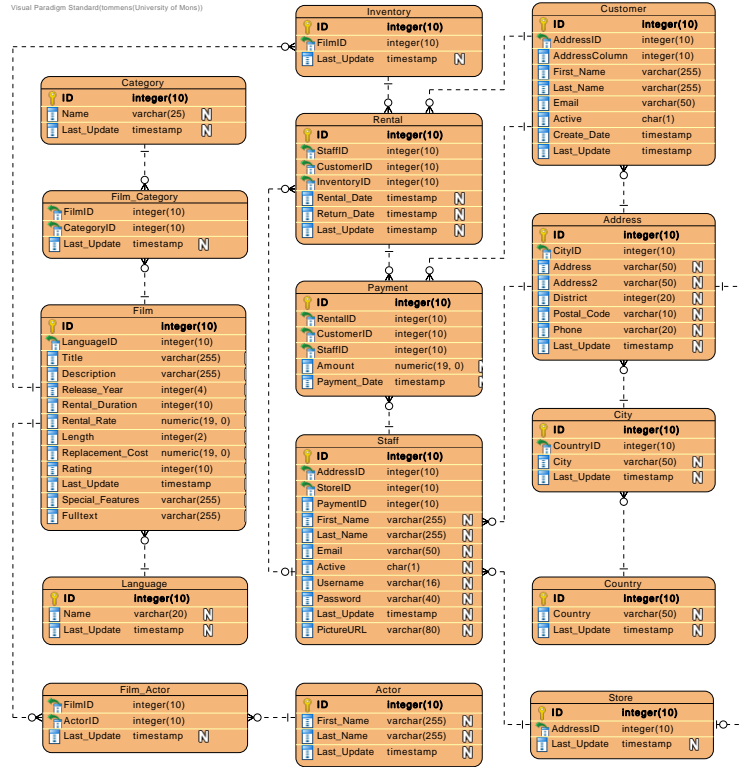


FIGURE 5 – Exemple d'un diagramme d'entité-relation pour représenter un modèle de données relationnelles.

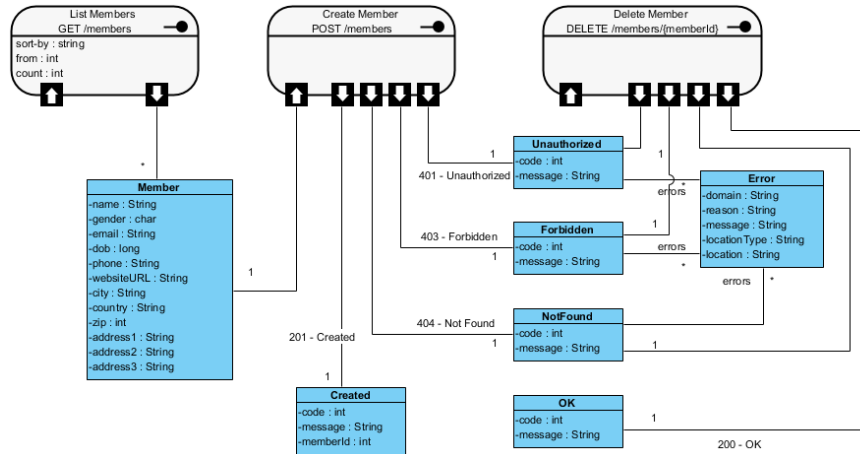


FIGURE 6 – Exemple d'un schéma visuel d'une API REST, réalisé avec l'outil Visual Paradigm.

- de l'interface visuelle pour la fonctionnalité de base ;
- de l'interface visuelle pour chacune des extensions choisies.

Les visualisations doivent être accompagnées d'un texte décrivant bien la façon d'interagir avec l'application afin de réaliser les fonctionnalités.

## Phase 2 (en Q2) : Projet de développement logiciel (S-INFO-106)

Cette phase concerne l'implémentation, les tests et le déploiement du projet logiciel.

**Code source.** Le projet logiciel sera implémenté en utilisant le langage de programmation Java 11. L'implémentation doit suivre les principes de l'*orienté objet*, en utilisant de manière correcte le mécanisme de l'héritage, et en utilisant des design patterns. L'implémentation doit également suivre une approche de *programmation défensive* en utilisant correctement le mécanisme de gestion d'exceptions de Java.

**Interface graphique.** L'interface graphique (GUI) des applications clients (pour les titulaires et pour les institutions financières) doit être réalisé avec JavaFX 11 (<https://openjfx.io>).

**Documentation.** Le code Java doit être documenté avec Javadoc.<sup>16</sup>

**Tests unitaires.** Nous insistons sur l'utilisation d'un processus de *test-driven development*<sup>17</sup>, en utilisant obligatoirement les *tests unitaires* en JUnit tout au long du développement. L'utilisation d'un framework de *mock objects* pour compléter les tests unitaires est fortement recommandée. (Voir la section 1.4.2.)

## 2.2 Organisation et conditions du travail

**Gestion de versions.** Les groupes doivent obligatoirement utiliser un système collaboratif de contrôle de versions (git) pour gérer l'évolution du code source de leur application. Les enseignants doivent avoir accès en lecture aux dépôts des étudiants tout au long du projet, pour surveiller le travail et les progrès de chaque groupe et de ses membres. Dans le cas où, pour une raison quelconque, un étudiant venait à se retrouver seul dans son groupe, l'utilisation du système de contrôle de versions resterait néanmoins obligatoire. Des "commits" réguliers doivent être effectués.

**Communication.** Slack (<https://slack.com>) est un outil de communication puissant, souvent utilisé par des communautés professionnels de développement logiciel. Un workspace a été créé pour communiquer sur le projet : [pgl21-22.slack.com](https://pgl21-22.slack.com). Les membres de chaque groupe doivent utiliser un channel dédié dans ce workspace pour communiquer entre eux. D'autres channels seront utilisés pour assurer la communication entre enseignants et étudiants, ou pour permettre des discussions techniques entre l'ensemble des étudiants.

**Travail en groupe.** Chaque groupe travaillera **séparément** à la réalisation d'une planification, modélisation et implémentation du jeu et de ses extensions. Chaque membre du groupe travaillera de manière individuelle à son extension. Toute collaboration entre les groupes est interdite sauf pour des discussions techniques spécifiques sur Slack, si approuvées et modérées par les enseignants dans un channel dédié.

**Contraintes de temps.** Les livrables à rendre pour chaque étape-clé doivent être déposés sur Moodle avant chaque échéance imposée. Puisque l'un des objectifs du projet consiste à apprendre à respecter les échéances imposées lors d'un projet de développement logiciel, *aucun retard ne sera toléré*.

**Exigences de qualité.** La fonctionnalité, la complétude et la qualité des livrables seront évaluées par les enseignants après l'échéance imposée pour chaque étape clé.

## 2.3 Outils

**Outil de modélisation.** Vous avez le libre choix des outils de modélisation. Beaucoup d'outils commerciaux ou open source sont disponibles sous la forme de stand-alone ou de plug-in pour des plateformes de développement intégrés.

Pour concevoir des maquettes graphiques pour des applications JavaFX, il existe plusieurs éditeurs graphiques pour le format fxml. Un exemple est JavaFX Scene Builder d'Oracle<sup>18</sup>.

L'UMONS possède une licence académique pour Visual Paradigm<sup>19</sup>. Cet outil supporte la modélisation

16. <https://en.wikipedia.org/wiki/Javadoc>

17. <http://www.agiledata.org/essays/tdd.html>

18. <https://www.oracle.com/java/technologies/javase/javafxscenebuilder-info.html>

19. <https://ap.visual-paradigm.com/university-of-mons>

et la conception des systèmes logiciels, incluant à la fois la modélisation UML et la modélisation des données avec les diagrammes d'entité-relation<sup>20</sup>. Il y a même du support pour créer des maquettes de l'interface utilisateur de votre système, et des visualisations permettant de faire le design d'un API REST<sup>21</sup> !

**Outils de développement.** L'application doit **obligatoirement** être implémentée en Java 11. Ce choix est justifié par le fait que cette version de Java correspondent à un long-term support (LTS).<sup>22</sup> Vous pouvez librement choisir votre *environnement de développement* (par exemple IntelliJ IDEA, Eclipse ou NetBeans) à condition que les enseignants qui évalueront l'application puissent facilement compiler et exécuter votre logiciel sans devoir installer cet environnement.

Pour réaliser les *tests unitaires* de l'application et ses extensions, le framework de test JUnit 5 est obligatoire.<sup>23</sup> Pour faciliter l'écriture des tests unitaires il est fortement recommandé d'utiliser une librairie de mocking. Nous recommandons l'utilisation de Mockito<sup>24</sup> mais d'autres librairies sont envisageables (par exemple, EasyMock<sup>25</sup>).

☞ L'utilisation de librairies tierces supplémentaires peut être envisagée. Vous devez cependant demander et obtenir l'accord explicite des enseignants avant tout usage d'une telle librairie.

**Moteur de production automatique.** L'utilisation du moteur de production gradle<sup>26</sup>, dans sa version 7.2, est obligatoire. C'est un outil en logiciel libre facilitant la gestion et l'automatisation de cycles de production de projets logiciels. Il permet aux étudiants et enseignants la compilation du code source et l'exécution de votre projet, sa documentation et ses tests unitaires à partir de la ligne de commande sans avoir à installer un environnement de développement Java quelconque. Vous devrez utiliser cet outil pour gérer vos dépendances logicielles, de sorte qu'un appel par ligne de commande suffise à valider, compiler, tester, et packager l'application de base ainsi que chaque extension séparément depuis un nouvel environnement de travail.

On peut trouver des tutoriels sur gradle ici : <https://gradle.org/guides/>

Et voici un tutoriel spécifiquement destiné à la production des applications Java :

<https://guides.gradle.org/building-java-applications/>

Il est fortement conseillé d'également utiliser le "Gradle wrapper" (gradlew : [https://docs.gradle.org/current/userguide/gradle\\_wrapper.html](https://docs.gradle.org/current/userguide/gradle_wrapper.html)) pour gagner encore en robustesse dans le processus de déploiement.

**Outil de gestion de versions.** L'utilisation de git<sup>27</sup> est imposée pour la gestion des versions et le travail collaboratif. Chaque groupe devra créer un dépôt **privé** sur la plateforme de son choix (exemples : GitHub, BitBucket, GitLab) et donner aux enseignants l'accès à celui-ci suivant les consignes qu'ils auront données. Un groupe n'aura pas accès aux dépôts des autres groupes. Les enseignants auront constamment accès aux dépôts fournis. Utilisez-les intelligemment, en respectant les bonnes pratiques. Au sein d'une branche, le code source sera placé dans un répertoire, dans le respect des conventions proposées par gradle. Toutes vos classes et autres ressources devront appartenir (directement ou indirectement) au package `be.ac.umons.gXX` où XX est le numéro de groupe.

**Système d'exploitation.** Vous pouvez utiliser n'importe quel système d'exploitation pour développer l'application. La seule contrainte est que tous les livrables de la phase d'implémentation soient indépendants de la plateforme choisie. Le code exécutable sera testé sur trois systèmes d'exploitation différents : MacOS X, Linux et Windows.

20. <https://www.visual-paradigm.com/solution/dbdesign/best-er-diagram-tool/>

21. <https://www.visual-paradigm.com/solution/rest-api-design-tool/>

22. [https://en.wikipedia.org/wiki/Java\\_version\\_history](https://en.wikipedia.org/wiki/Java_version_history)

23. <https://junit.org/junit5>

24. <https://site.mockito.org>

25. <http://easymock.org>

26. <https://gradle.org>

27. <http://git-scm.com>

Une attention particulière doit être portée aux problèmes d'encodage des caractères qui rendent les accents illisibles sur certains systèmes d'exploitation. Un autre problème récurrent est l'utilisation des chemins représentant des fichiers : Windows utilise une barre oblique inversée (backslash) tandis que les systèmes dérivés d'Unix utilisent une barre oblique (slash). La constante `File.separator` donne une représentation abstraite du caractère de séparation.

## 3 Livrables du projet

Chaque phase du projet aura ses propres livrables. Si le livrable contient un rapport, nous conseillons vivement d'utiliser  $\text{\LaTeX}$  pour sa rédaction.<sup>28</sup>

### 3.1 Démarrage

Cette phase marque le début du projet. L'énoncé sera distribué et les étudiants seront affectés aux groupes. Sauf exception, chaque groupe sera composé de trois étudiants. Chaque groupe doit réaliser les fonctionnalités de base. Chaque membre du groupe doit réaliser une extension.

**Livrable : liste de groupes.** Au terme de cette phase, une liste contenant l'attribution des étudiants aux groupes sera fournie par les assistants. Excepté en cas de force majeure, les groupes ne pourront plus être modifiés lors du déroulement du projet.

**Livrable : accord sur les extensions.** Les enseignants du cours doivent marquer leur accord sur les extensions choisies par chaque groupe, ainsi que sur l'affectation de ces extensions aux membres du groupe.

### 3.2 Phase d'analyse et de conception (S-INFO-015)

La conception (*design*) de l'application sera modélisée de manière individuelle par chaque groupe en utilisant le langage de modélisation UML 2.5 ou version supérieure et en proposant une maquette de l'interface utilisateur.

**Livrable 1 : les maquettes de l'interface utilisateur.** Une maquette présente de manière visuelle l'interface utilisateur de l'application à réaliser, et la façon d'interagir avec cette interface. Elle sera réalisée au moyen d'une application dédiée choisie par le groupe. La maquette visuelle doit être accompagnée d'une description textuelle décrivant la façon d'utiliser le système logiciel.

Plusieurs maquettes doivent être rendues : une pour chaque application de la fonctionnalité de base du système ; et une pour chaque extension choisie.

**Livrable 2 : le rapport de modélisation.** Ce rapport doit comporter les différents modèles de conception. Ceci inclut les diagrammes UML, ainsi qu'un modèle de données relationnelles et un schéma de l'API REST. Chaque modèle doit être accompagné par une description textuelle justifiant les choix de conceptions qui ont été faits pour chaque diagramme :

- Un ou plusieurs diagrammes d'entité-relation (ERD), décrivant le(s) modèle(s) de données relationnelles.
- Un schéma ou modèle décrivant le design de l'API REST qui sera proposé.
- Un diagramme de cas d'utilisation, accompagné d'une description semi-structurée de chaque cas d'utilisation selon le canevas présenté lors du cours.
- Un interaction overview diagram présentant la vue d'ensemble des interactions entre les cas d'utilisation.
- Les diagrammes de classes et de paquetage présentant le design de la structure de l'application.
- Les diagrammes de séquences représentant le design du comportement de l'application.

Optionnellement, d'autres diagrammes UML peuvent être rendus, comme les diagrammes de composants, diagrammes d'activités et statechart (diagrammes d'états comportementaux). Les diagrammes optionnels seront considérés en bonus lors de l'évaluation du projet.

☞ Pour faciliter la lecture, les images des diagrammes UML dans le rapport doivent utiliser un fond blanc ou transparent. Le contenu des diagrammes doit rester lisible sans soucis après impression du rapport en noir et blanc.

---

28. L'outil  $\text{\LaTeX}$  sera également utilisé pour la rédaction du projet et du mémoire du master en sciences informatiques.



### 3.3 Phase de développement logiciel (S-INFO-106)

Chaque groupe doit travailler de manière séparée lors de la phase d'implémentation, en respectant toutes les licences (open source ou autres) des classes et bibliothèques tierces utilisées.

☞ **Le plagiat** (du code venant des autres groupes, d'Internet, ou de n'importe quelle autre source) **sera lourdement pénalisé** et peut mener à un échec total pour le projet ou même pour l'année d'études (cf. règlement des études).

Si vous désirez réutiliser du code (p.ex., des bibliothèques) open source dans le cadre de votre projet, vous devez obtenir l'accord préalable des enseignants, et vous devez indiquer cela clairement dans les commentaires et la documentation de votre code source ainsi que dans vos rapports.

**Livrable 1 : le code source.** Une version complète du code source (en Java) et de l'interface graphique (en JavaFX) de l'application de base et des extensions.

**Livrable 2 : le code compilé.** Une configuration **gradle** permettant de compiler automatiquement et sans erreur le code exécutable et des archives jar de la fonctionnalité de base de l'application, ainsi que de chaque extension individuellement. Des versions précompilées (complètes, auto-exécutables et indépendantes de la plateforme) sous la forme d'archives .jar, de la fonctionnalité de base, ainsi que de chaque extension individuellement. Attention à bien inclure les dépendances éventuelles dans vos archives .jar (via gradle). N'hésitez pas à tester l'exécution de vos archives sur des machines "neutres" autres que celles que vous utilisez pour vos développements.

**Livrable 3 : la documentation du code.** Le code doit être documenté avec JavaDoc. Celle-ci doit permettre au lecteur une compréhension d'ensemble du système développé. La JavaDoc doit contenir une documentation complète pour le projet ; les packages ; les classes, interfaces et énumérations ; les méthodes principales. La configuration **gradle** doit permettre de générer automatiquement cette documentation.

**Livrable 4 : les tests unitaires.** Les tests unitaires en JUnit doivent être présents, et la suite de tests doit être exécutable automatiquement avec **gradle**. Cette exécution ne peut pas donner lieu à des échecs ou des erreurs.

Les *tests unitaires* doivent être utilisés tout au long de la phase d'implémentation. Une approche de développement dirigé par les tests est imposée : l'écriture du code doit être alternée avec celle des tests, et tous les tests doivent réussir avant de progresser dans l'écriture du code. Une autre bonne pratique qui sera évaluée est l'utilisation de *tests de régression*. Pour chaque défaut (erreur, bug, mauvais comportement) rencontré, un test unitaire doit être écrit qui met en évidence ce défaut. Vous n'aurez ainsi pas à comprendre et résoudre deux fois le même problème. Les tests doivent être fournis pour les fonctionnalités de base ainsi que pour chaque extension réalisée.

**Livrable 5 : le rapport d'implémentation.** Le rapport d'implémentation doit contenir une motivation des choix d'implémentation par rapport à la modélisation UML. Si le design établi lors de la phase de modélisation a été revu, les modifications du design doivent être présentées et dûment justifiées.

Le rapport d'implémentation doit aussi inclure : une description et une motivation des design patterns utilisés dans le code ; une description des algorithmes importants ; les commandes **gradle** pour générer le code compilé et la documentation, ainsi que pour exécuter le code et les tests.

**Livrable 6 : une vidéo du manuel d'utilisation.** Une **vidéo** d'environ 10 à 15 minutes, accessible sur internet par un lien URL précisé dans le rapport d'implémentation (et accessible aux enseignants sans installation de logiciel ou inscription à un quelconque service). La vidéo doit être accompagnée par une explication orale, et doit illustrer *toutes les fonctionnalités de base ainsi que les fonctionnalités de chaque extension*.

Cette vidéo doit également servir comme mode d'emploi expliquant le fonctionnement de l'application et des extensions. Chaque utilisateur doit être capable d'utiliser l'application après avoir vu la vidéo. Les éventuels problèmes, bugs, ou fonctionnalités manquantes doivent être mentionnés dans la vidéo. (Si des logins ou mots de passe sont requis, ils doivent être fournis dans le rapport d'implémentation.)

☞ Il est permis de fournir des vidéos séparées pour la fonctionnalité de base et pour chaque extension. Dans ce cas, chaque vidéo fournie ne peut pas dépasser les 5 minutes.

**Défense orale de l'implémentation.** Le projet se termine par une défense orale. Celle-ci est une dernière réunion d'inspection durant laquelle le code exécutable sera présenté aux enseignants et le code source et les tests unitaires seront défendus. Les enseignants donneront un feedback oral et écrit sur le contenu et la qualité du projet.

## 4 Critères de recevabilité

☞ Cette check-list reprend l'ensemble des consignes à respecter pour la remise du projet.

Le non-respect des critères implique la non-recevabilité du projet. Celui-ci sera dès lors considéré comme non rendu et l'étudiant ou le groupe sera sanctionné par une note de 0/20 à la partie rendue !

**Respect des échéances** La date et heure de limite de remise est indiquée sur Moodle et devra être respectée à la minute. Aucun délai ne sera accordé, et aucun retard ne sera toléré. Il vous est donc conseillé d'uploader des versions intermédiaires avant la version définitive (seule la dernière version rendue à temps sera évaluée).

**Absence de plagiat** Toute présence de plagiat implique la non-recevabilité du projet. Un outil automatisé sera utilisé afin de vérifier l'absence du code dupliqué entre les différents groupes.

**Formats d'archive et de fichiers** Lors de chaque phase du projet, votre travail devra être remis sous forme d'une seule archive **.zip** dont l'intitulé aura le format suivant :

1. Pour la phase d'analyse et de conception : PAC<numéro de groupe>.zip
2. Pour la phase de développement logiciel : PDL<numéro de groupe>.zip

Les rapports contenus dans chaque archive doivent être en format **PDF**. Chaque rapport doit commencer par une page de garde indiquant l'intitulé du rapport, l'année académique, le numéro du groupe, et pour chaque étudiant membre du groupe le nom, matricule et l'extension choisie.

☞ Les images intégrées dans les rapports de chaque phase doivent être bien lisibles, en format A4, même après impression en noir et blanc. (Vous pouvez toujours joindre les images originales et leurs sources dans le dossier contenant le rapport.)

**Contenu d'archive pour la phase d'analyse et de conception** Votre archive doit obligatoirement contenir les éléments suivants :

- Le rapport de *modélisation* incluant toutes les images, et portant le nom PAC<numéro de groupe>-rapport.pdf
- Les *maquettes de l'interface utilisateur* portant le nom PAC<numéro de groupe>-maquette<numéro de la maquette >.pdf

☞ Veillez à bien séparer la partie commune (fonctionnalité de base réalisée en groupe) et la partie individuelle (extensions réalisées individuellement par chaque étudiant).

**Contenu d'archive pour la phase de développement logiciel** Votre archive doit obligatoirement contenir les éléments suivants :

- Le rapport d'*implémentation* incluant toutes les images, et portant le nom PDL<numéro de groupe>-rapport.pdf
- Une (ou plusieurs) vidéos, accessible(s) aux enseignants par un lien internet (dont l'URL est précisée dans le rapport d'implémentation), présentant *toutes les fonctionnalités de base et de chaque extension*, et illustrant comment l'application peut être utilisée, tout en mentionnant les éventuels problèmes ou manquements restants.
- Des archives **jar** auto-exécutables (pour les fonctionnalités de base et pour chaque extension), portant le nom PDL<numéro de groupe>-<nom de l'archive>.jar
- Le code source et la documentation JavaDoc contenant tous les fichiers nécessaires à sa compilation ainsi qu'à son exécution.
- Les tests unitaires contenant tous les fichiers nécessaires à sa compilation ainsi qu'à son exécution.

☞ La compilation du code source, de la documentation, de l'archive auto-exécutable et des tests unitaires doit être réalisable au moyen d'une commande **gradle** ou **gradlew** sans que cela ne génère d'erreur. Le rapport doit préciser les commandes à utiliser.

☞ Veillez à bien séparer la partie commune (fonctionnalité de base réalisée en groupe) et la partie individuelle (extensions réalisées individuellement par chaque étudiant).

## 5 Critères d'évaluation

Tous les étudiants du même groupe obtiendront une note qui tient compte du travail individuel (l'extension réalisé par chaque étudiant) et le travail en commun (les fonctionnalités de base). Les livrables rendus seront évalués selon les critères suivants :

### 5.1 Pour la phase d'analyse de de conception

#### 5.1.1 Maquette de l'interface utilisateur

La maquette couvre-t-elle tous les aspects de l'énoncé? Toutes les fonctionnalités sont-elles prises en compte? La maquette de l'interface graphique est-elle conviviale, intuitive et ergonomique?

Une maquette n'est pas juste un ensemble d'images représentant comment on propose de visualiser l'interface graphique. Les images doivent obligatoirement être accompagnées du texte expliquant comment l'utilisateur est sensé utiliser l'interface graphique pour réaliser les fonctionnalités de base et des extensions fournies par les applications clients.

#### 5.1.2 Rapport de modélisation

##### Modèle de données

Le modèle de données couvre-il tous les aspects de l'énoncé? Toutes les exigences sont-elles prises en compte?

Les *diagrammes d'entité-relation* (ERD) sont-ils syntaxiquement et sémantiquement cohérents?

Les ERD présentent-ils toutes les entités, leurs attributs et relations? Les multiplicités, clés primaires et étrangères sont-elles bien représentées? Les tables incluses dans le diagramme couvrent-elles toutes les fonctionnalités envisagées? Observe-t-on de mauvaises pratiques? Des optimisations sont-elles possibles?

##### Design de l'API REST

Le schéma ou modèle décrivant le design de l'API REST est-il complet et compatible avec les fonctionnalités demandés?

##### Modèles UML

**Conformité au cahier des charges / Complétude** Les diagrammes UML utilisés sont-ils complets? Couvrent-ils tous les aspects de l'énoncé? Toutes les exigences (fonctionnelles et non fonctionnelles) sont-elles prises en compte?

**Style** Les diagrammes sont-ils bien structurés? Les diagrammes UML respectent-ils la version de la norme imposée? Suivent-ils un style de conception orienté objet? (Par exemple, une bonne utilisation de la généralisation et de l'association entre les classes, utilisation d'interfaces et de classes abstraites, une description des attributs et des opérations pour chaque classe.)

**Cohérence** Les diagrammes UML sont-ils syntaxiquement et sémantiquement cohérents? N'y a-t-il pas d'incohérences : (i) dans les diagrammes (p.ex. opérations et attributs dans les classes, multiplicités sur les associations, ...); (ii) entre les différents diagrammes? (p.ex. Les objets et messages dans un diagramme de séquences correspondent-ils aux classes et opérations dans le diagramme de classes?)

**Exactitude** Les différents éléments d'un diagramme sont-ils utilisés correctement? Par exemple :

1. Dans un *diagramme de cas d'utilisation*, les acteurs sont-ils correctement définis? Les notions de généralisation, d'extension et d'inclusion sont-elles correctement mises en œuvre? Fait-on appel aux points d'extension lorsqu'ils sont nécessaires? Y a-t-il une description textuelle ou structurelle de chaque cas d'utilisation?

2. Dans un *diagramme de classes*, les multiplicités sur les associations sont-elles judicieusement utilisées ? L'utilisation de la composition, de l'agrégation et de l'association est-elle pertinente ? Les responsabilités (fonctionnalités) sont-elles bien distribuées au différentes classes ? Observe-t-on la présence de *design patterns* et l'absence d'*antipatterns* (par exemple des "god class") ? Les opérations et attributs définis dans chaque classe sont-ils suffisants pour permettre à l'application de fonctionner correctement ?
3. Dans un *diagramme de séquence*, les opérations appelées correspondent-elles à celles décrites dans le diagramme de classes ? Les objets commencent-ils et finissent-ils leur vie au bon moment ? Les objets communiquant entre eux sont-ils connectés ensemble ? Les fragments combinés sont-ils utilisés correctement pour représenter des boucles, des conditions, des exceptions, du parallélisme ?
4. Dans un *diagramme d'états comportementaux (statechart)*, les états composites sont-ils utilisés pour améliorer la compréhensibilité du diagramme ? Les états modélisés ne sont-ils pas *artificiels* ? Reflètent-ils le comportement exact de l'application ? Les transitions représentent-elles fidèlement les différents changements pouvant survenir ? Les gardes, événements et actions (sur les transitions et au sein des états) sont-ils utilisés correctement ?

**Compréhensibilité** Les diagrammes sont-ils faciles à comprendre ? Ont-ils le bon niveau de détail ? Pas trop abstraits, pas trop détaillés ?

**Lisibilité** Les diagrammes ne peuvent pas être dessinés à la main et doivent être lisibles sans devoir agrandir le document.

## 5.2 Pour la phase d'implémentation

### 5.2.1 Qualité du code source

**Conformité au cahier des charges** Toutes les fonctionnalités requises sont-elles implémentées ?

**Conformité au design** Le code source est-il cohérent avec les modèles UML ?

**Style** Le code Java respecte-t-il la version Java imposée ?

Le code est-il bien structuré (en packages, classes, méthodes, ...) ?

Le code suit-il les principes de la *programmation orientée objet*, en utilisant le mécanisme de typage, héritage, polymorphisme, liaison tardive, interfaces, classes et méthodes abstraites... ? Le code n'est-il pas inutilement complexe ? À tout moment, il faut éviter un style procédural avec des méthodes complexes et beaucoup d'instructions conditionnelles.

Le code suit-il un *style de programmation défensive* ? (Afin de réduire les erreurs lors de l'exécution du programme, le programme doit utiliser au maximum les mécanismes de typage, d'exceptions et d'assertions.)

Les *design patterns* sont-ils utilisés lors de l'implémentation ?

Les *conventions de nommage* sont-elles respectées ? <sup>29</sup>

La *duplication de code* et la présence de *code mort* sont-elles évitées ?

**Lisibilité** Le code est-il facile à lire et comprendre ?

### 5.2.2 Testabilité

Les tests unitaires accompagnant le code source seront évalués selon les critères suivants :

**Processus** Le processus de test-driven development a-t-il été suivi ?

**Couverture** Les tests unitaires couvrent-ils toute la fonctionnalité requise ? Y a-t-il des tests superflus ?  
Toutes les classes et méthodes importantes sont-elles couvertes par des tests ?

**Exactitude** Les tests sont-ils corrects ? Tous les tests unitaires fonctionnent-ils sans échec ni erreur ?

---

29. <https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html>

**Qualité** Les tests sont-ils de bonne qualité (pas de tests trop simples, des tests avec un comportement trivial, ...) ? Y a-t-il une librairie de *mocking* qui a été utilisé pour la création des *mock objets* lors des tests comportementaux ?

**Automatisation** Est-il possible, grâce à *gradle*, d'exécuter tous les tests unitaires en une seule fois ?

### 5.2.3 Exécutabilité et fonctionnalité

**Compilation** Y a-t-il des problèmes ou warnings lors de la compilation ? Est-il possible de compiler et exécuter le code (et les tests et le JavaDoc) avec *gradle* ?

**Ergonomie et convivialité** L'application (et son interface graphique) est-elle conviviale, intuitive et facile à utiliser ?

**Fluidité et efficacité** L'utilisation de l'application est-elle fluide et performante ? N'y a-t-il pas de temps d'attente inutile ?

**Complétude** Toutes les exigences du cahier des charges sont-elles prises en compte par l'application ?

**Exactitude et fiabilité** Le programme ne doit pas contenir de bugs, et ne doit pas planter.

L'application fonctionne-t-elle correctement dans des circonstances normales ?

L'application fonctionne-t-elle correctement dans des circonstances exceptionnelles (p.ex., données erronées, format de données incorrect, problème de réseau, problème de sécurité, ...) ?

**Indépendance de la plate-forme** Le code produit est-il exécutable sans aucun souci sur trois systèmes d'exploitation différents (MacOS X, Linux et Windows) ? N'y a-t-il pas de problème avec des caractères spéciaux, chemins de fichier, ... ?

### 5.2.4 Documentation

**Rapport d'implémentation** Le rapport d'implémentation du projet est-il bien rédigé (style d'écriture, facilité de lecture, structure, ...) ? Justifie-t-il tous les choix d'implémentation importants (p.ex. les librairies externes utilisées, les design patterns utilisés, les déviations éventuelles du cahier de charges, les fonctionnalités non-implémentées, les problèmes rencontrés, les limitations de l'application réalisée) ?

**Vidéo** Les vidéos fournies présentent-t-elles toutes les fonctionnalités de base, ainsi que les fonctionnalités de chaque extension ? Sont-elles suffisamment complètes pour qu'un utilisateur puisse utiliser l'application sans problème ? Contiennent-elles une description orale du déroulement de l'application ? Les éventuels problèmes ou manquements restants sont-ils bien mentionnés ?

**JavaDoc** Le code est-il bien documenté avec JavaDoc ? La JavaDoc est-elle présente et est-elle suffisamment complète ? Est-elle de bonne qualité ? Les informations qu'on trouve dedans sont-elles pertinentes et utiles pour quelqu'un qui veut comprendre ou modifier le code ? Est-il possible, grâce à *gradle*, de générer la JavaDoc ?

## 6 Dates importantes

Le projet est jalonné de dates reprises ci-dessous. Dans le cas où la date est associée à la remise d'un livrable, elle constitue l'extrême limite de remise du livrable sur la plateforme Moodle. **Aucun retard ne sera toléré.**

### **S-INFO-015 “Projet d’analyse et de conception”** (Bloc 2 Bachelier Sciences Informatiques)

**Le 28 septembre 2021** Présentation du projet et distribution de l'énoncé. Proposition de formation des groupes.

**Le 14 octobre 2021** Les groupes sont définitivement formés. Début de la phase d'analyse et de conception.

**Vendredi 17 décembre 2021** Remise des livrables pour la phase d'analyse et de conception :

Rapport de modélisation : UML + modèle de données + design de l'API REST

Maquette de l'interface utilisateur

**Le livrable doit bien distinguer la maquette et les modèles pour la fonctionnalité de base (réalisée en groupe) et les extensions (réalisées de manière individuelle).**

**Feedback** *Un feedback écrit sera donné aux étudiants concernant la note obtenue et les points à améliorer.*

### **S-INFO-106 “Projet de développement logiciel”** (Bloc 2 Bachelier Sciences Informatiques) (Bloc complémentaire Master Sciences Informatiques)

**Lundi 7 février 2022** Début de la phase de développement logiciel.

**Lundi 21 mars 2022** Remise d'une pré-version de l'application contenant les fonctionnalités de base.

Lors de cette remise nous vérifierons si votre travail rendu respecte plusieurs critères de recevabilité importants attendus pour cette pré-version :

**Elle DOIT être installable, compilable, exécutable et testable avec gradle. Des tests unitaires DOIVENT déjà être présent pour tester le code (car vous devez suivre un processus de test-driven development). Du JavaDoc DOIT être présent pour documenter le code. Des archives jar auto-exécutables DOIVENT être présents.**

**Lundi 25 avril 2022** Remise finale des livrables pour la phase de développement logiciel :

Rapport d'implémentation

Vidéo de la fonctionnalité (manuel d'utilisation)

Scripts gradle, code source, tests unitaires, archives auto-exécutables, JavaDoc

**Le livrable doit bien distinguer la fonctionnalité de base (réalisée en groupe) et les extensions (réalisées de manière individuelle).**

**Mercredi 18 mai 2022** Défenses orales, tests d'acceptation par les enseignants.

**Votre présence est obligatoire ! (Absence injustifiée = 0/20)**