Laboratoire d'Ingénierie du Logiciel
Étude de cas DUFLOUZ

Version 3.1

Vincent Englebert Fabian Gilson

29 octobre 2013

Table des matières

Historique

1.0 Création initiale

1.1 Modifications mineures

1.2 Modifications du dom. d'application + autres modifications mineures.

2.0 Corrections majeures et remise en forme des scenarii, spécifications et diagrammes UML (UML 2.2).

2.1 Ajout du scénario de "Retrait" en diag. de robustesse, remise en page et diverses corrections mineures.

2.2 Ajout des contraintes et outils, et corrections mineures.

3.0 Mise en forme des diagrammes en astah* et remise en forme du document. Corrections majeures des UC, spécifications et machines à états. Ajout des GUI et standards. Levée de quelques imprécisions et contradictions.

3.1 Corrections des diagrammes d'états et du processus métier.

Préambule

Avant toute chose, il est important de préciser que cette étude de cas est *académique*. Elle ne correspond à aucune demande réelle d'un client. Elle est volontairement incomplète, son seul but est d'illustrer la méthode que vous emploierez lors du laboratoire. Ainsi, par exemple, toutes les interactions possibles entre composants ne sont pas explicitées, de même, tous les services offerts par l'application ne sont pas spécifiés.

Enonce

Notre client est la toute nouvelle banque DUFLOUZ qui nous demande de concevoir son infrastructure logicielle. Pour cette étude de cas, nous ne nous intéresserons qu'aux guichets automatiques (ATM), et ferons donc abstraction des agences.

La banque possède un certain nombre de clients, chaque client possédant un ou plusieurs comptes, chacun étant identifié par un numéro au format IBAN de la forme **BE***kk-BBBC-CCCC-CCKK*) ; BE = code pays ; k = *checksum digit* ; B = code banque ; C = numéro compte ; K = *checksum* numéro compte (C). Chaque client dispose d'une carte bancaire à laquelle est associée un code secret et donnant accès à ses comptes.

La banque dispose d'un ensemble d'ATM connectés via un réseau. Chaque ATM possède un lecteur de cartes magnétiques, un distributeur de billets, un clavier, un écran tactile et une imprimante de reçus. Un ATM permet à un client de retirer de l'argent d'un des comptes correspondants à la carte bancaire insérée, de consulter le solde de ses comptes ou de virer de l'argent vers un autre compte. Une transaction est initiée lorsqu'un client insère sa carte dans le lecteur. Sur la puce de cette carte se trouvent le numéro de la carte, sa date de mise en service et sa date d'expiration. Le système valide la carte en déterminant si la date d'expiration n'est pas dépassée, si le code secret fourni par le client correspond à celui maintenu par le système et si la carte n'est pas déclarée comme perdue ou volée. Le client dispose de trois tentatives pour entrer le code secret correct, après quoi la carte sera confisquée par l'appareil.

Si le code entré est correct, le client a le choix entre trois alternatives : retirer de l'argent, consulter le solde d'un des comptes liés à la carte ou virer de l'argent vers un autre compte. Pour qu'un retrait soit approuvé, il faut que le client dispose d'un montant suffisant sur le compte demandé, qu'il ne dépasse pas le montant journalier maximum autorisé et que l'ATM dispose de suffisamment d'argent pour satisfaire la requête. Lorsque ce retrait est approuvé, l'argent est débité, un reçu est imprimé (si demandé) et la carte est éjectée. Pour qu'un virement soit approuvé, il faut que le compte débiteur dispose d'un solde suffisant. Le client peut annuler une transaction à n'importe quel moment, l'ATM éjecte alors la carte.

Un opérateur d'ATM peut démarrer et arrêter un ATM en vue de le réapprovisionner.

1 Introduction

Le présent document présente l'analyse des besoins, la conception, ainsi que les détails d'implémentation de l'application de gestion des guichets automatiques de la société Duflouz. Cette banque est nouvelle sur le marché et nous a confié la réalisation de leur infrastructure logicielle *from-scratch*. Nous nous sommes concentrés sur les guichets automatiques (ATM) et n'avons donc pas pris en compte la gestion des agences. L'application développée a pour but d'offrir aux clients de la banque Duflouz un moyen simple et rapide afin d'effectuer des retraits d'argent, des virements ainsi que de consulter les soldes de leurs comptes via ces machines ATM.

Ce document présente tout d'abord l'analyse des exigences à la section 2. Cette analyse présente, dans l'ordre, la modélisation du domaine d'application avec son dictionnaire des données, une description des utilisateurs, la modélisation des cas d'utilisation du système, une liste d'exigences non fonctionnelles, la modélisation du processus métier et enfin, les choix des outils utilisés pour le projet et le planning des tâches à venir. Ensuite, à la section 3 nous présentons la conception logique du système. Nous commençons par réécrire nos cas d'utilisation en diagrammes de robustesse afin d'identifier les composants de base de l'application à développer. Nous donnons ensuite une représentation statique de l'architecture logicielle du système sous la forme d'un diagramme de composants. Nous formalisons chaque scénario d'utilisation sous forme de diagrammes de séquence montrant les échanges de messages entre composants, qui sont spécifiés de manière semi-formelle par des pré/post conditions et des diagrammes d'états. Dans la section 4, nous décrivons les interfaces graphiques du futur système et nous posons les choix technologiques pour le

développement de celui-ci. Nous formalisons les services distants offerts par les composants logiciels sous forme d'interface IDL. Nous décrivons l'architecture physique sous forme de diagramme de classes et nous schématisons son déploiement afin de représenter les contraintes liées à l'infrastructure physique. Enfin, nous présentons la stratégie de test appliquée lors de l'implémentation à la section 5.

2 Analyse des exigences

2.1 Modélisation du domaine d'application

Afin de représenter le domaine d'application, nous utilisons un langage de modélisation relationnel nommé ERA (*Entités-Relations-Attributs*) à la figure 1. Nous listons ensuite les concepts exposés dans le modèle avec une description de leurs sémantiques.

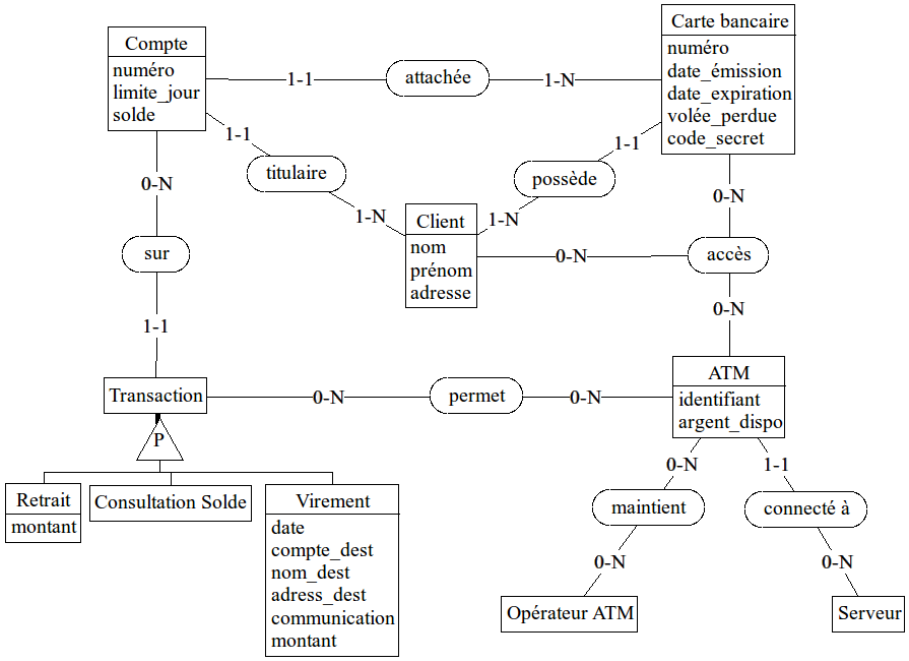

FIGURE 1 – Modèle du domaine d'application

Accès un *accès* représente une *transaction* d'un *client* auprès d'un *ATM* au moyen de sa *carte bancaire*.

ATM Un *ATM* (Automatic Teller Machine) est une machine permettant d'effectuer des *transactions* sur les *comptes* d'un *client*. Elle dispose d'un lecteur de cartes magnétiques, d'un distributeur de billets, d'un clavier, d'un écran tactile et d'une imprimante de reçus. Elle ne peut être démarrée, stoppée ou réapprovisionnée que par un *opérateur*. Elle dispose d'un numéro identifiant ainsi que d'une réserve d'argent.

Carte Bancaire Une *carte bancaire* est liée à un ou plusieurs *comptes* d'un *client*. Elle certifie que son détenteur est bien le possesseur de ses *comptes* s'il connait effectivement le code secret attaché à la carte. Elle permet de faire des transactions via un *ATM* pour autant que la date limite de la carte ne soit pas dépassée et qu'elle ne soit pas déclarée comme volée ou perdue. Une carte possède un numéro unique, une date d'émission et d'expiration et peut être déclarée volée ou perdue.

Client Un *client* est une personne disposant d'un ou plusieurs *comptes* auprès de la banque. Il est identifié par son nom, son prénom et son adresse.

Compte Un *compte* désigne à la fois un *client* au sein de la banque ainsi que l'état de ses avoirs dans la banque. Un compte est identifié par un numéro, dispose d'un solde et les *retraits* qu'un client peut effectuer sur ce compte sont limités par une limite journalière.

Consultation solde La *consultation du solde* consiste à interroger le *serveur* afin de connaître le solde d'un *compte* donné au moyen d'un affichage et/ou d'un *reçu*.

Opérateur (ATM) Un *opérateur* est la seule personne habilitée à démarrer, stopper et réapprovisionner un ATM.

Reçu Un *reçu* est un document papier reprenant le nom et le numéro de compte du *client* ainsi que soit le solde de l'un de ses *comptes*, soit le montant d'un *retrait*, soit le récapitulatif d'un *virement*.

Retrait Un *retrait* consiste à débiter un *compte* donné d'une somme d'argent donnée et de dispenser au *client* cette somme en liquide (pour autant que le compte dispose d'un solde suffisant et que la limite journalière ne soit pas dépassée).

Serveur Le *serveur* de la banque est l'unité centrale contenant les informations sur les *clients* de la banque et sont interrogés par les *ATMs* lors des *accès* par les *clients*.

Transaction Une *transaction* est soit un *retrait*, soit une *consultation de solde*, soit un *virement*.

Virement Un *virement* consiste à transférer un montant donné d'un *compte* vers un autre *compte*, à une certaine date. Un *virement* peut être complété d'une communication textuelle.

2.2 Profil des utilisateurs et intéressés

2.2.1 Client

Cette catégorie regroupe l'ensemble des utilisateurs potentiels des guichets automatiques.

Attributs physiques
- âge : plus de 12 ans (limite pour la détention d'une carte de débit en Belgique)
- sexe : non pertinent
- handicaps physiques éventuels : ce profil n'inclut pas les personnes atteintes de cécité, ou de tout handicap les empêchant de manipuler un clavier placé à une certaine hauteur et un écran tactile (handicap moteur)

Attributs mentaux
- les utilisateurs ne possèdent pas d'aptitudes distinctives
- déficiences mentales : certains utilisateurs peuvent avoir des troubles de la mémoire (code PIN)
- motivation : la motivation à l'utilisation du système est supposée relativement élevée, vu le gain de temps qu'elle constitue par rapport aux opérations faites à un guichet traditionnel ; néanmoins, ce profil peut contenir des personnes réticentes à toute technologie

Compétences
- expérience de la tâche : la plupart des utilisateurs ne sont pas censés avoir déjà utilisé un tel système auparavant
- expérience avec des ordinateurs : une certaine portion de la population n'a jamais utilisé d'ordinateur et a très peu d'expérience avec des systèmes électroniques
- connaissances générales et niveau de qualification : aucun (nivellement par le bas)
- langue : français

2.2.2 Opérateur ATM

Cette catégorie reprend l'ensemble des personnes s'occupant de la maintenance des ATM.

Attributs physiques
- âge : compris entre 18 et 65 ans
- sexe : non pertinent
- pas de handicap physique contraignant (vue et motricité)

Attributs mentaux
- pas d'aptitude intellectuelle distinctive requise
- pas de déficience mentale
- motivation : motivation à la tâche positive et connaissances techniques suffisantes de l'ATM

Compétences
- expérience de la tâche : moyenne à grande
- expérience des ordinateurs : non nécessaire
- niveau de qualification : stage de formation à la manipulation des ATM
- langue : français

2.3 Définition des objectifs des utilisateurs

Nous définissons les objectifs des utilisateurs grâce aux diagrammes UML de Use Cases (UC) aux figures 2 et 3.

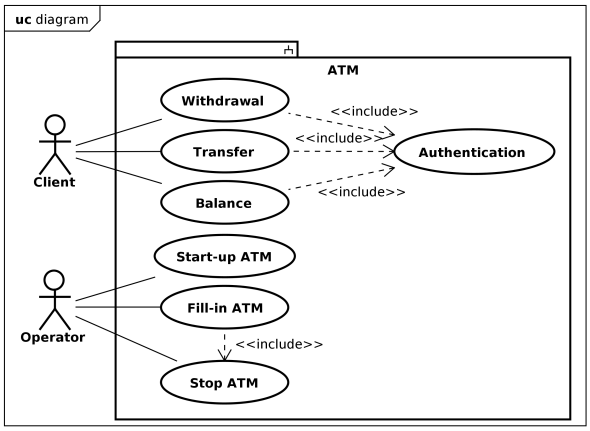

FIGURE 2 – Use Cases - ATM

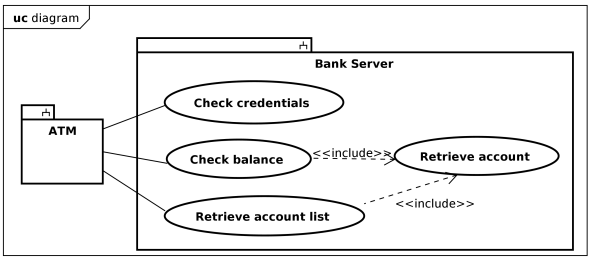

FIGURE 3 – Use Cases - Bank Sever

2.4 Scénarios d'utilisation

2.4.1 Identification du client

Nom Identifier client

Résumé Le système identifie le client et affiche le menu d'options.

Acteur Client

Pré-condition L'ATM est disponible et il affiche un message d'accueil.

Post-condition Le client est identifié et peut effectuer une transaction ou quitter.

Description

Client	**ATM**	**Serveur**
1. Le client insère sa carte dans le lecteur		
	2. Si la carte est reconnue, l'ATM lit le numéro de carte 3. L'ATM vérifie la date d'expiration 4. L'ATM demande au serveur si la carte a été perdue ou volée	
		5. Le serveur répond que la carte n'est ni perdue, ni volée
	6. L'ATM demande le code secret au client et affiche l'option "Annuler"	
7. Le client entre son code secret		
	8. L'ATM demande au serveur de vérifier le code secret	
		9. Le serveur valide le code secret 10. Le serveur envoie la liste des comptes accessibles avec cette carte
	11. L'ATM affiche le menu avec les cinq options ("Retrait", "Consulter solde", "Virement" et "Annuler")	

Alternatives

2.b Si le serveur ne reconnaît pas la carte, la carte est éjectée.
3.b Si le serveur détermine que la date de la carte est expirée, la carte est confisquée par l'ATM.
5.b Si la carte est déclarée volée ou perdue, la carte est confisquée par l'ATM.
6.b Si le client choisit "Annuler", le système annule la transaction et éjecte la carte.
9.b Si le code fourni par le client ne correspond pas au code maintenu par le système, le système redemande le code au client.
9.c Si le client fournit trois fois un mauvais code, la carte est confisquée par l'ATM.
Dans tous les cas : un message d'information adéquat est affiché sur l'écran (carte confisquée, mauvais code, ...).

2.4.2 Scénario du retrait d'argent

Nom Retirer du liquide

Résumé Le client retire une certaine somme d'argent d'un compte bancaire valide.

Acteur Client

Dépendance Use case *identifier client*

Pré-condition L'ATM est disponible et il affiche le menu d'options

Post-condition Le compte choisi a été débité de la somme demandée et, si demandé et si l'imprimante de l'ATM contient assez de papier, un reçu a été imprimé.

Description

Client	**ATM**	**Serveur**
1. Le client sélectionne l'option "Retrait", entre le montant et sélectionne le numéro de compte		
	2. L'ATM demande au serveur si le client dispose de suffisamment d'argent sur ce compte et si la limite journalière n'est pas dépassée	
		3. Le serveur fait les vérifications nécessaires et autorise le retrait
	4. L'ATM affiche un message indiquant que le retrait est accepté et propose d'imprimer un reçu	
5. Le client choisit de ne pas imprimer de reçu		
	6. L'ATM délivre l'argent 7. L'ATM éjecte la carte 8. L'ATM affiche un message d'accueil	

Alternatives

2.b Si l'ATM ne dispose plus assez d'argent, le système affiche un message d'erreur, éjecte la carte et arrête l'ATM.
2.c Si le serveur détermine que le solde du compte est insuffisant, il affiche un message d'erreur et éjecte la carte.
2.d Si le serveur détermine que la limite journalière a été dépassée, il affiche un message d'erreur et éjecte la carte.
5.b Si le client choisit d'imprimer un reçu, l'ATM imprime un reçu indiquant le montant retiré et le solde du compte. Si l'ATM n'a plus de papier, il prévient l'utilisateur ainsi que les suivants via un message lors de l'introduction de la carte.

2.4.3 Représentations alternatives des scenarii d'identification et de retrait

Une représentation alternative des scenarii de cas d'utilisation peut se faire avec les diagrammes de robustesse. Cette représentation semi-formelle a l'avantage d'offrir une vue plus concise et plus schématique des scenarii et est présentée aux figures 4 et 5. Les branchements sont représentés par des liens annotés par des conditions booléennes (de type *ok*, *nok* ou avec une condition plus explicite ou composée) et les flux alternatifs sont représentés avec des contrôleurs colorés en rouge (ou noircis en N/B).

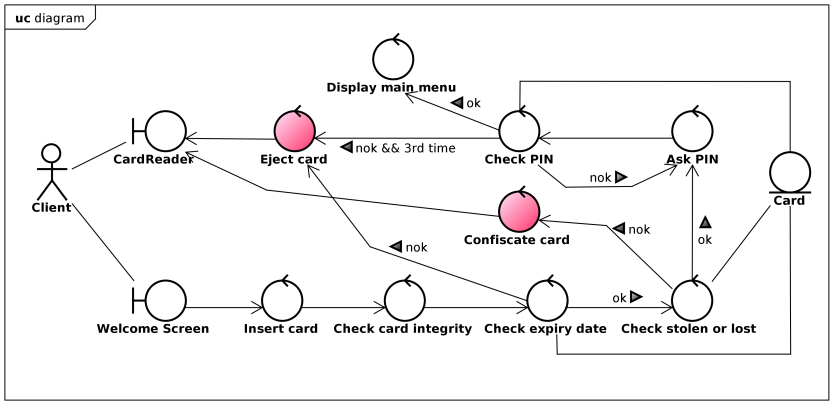

FIGURE 4 – Représentation alternative du scénario d'identification

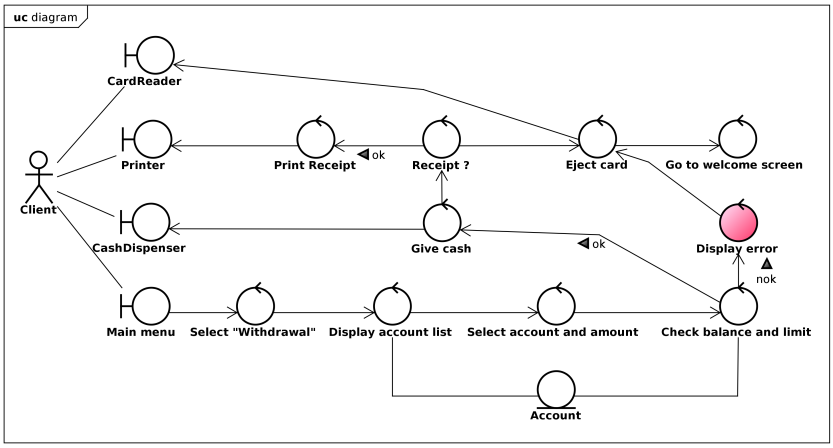

FIGURE 5 – Représentation alternative du scénario de retrait d'argent

2.5 Spécification des exigences non-fonctionnelles

Une série d'exigences non fonctionnelles sont importantes pour les commanditaires.

Sécurité Il est évident que la sécurité joue un très grand rôle ici. Il faudra premièrement que la communication des ATM avec le serveur soit sécurisée, c'est-à-dire qu'aucun ordinateur ne pourra se faire passer pour un ATM. Ceci ne devrait pas poser de problème, car le réseau utilisé est totalement privé. Il faudra également qu'aucune fraude soit permise lors de l'utilisation des ATM. Ceci sera garanti en limitant strictement les interactions possibles à celles décrites dans les scenarii d'utilisation.

Scalability Il est plus que probable que la banque installe dans le futur de nouveaux ATM. Le système devra pouvoir continuer à fonctionner dans les mêmes délais qu'actuellement.

Réactivité Le temps nécessaire à une transaction, si on considère que le client réponde instantanément aux questions du système, ne devra jamais excéder une minute, sauf dans le cas d'un virement, suivant les données à introduire ou dans le cas d'un problème matériel indépendant de notre système (panne physique d'un périphérique, par exemple).

Nous décidons également de nous imposer les contraintes, outils et standards suivants :

Modélisation Nous utiliserons le logiciels Astah* 6.7 community pour la modélisation UML et DB-main 9.1 pour la modélisation de bases de données. Tous deux ont l'avantage d'être gratuit et sont largement utilisés. Ils sont également disponibles sur Windows et Linux. Astah* permet de créer aisément des diagrammes et proposent un certains nombre de vérifications sémantiques des diagrammes UML. DB-main a l'avantage de générer du code SQL depuis un diagramme conceptuel de base de données avec peu (ou pas) d'interventions humaines.

Documentation Les documents seront écrits en Latex suivant un *template* prédéfini (celui-là même utilisé pour le présent document). Les comptes-rendus seront également formatés en LaTex.

Gestion de projet Le planning sera tenu à jour dans une feuille de calcul de la suite LibreOffice 4.

Tests fonctionnels Les tests fonctionnels, leurs résultats et les personnes assignées comme responsable de ceux-ci seront maintenus dans un feuille de calcul de la suite LibreOffice 4.

Versioning La maintenance des versions des fichiers de documentation, ainsi que des fichiers sources sera assurée par un serveur Subversion 1.6.

Peer-review Nous pratiquerons le *Peer-review* lors de la modélisation. Un analyste autre que la personne responsable du document de modélisation révisera le délivrables avant acceptation.

Test Les tests unitaires du logiciel seront systématiquement effectués au moins par le développeur responsable. Les tests fonctionnels seront quant à eux effectués par un développeur ou un analyste autre que la personne responsable de la fonctionnalité. Des tests de régression seront systématiquement appliqués lors de chaque intégration au produit *stable*.

2.6 Modélisation des processus métiers

Le cœur de métier du cas exposé à la figure 6 est l'utilisation d'un guichet automatique. Nous modéliserons donc le processus lié à ce guichet au moyen d'un diagramme d'activité. Notez que l'on ne tient pas compte des possibilités d'annulation à chaque étape. Nous nous concentrons ici sur *"comment l'ATM rencontre les besoins d'un client"*.

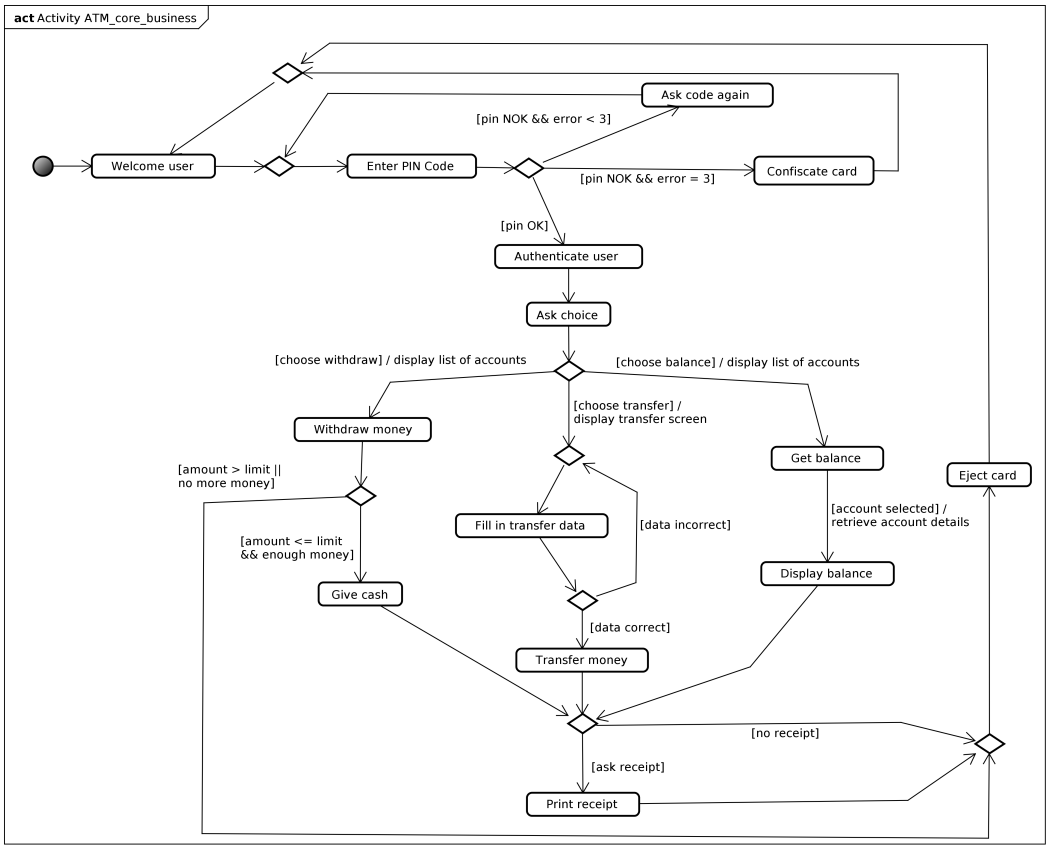

FIGURE 6 – Processus Métiers d'un ATM

2.7 Planning

Afin de mener à bien la réalisation de ce projet, nous avons défini un planning résumé à la figure 7. Il est présenté sous la forme d'un diagramme de *Gantt*. Ce type de représentation doit être complétée d'un tableau reprenant les noms des personnes assignées à chaque tâche ainsi que les estimations de temps pour chacune d'elles.

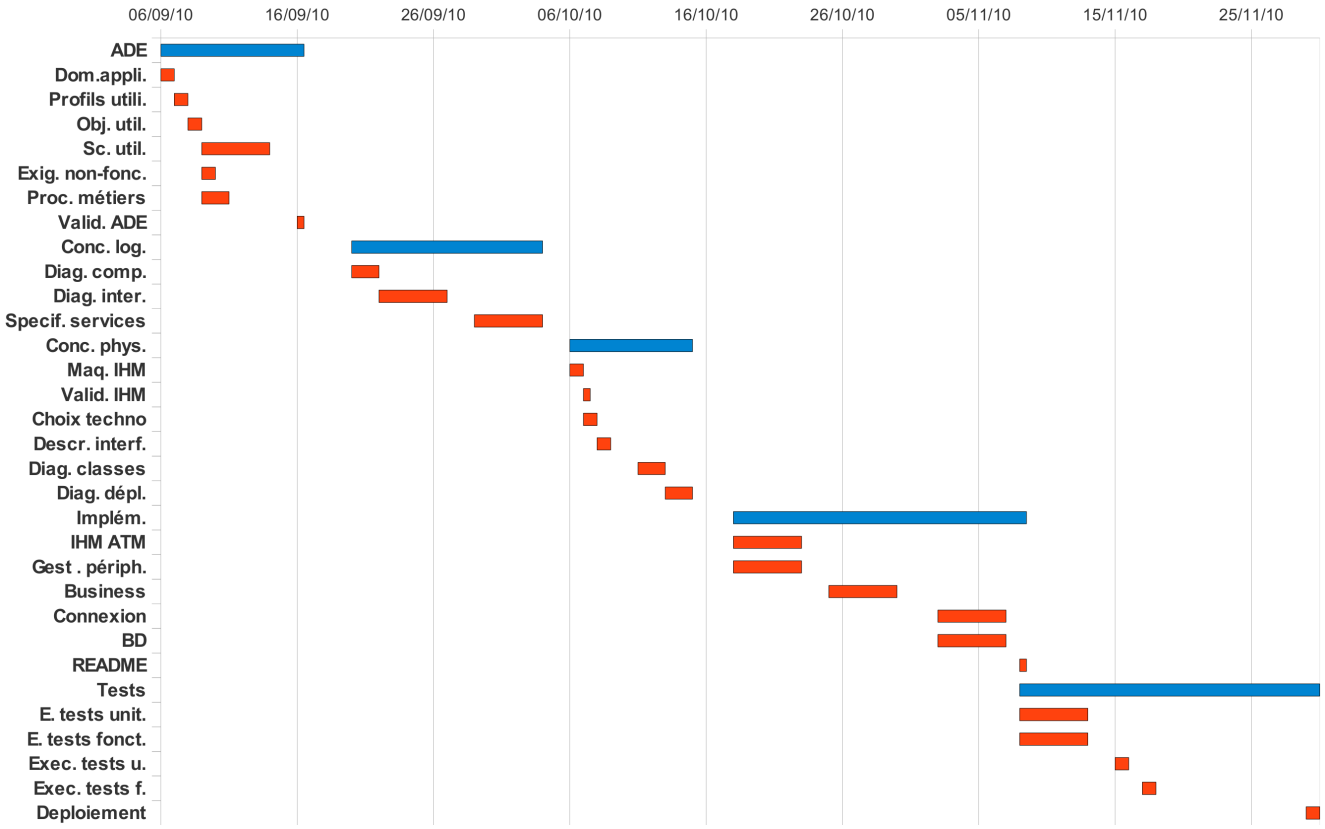

FIGURE 7 – Planning des différentes tâches

3 Conception logique

3.1 Diagrammes de robustesse

A partir de chacun des cas d'utilisation, nous élaborons un diagramme de robustesse permettant une première découpe naïve en composants. Nous le faisons ici pour l'UC *Retrait* à la figure 8.

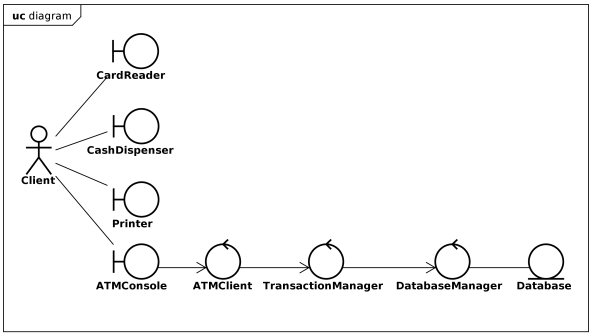

FIGURE 8 – Diagramme de robustesse - Retrait

3.2 Architecture logique

Le but sera ici de dégager un ensemble de composants, notamment à partir des diagrammes de robustesse élaborés précédemment. Nous utilisons les diagrammes de composants UML pour représenter un vue statique du système à la figure 9, décrivant l'architecture de haut niveau.

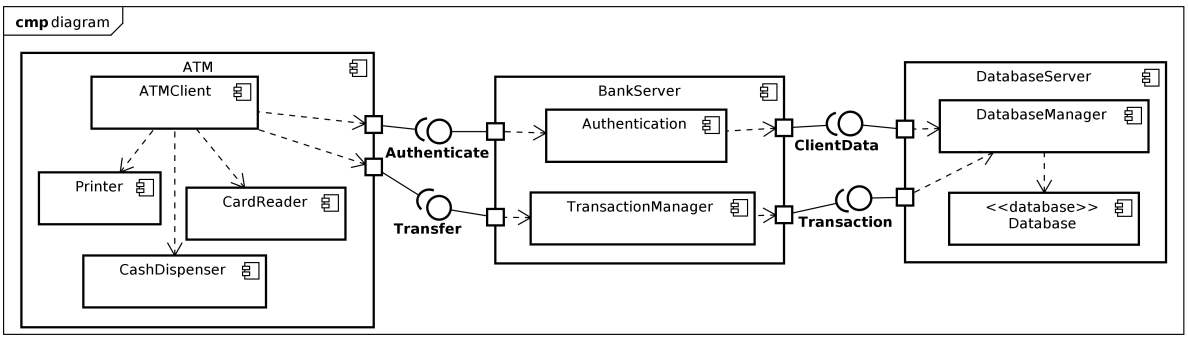

FIGURE 9 – Architecture logique - diagramme de composants

Nous séparons l'application en trois sous-systèmes : `ATM`, `BankServer` et `DatabaseServer`. Cette séparation est motivée par l'aspect géographique pour l'ATM et le serveur, et d'abstraction technologique pour la BD.

Le composant `ATMClient` encapsule le logiciel d'opération de l'ATM et dialogue avec le serveur de la banque. Il utilise également les périphériques disponibles, en l'occurrence l'imprimante de reçus, le lecteur de cartes et le distributeur de billets.

Du côté du sous-système `BankServer`, deux composants sont isolés : un composant d'identification (`Authentication`) et un composant se chargeant des transactions (`TransactionManager`). Le composant d'identification vérifie que le code fourni est le bon et que la carte n'est pas marquée comme *volée* ou *perdue*. Le composant de transaction se charge de la partie *business*, *i.e.* de la gestion des cas d'utilisation de retrait, de consultation de solde et de virement.

Les composants `Authentication` et `TransactionManager` utilisent tous les deux une base de données regroupant toutes les informations des clients, c'est-à-dire leurs comptes, les soldes de ces comptes, leurs limites journalières, les codes secrets des clients, etc. Néanmoins, nous créons un composant `DatabaseManager` fournissant une abstraction de la base de données aux autres composants, agrégeant les données qui leurs sont nécessaires et facilitant la mise en place d'une *réplication* ou le changement de technologie.

3.3 Diagrammes de séquence

Les diagrammes de séquence décrivent la dynamique du système en montrant les interactions entre composants. Ici, seul le flux normal du scénario de retrait (avec identification) est représenté. Dans une analyse complète, il faudrait expliciter tous les scenarii avec leurs alternatives (sur un même schéma, sur un schéma différent ou avec une note explicative, suivant la complexité de la situation).

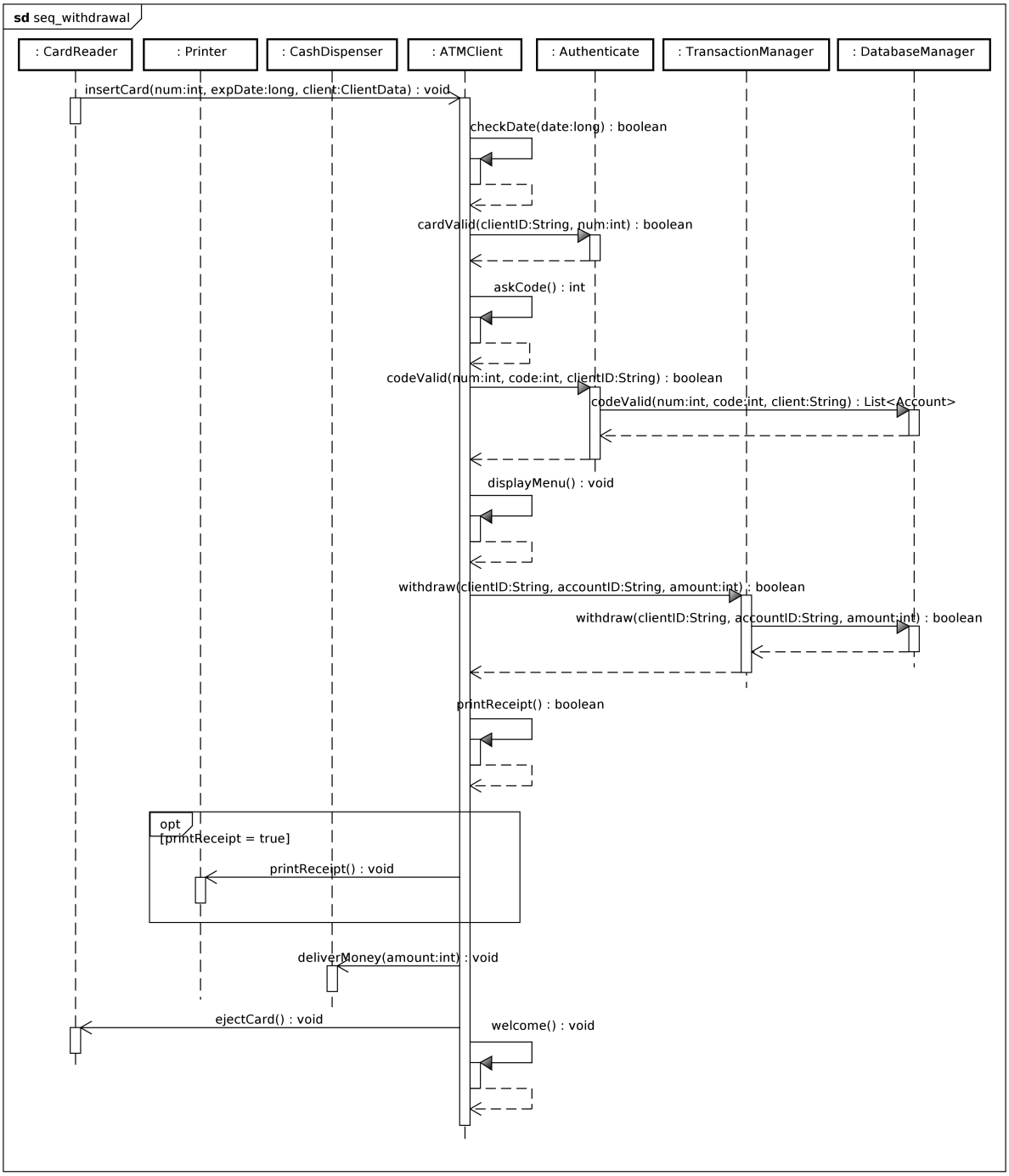

FIGURE 10 – Diagramme de séquence - *happy case* du retrait (avec connexion)

3.4 Spécifications des services

A titre d'exemple, nous présentons ici les spécifications des services `Authenticate.codevalid` et `ClientData.codeValid`. En préambule, nous définissons les types complexes que nous utiliserons dans nos spécifications (ainsi que, le cas échéant, les primitives de manipulation de ces structures). Les spécifications sont données en logique de premier ordre (ou calcul des prédicats) où nous ajoutons quelques types complexes et primitives associées [1].

3.4.1 Types complexes et primitives

- `ClientData` représente les données personnelles liées à un client

  id une chaîne de caractères unique composée d'exactement 20 caractères identifiant le client

  name chaîne de caractère contenant le nom du client

  surname ...

- `Account` représente un compte

  num chaîne de caractères unique identifiant le numéro de compte au format IBAN belge **BE***kk-BBBC-CCCC-CCKK* avec BE = code pays ; k = *checksum digit* ; B = code banque ; C = numéro compte ; K = *checksum* numéro compte (C)

  limit nombre entier donnant la limite journalière acceptable de ce compte

  balance nombre rationnel représentant le solde de ce compte

- `List<Account>` est une liste ordonnée d'`Account`
- `FullAccount` structure regroupant les données d'un client, un compte dont il est titulaire ainsi que les données du co-titulaire si pertinent.

  cdata du type `ClientData` regroupant les données personnelles du client

  cocdata du type `ClientData` regroupant les données personnelles du client co-titulaire du compte (peut être vide)

  account du type `Account` regroupant les données du compte détenu par le client repris dans `cdata` (possiblement co-detenu avec le client repris dans `cocdata`)

- `List<FullAccount>` est une liste ordonnée de `FullAccount`

3.4.2 Spécifications

(a) `Authentication implements Authenticate`

(i) `Authenticate.codeValid`

Signature `codeValid (num:int, code:int, clientID:string) : List<Account>`

Description Vérifie si le code entré par l'utilisateur est bien celui de la carte associée au client repris sur la carte

Dépendances externes

- `DatabaseManager.codeValid (num:int, code:int, clientID:String) : List<Account>`

Dépendances internes aucune

Pré

- `num` et `code` sont des entiers strictement positif et non mutables.
- `code` est inférieur à 9999.
- `clientID` contient l'identifiant d'un client et est non nul.

Post

- fait suivre l'appel à `ClientData.codeValid` et en renvoie le résultat. Retourne une liste vide si l'appel n'a pu se terminer correctement.

1. idéalement, il faut utiliser un langage du type *Object Constraint Language* (OCL), mais nous nous contenterons d'une formalisation en français et logique de premier ordre dans le cadre de ce laboratoire.

(b) `DatabaseManager implements ClientData, Transaction`

(i) `ClientData.codeValid`

Signature `codeValid (num:int, code:int, clientID:String) : List<Account>`

Description Vérifie si le code entré par l'utilisateur est bien celui de la carte associée au client repris sur la carte

Dépendances externes aucune

Dépendances internes aucune

Pré

- `num` et `code` sont des entiers strictement positif et non mutables.
- `code` est inférieur à 9999
- `clientID` contient l'identifiant du client et est non nul.

Post

- Une liste vide est renvoyée si `code` ne correspond pas au code maintenu dans la base de donnée pour la carte `num` et le client identifié par `clientID` ou si le client n'existe pas dans la base de données.
- Dans l'autre cas, la liste des comptes associés à la carte identifiée par `num` et au client identifié par `clientID` est renvoyée.

3.4.3 Diagrammes d'états des services

Les diagrammes d'états décrivent le comportement interne d'un service. A titre d'exemple, nous ne présenterons que le diagramme du composant `DatabaseManager` pour le service `codeValid` à la figure 11.

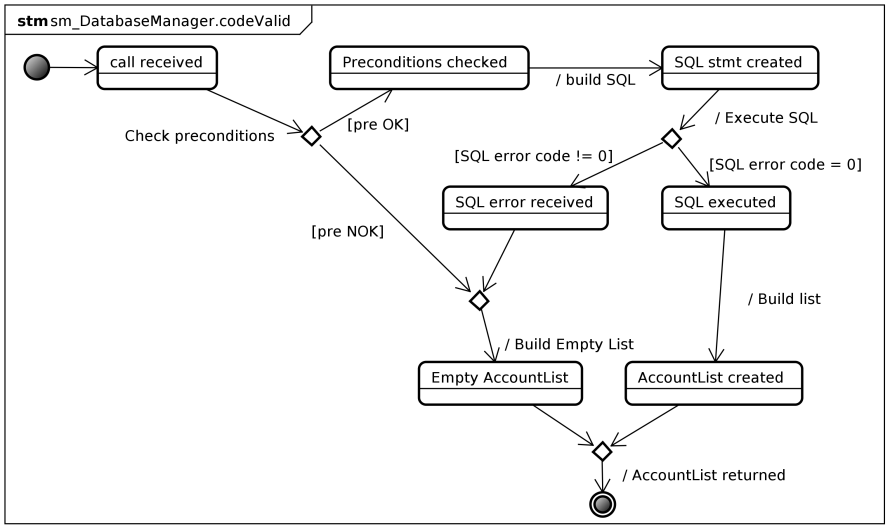

FIGURE 11 – Architecture logique - diagramme d'états de la validation de code secret

3.4.4 Conception logique de la base de donnée

Après avoir défini et spécifié les services, un schéma de base de données doit être produit afin de définir quelles seront les données qui devront être conservées de manière persistante. Ce schéma, au format *logique*, doit contenir toutes les données utiles, leurs types, les clés primaires et étrangères ainsi que les indexes.

4 Conception physique

Dans cette étape, nous expliciterons tout d'abord la métaphore qui sera utilisée pour l'interface homme/machine d'un ATM. Nous choisirons ensuite quelle(s) technologie(s) d'implémentation correspond(ent) le mieux à notre problème. Nous présenterons une vue plus détaillée du système à l'aide d'un diagramme de classes UML. Finalement, nous présenterons une structure de déploiement possible pour notre application.

4.1 IHM de l'ATM client

Le but ici est de montrer des représentations des écrans qui seront présentées au client ainsi que leur enchaînement (avec un diagramme d'états, par exemple). La construction de ceci se fait à partir des scenarii d'utilisation. Le but de cette maquette est de valider les use cases et les scenarii à travers les écrans de l'IHM [2], présentés aux figures 12 à 15.

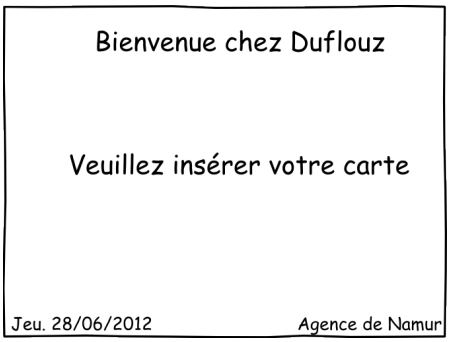

FIGURE 12 – IHM - Accueil

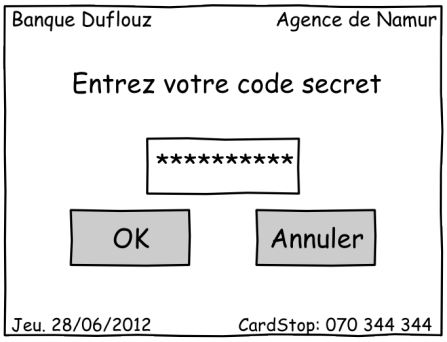

FIGURE 13 – IHM - Introduction du code

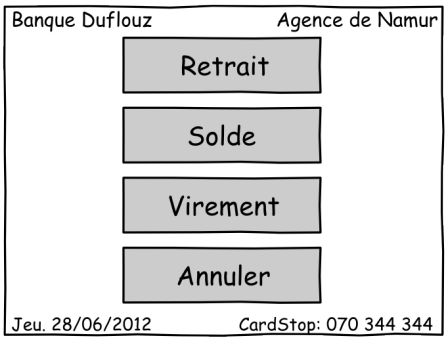

FIGURE 14 – IHM - Menu

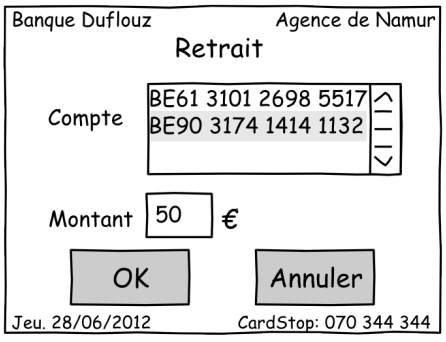

FIGURE 15 – IHM - Retrait

La figure 16 présente un exemple de message à l'intention de l'utilisateur. Ici, nous montrons la demande de confirmation d'un retrait.

FIGURE 16 – IHM - Exemple de message

L'enchaînement des interfaces est présenté à la figure 17 sous forme d'un diagramme d'état.

2. http ://www.onextrapixel.com/2010/09/29/40-brilliant-examples-of-sketched-ui-wireframes-and-mock-ups/

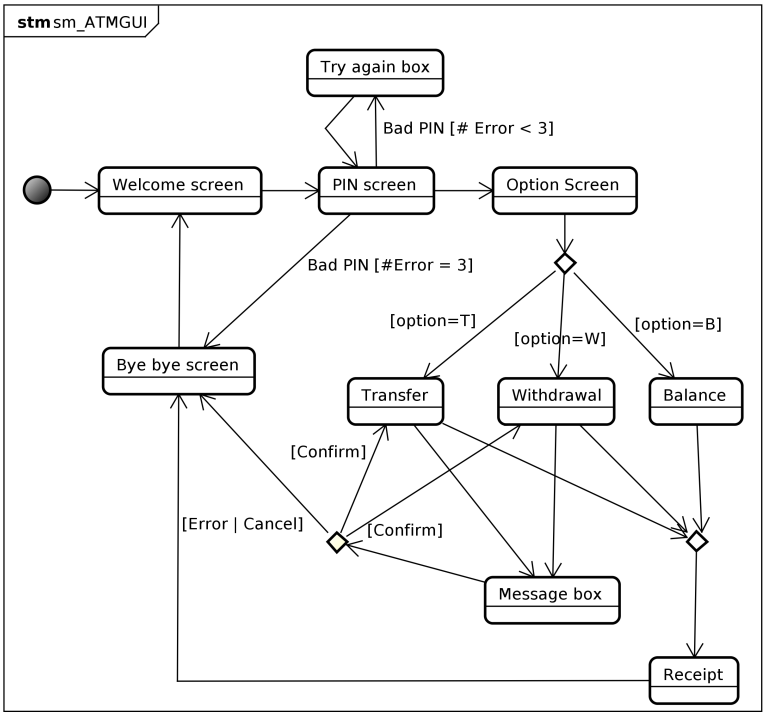

FIGURE 17 – IHM - diagramme d'enchaînement des interfaces

4.2 Choix des technologies cibles et standards

Premièrement, pour des raisons d'hétérogénéité (terminaux ATM *vs* serveur), nous utiliserons la technologie middleware Corba qui permettra aux composants de communiquer les uns avec les autres malgré qu'ils soient implémentés dans des langages différents spécifiques à la plate-forme sur laquelle ils seront installés. Néanmoins, pour cette étude de cas, nous implémenterons tous les composants en Java en utilisant les convention de code du langage Java [3].

La base de données étant déployée sur un SGBD relationnel, de type Oracle, supportant le SQL, nous utiliserons l'API JDBC [4] pour discuter entre les composants Java et le SGBD.

Nous utiliserons l'API `org.apache.log4j` pour tracer les exécutions du programme ainsi que les erreurs éventuelles. Entre autres avantages, cette API permet d'imprimer vers plusieurs types de supports (console, fichier, socket TCP), même simultanément. Elle permet également le raffinage des messages suivant leurs niveaux de *criticalité* (debug, warning, error, etc) tout en étant d'un *coût calcul* très faible.

Nous utiliserons l'environnement de développement intégré (IDE) Eclipse Kepler, car il est gratuit, est éprouvé à une échelle industrielle et dispose d'un *debugger* simple d'utilisation. Il permet également l'ajout de *plugins* variés et utiles pour, par exemple, le versioning, le *profiling*, l'optimisation de code, etc.

Pour les tests unitaires, nous utiliserons l'api `org.junit`. Cette bibliothèque permet d'écrire une fois du code de tests, mais d'en automatiser son exécution (à chaque intégration de nouveaux modules et corrections de bugs, notamment).

4.3 Descriptions IDL des interfaces

Le listing 1 présente le fichier IDL reprenant les différentes interfaces inter-composants.

```
1 /**
2 * This file defines all interfaces for DUFLOUZ ATM management.
3 */
```

3. version du 20 avril 1999, PDF disponible à l'adresse http://www.oracle.com/technetwork/java/codeconventions-150003.pdf

4. http://download.oracle.com/javase/tutorial/jdbc/overview/index.html

```
4  module be {
5    module duflouz {
6
7      module common {
8        /**
9        * Common data interface for DUFFLOUZ bank
10       * @author Piter de Vries
11       */
12       interface commonData {
13         /**
14         * Client personnal data
15         */
16         struct ClientData { /* skipping details */ };
17         /**
18         * Account-related information
19         */
20         struct Account { /* skipping details */ };
21         /**
22         * List of accounts
23         */
24         typedef sequence<Account> Accounts;
25         /**
26         * List of accounts attached to a client
27         */
28         struct ClientAccounts { /* skipping details */ };
29         /**
30         * Date structure
31         */
32         struct Date { /* skipping details */ };
33       };
34     };
35
36     module atm {
37       /**
38       * Card reader interface for DUFLOUZ ATM
39       * @author Gurney Halleck
40       */
41       interface CardReader {
42         /**
43         * Handle the insertion of a bank card
44         * @param num card ID number
45         * @param expDate card expiration date
46         */
47         void insertCard (in long num, in common::commonData::Date expDate);
48       };
49
50       /**
51       * Console interface for DUFLOUZ ATM
52       * @author Gurney Halleck
53       */
54       interface Console {
55         /**
56         * Sends the card code typed in by the client at the console
57         * @param code the bank card PIN code
58         */
59         void sendCode (in long code);
60       };
61     };
62
63     module bank {
64       module iface {
65         /**
66         * Interface for clients authentication from ATM
67         * @author Duncan Idaho
68         */
69         interface Authenticate {
70           /**
71           * Checks if a given code corresponds to the given card
72           * @param num a card ID number
73           * @param code the PIN code introduced by the user
74           * @param client the client ID
75           * @return the list of accounts attached to this card if the code is valid, an empty list
                 otherwise
76           */
77           common::commonData::ClientAccounts codeValid (in long num, in long code, in string client);
78         };
79
80         /**
81         * Interface for ATM account operations
82         * @author Duncan Idaho
83         */
84         interface Transfer {
85           /**
86           * Withdrawal management
87           * @param accountNbr account number to be modified
```

```
88            * @param amount amount to be substracted to the account
89            * @return true if withdrawal succeeded, false otherwise (daily limit or not enough money)
90            */
91           boolean withdraw (in long accountNbr, in long amount);
92           /* skipping others */
93         };
94       };
95     };
96
97     module database {
98       module iface {
99         /**
100        * Client data management
101        * @author Thufir Hawat
102        */
103        interface ClientData { /* skipping details */};
104        /**
105        * Transaction management
106        * @author Master Bijaz
107        */
108        interface TransactionData { /* skipping details */};
109      };
110    };
111  };
112 };
```

Listing 1 – Fichier IDL

4.4 Diagramme de classes

Nous présentons l'architecture physique de notre application grâce à un diagramme de classes à la figure 18. Nous nous concentrons ici sur la découpe en packages et classes, ainsi que de la justification de cette découpe. Il ne s'agit nullement de donner une nouvelle fois la sémantique du système ou des composants du systèmes, mais bien de décrire les choix liés à l'implémentation.

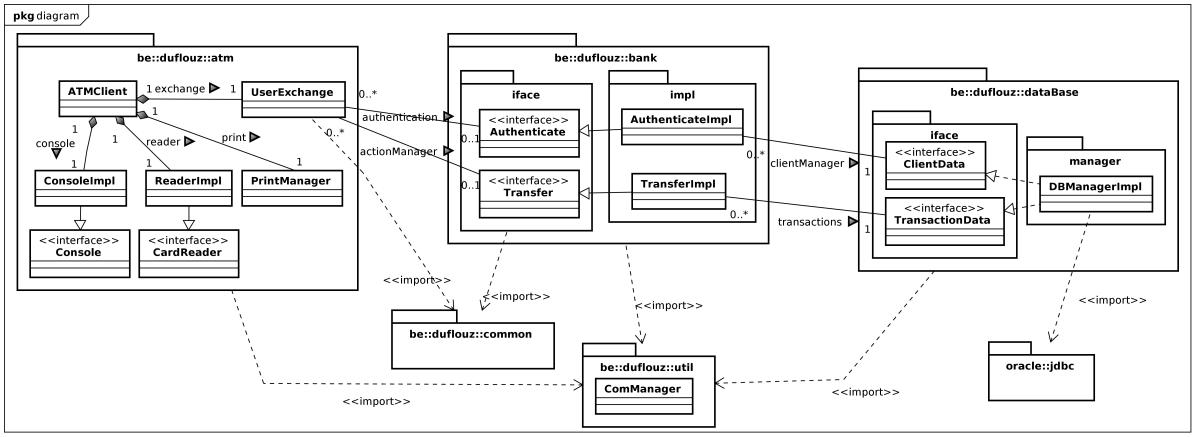

FIGURE 18 – Architecture physique - diagramme de classes

L'application est divisée en trois parties correspondant aux trois composants de haut niveau présentés à la figure 9. L'application côté ATM se trouve dans le package `be.duflouz.atm` et la classe principale (`main`) est `ATMCLient`. Les périphériques (imprimante, console et lecteur) sont gérés via des classes distinctes. La classe `UserExchange` agit comme une *façade* et centralise la communication avec le serveur de la banque.

Le serveur est implémenté dans le package `be.duflouz.bank`. Les interfaces Java (générées depuis le fichier IDL) liées aux services client sont séparées dans un package `bank.iface` (afin de faciliter le déploiement futur), les implémentations étant dans le package `bank.impl`. Les structures de données communes ont été placées dans le package `bank.common`.

Du côté du gestionnaire de la base de données, la même distinction que celle effectuée côté serveur a été faite entre interface et implémentation. L'implémentation utilisant l'API JDBC, la classe d'implémentation importe cette librairie.

Pour les aspects purement technique de la communication avec le middleware Corba, une classe générique de type *singleton* a été créée et placée dans le package `be.duflouz.util`. Cette classe regroupe des méthodes d'initialisation, inscription ou recherche d'un servant, etc.

4.5 Architecture de déploiement

Le diagramme de la figure 19 présente une instance possible de la configuration matérielle (ici, avec trois ATM). Des contraintes découvertes à ce niveau pourraient amener à revoir certains détails de l'architecture physique. Ces contraintes peuvent être la topologie du réseau, la puissance des machines, la bande passante du réseau,... UML étant très pauvre sémantiquement pour ce genre de diagramme, vous pouvez annoter chacun des éléments du diagramme avec ces contraintes.

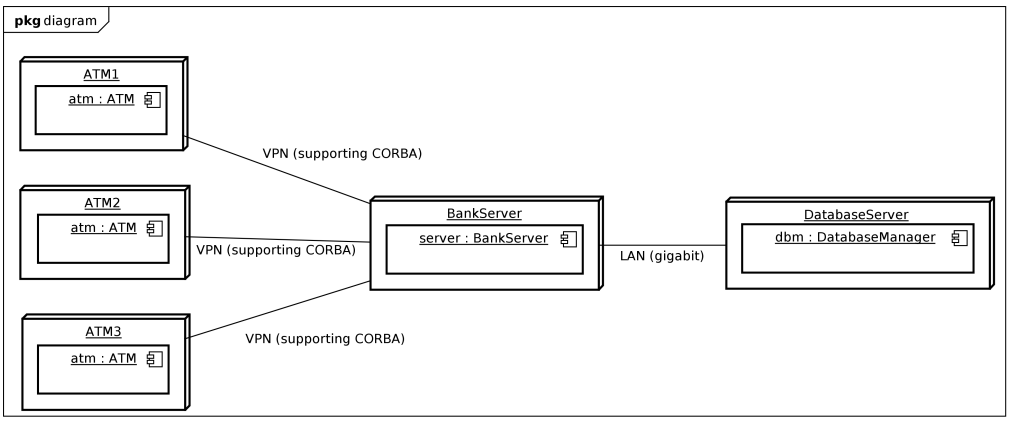

FIGURE 19 – Architecture de déploiement

5 Implémentation

En préambule à l'implémentation, il sera nécessaire de créer une batterie de tests aussi bien techniques (connexions au serveur, montée en charge,...) que fonctionnels. Pour ces derniers, on se basera sur les use cases et les scénarios. Pour les deux types de tests, nous utiliserons une feuille de tableur (type excel) avec :

1. identifiant du test
2. description ou lien vers l'UC concerné (si pertinent)
3. valeur(s) d'input (pour chaque paramètre) et/ou état du système (si pertinent)
4. output(s) calculé(s) et/ou état du système (si pertinent)
5. output(s) observé(s) et/ou état du système (si pertinent)
6. note(s) éventuelle(s)

Compte tenu des ressources disponibles et conformément au planning présenté à la section 2.7, l'application sera développée en deux étapes parallèles. D'un côté, les classes de l'IHM et des gestionnaires de périphériques (`Printer`, `CardReader`, etc) et de l'autre, les classes business et connexions distantes.

Une série de tests unitaires seront écrits [5] et utilisés afin de vérifier le comportement interne des classes d'implémentation. Afin de faciliter le développement, les tests, la maintenance et le monitoring, nous utiliserons le système de *log* `log4j` [6].

6 Conclusion

Le présent document donne un exemple incomplet d'une analyse des besoins et d'une conception logique et physique. Nous avons brièvement rappelé les buts de l'application à développer. Nous avons

5. voir notamment JUnit www.junit.org
6. http://logging.apache.org/log4j/1.2/

ensuite modélisé le domaine d'application et détaillé les termes importants liés au domaine. Nous avons défini le profil et les objectifs des utilisateurs (sous forme de *use cases*) avant de détailler leurs déroulements au travers de scenarii. Nous avons spécifié les exigences non-fonctionnelles. Nous avons modélisé le processus métier à l'aide d'un diagramme d'activité, ainsi que les diagrammes de robustesse relatifs aux différents cas d'utilisation. Nous avons ensuite donné une vue haut niveau de l'application via un diagramme de composants UML (les interfaces étant formalisées en IDL). Nous avons donné une description de la dynamique du système via des diagrammes de séquence UML exprimant les interactions (services) entre composants. Ces services ont ensuite été spécifiés à l'aide de pré/post-conditions. Nous avons exposé nos choix d'implémentation, de standards et d'outils. Nous avons présenté des *sketches* des interfaces graphiques. Nous avons présenté une vue statique de l'application avec un diagramme de classes UML. Nous avons un également donné une représentation possible du déploiement de l'application. Nous avons finalement exposé notre stratégie d'implémentation ainsi que nos conventions de tests.

Remerciements

Document réalisé sur base d'une première version écrite par Karl Noben et François Vermaut.

A Fichier README

A titre d'exemple, nous donnons au listing 2 un fichier README qui devrait toujours accompagner vos sources/binaires. Les informations importantes qui doivent notamment s'y retrouver sont les règles de copyright et redistribution, les instructions de déploiement et de démarrage de l'application, le contenu de votre archive/dossier, les dépendances externes ainsi que les auteurs et points de contact.

```
Forewords & copyright
---------------------
This program is free software; you can redistribute it and/or modify it under the terms
of the GNU General Public License as published by the Free Software Foundation; either
version 2 of the License, or any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY;
without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
See the GNU General Public License for more details. You should have received a copy of
the GNU General Public License along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Copyright (C) 2010 Emperor-God Leto II Atreides


Target system architecture
--------------------------
All Unix-based systems


How to run the application for the first time
---------------------------------------------
To run the application, open a terminal, go to "mainDirectory" and type ./run.sh


Content
-------
# directories

* mainDirectory: the main directory with executables
* mainDirectory/ATM: ATM related executables and specific libraries
* mainDirectory/DBServer: Database server and access manager executables and libraries
* mainDirectory/BankServer: Bank server executables
* lib: needed libraries
* conf: configuration files
* doc: code documentation

# main files

* in mainDirectory
run.sh: first-time executable. Used to initialise or reconfigure proper installation

* in mainDirectory/ATM
devices.sh: device manager executable (if started indepently for debugging purpose)
gui.sh: ATM main application (normal start)

* in mainDirectory/DBServer
run.sh: Database server subsystem

* in mainDirectory/BankServer
run.sh: Bank server subsystem


Dependencies
------------
# Oracle Java 1.6 or higher
# Apache log4j 1.2 or higher
# FUNDP myLogger v0.1 or higher


Author
------
Emperor-God Leto II Atreides (Arrakis - Atreides House - 2011 - mailto:leto@arrakis.du)
```

Listing 2 – Exemple de fichier README