Documentation technique V2

GROUPE 3A3
PHAKEOVILAY Andrew
ZEJNULLAHI Egxon
ZOULI-BARRERE Karim

Table des matières

1. Présentation de l'application

L'application permet de gérer des comptes bancaires de dépôt pour des clients. L'application possède actuellement quelques fonctionnalités tels que modifier les informations clients, créer un nouveau client, consulter un compte, créer un compte, créditer/débiter un compte, effectuer un virement, clôturer un compte, gérer les employés et rendre inactif un client à l'aide des boutons interactifs et des champs à remplir.

2. Répartition des fonctionnalités

Afin d'avancer plus rapidement nous nous sommes séparés les fonctionnalités à coder de la façon suivante :

Egxon Zejnullahi:

•Créer un compte

•Gérer (faire le « CRUD ») les employés (guichetier et chef d'agence)

• Simulation Emprunt et Simulation Assurance

Karim Zouli-Barerre :

•Créditer/débiter un compte (java et BD avec procédure stockée)

• Débit exceptionnelle

• Création d'un PDF

Andrew Phakeovillay :

• Effectuer un virement de compte à compte

• Clôturer un compte

• Le CRUD des prélèvements automatiques

3. Architecture

Le projet de l'application est travaillé sous Eclipse où nous pouvons catégoriser les classes sous des packages afin d'avoir une meilleure visibilité et un meilleur repère pour travailler.

Architecture

Une partie java avec des packages java et une partie avec des packages pour la base de donnée.

Nous avons un fichier .jar nommé ojdbc6.jar qui est une librairie.

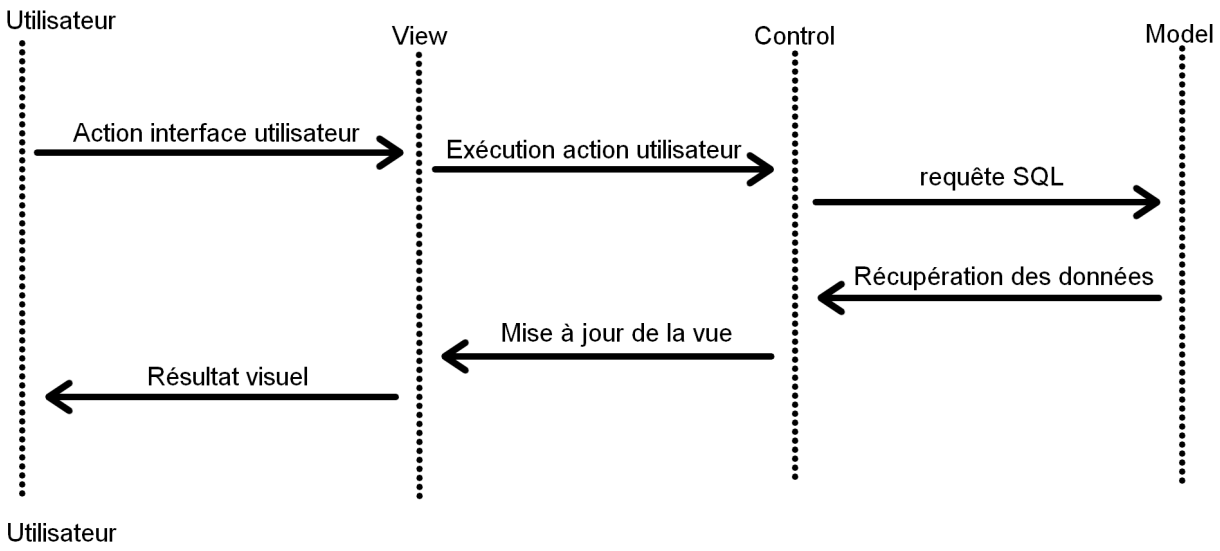
Nous avons des packages :

- application qui contient tous les packages dans lesquelles on retrouve les .java liés à l'application.
- application.control qui contient des contrôleurs liés aux fichiers FXML permettant aussi d'accéder aux données
- application.tools qui contient les fichiers .java essentiel au fonctionnement d'application.view et application.control, car dans application.tools permet le fonctionnement des contrôleurs pour afficher l'interface.
- application.view où il y a les fichiers fxml permettant de faire des schémas à l'aide de Scene Builder et il y a les classes contrôleurs permettant de faire fonctionner des contrôles de l'interface à l'aide des fonctions
- model.data qui contient les différents constructeurs utiles pour intéragir avec la base de donnée. Chaque constructeur correspond à une table sur la base de donnée.
- model.orm qui contient les classes java connectés à la base de donnée, les classes effectuent des requêtes SQL et communique avec la classe model.data.
- model.exception gère les exceptions lorsque un problème intervient avec la base de donnée quand une erreur survient lors de l'exécution des classes du package model.orm

3.1 Modèle MVC

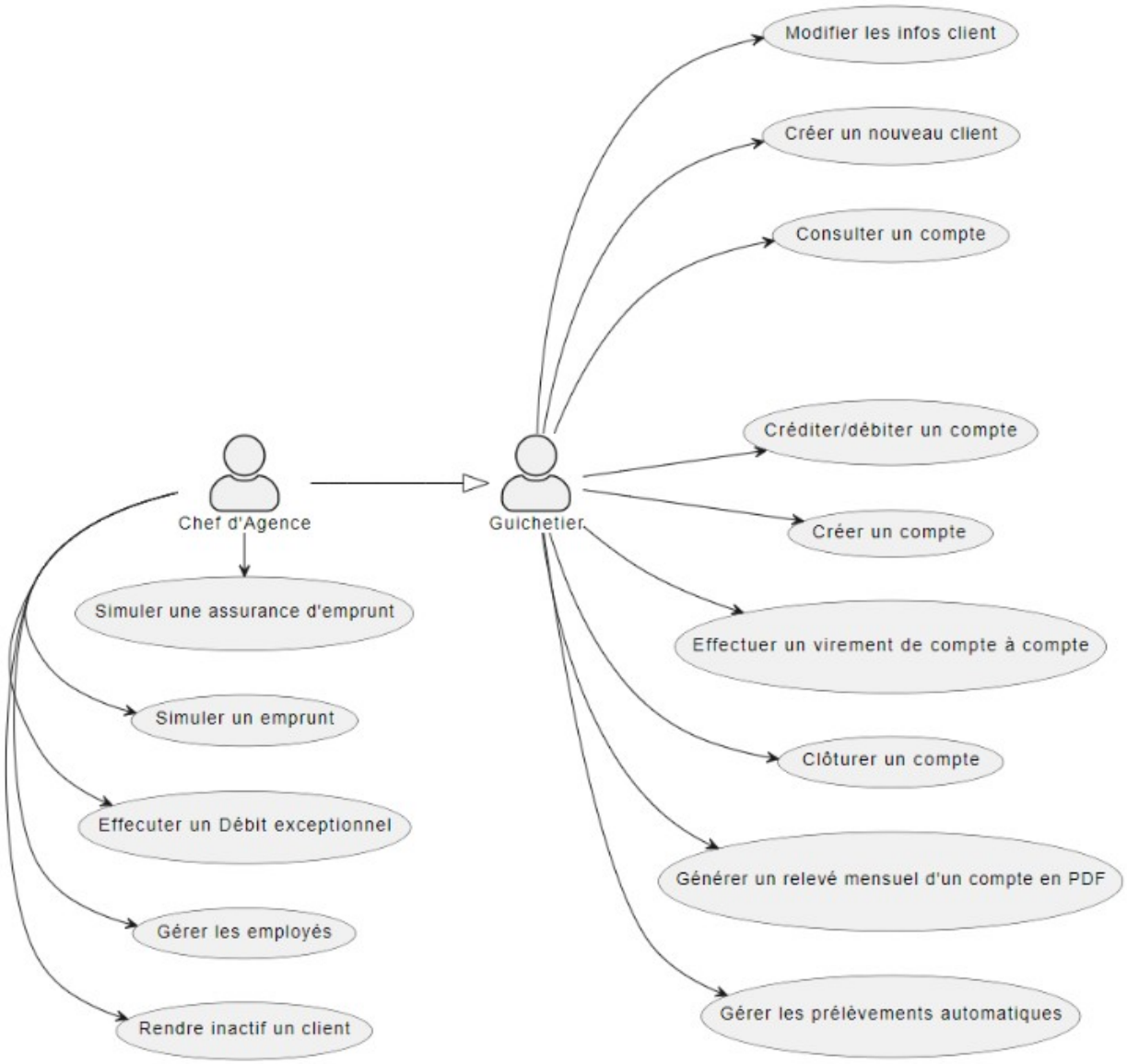
L'application utilise le modèle MVC (Model View Controller). L'utilisateur interagit les informations avec la vue qui transfère les actions au contrôleur, le contrôleur va ensuite convertir les actions pour qu'elles interagissent avec la base de données dans la partie model et le model exécute la requête pour l'intégrer dans la base de données. Nous pouvons voir le fonctionnement par le schéma ci-dessous :

MVC


Utilisateur
View
Control
Model
Action interface utilisateur
Exécution action utilisateur
requête SQL
Récupération des données
Mise à jour de la vue
Résultat visuel
Utilisateur

4. Présentation et explication de tous les diagrammes demandés

4.1 Use Case de l'application



Nous avons deux types d'utilisateurs :

Le guichetier peut :

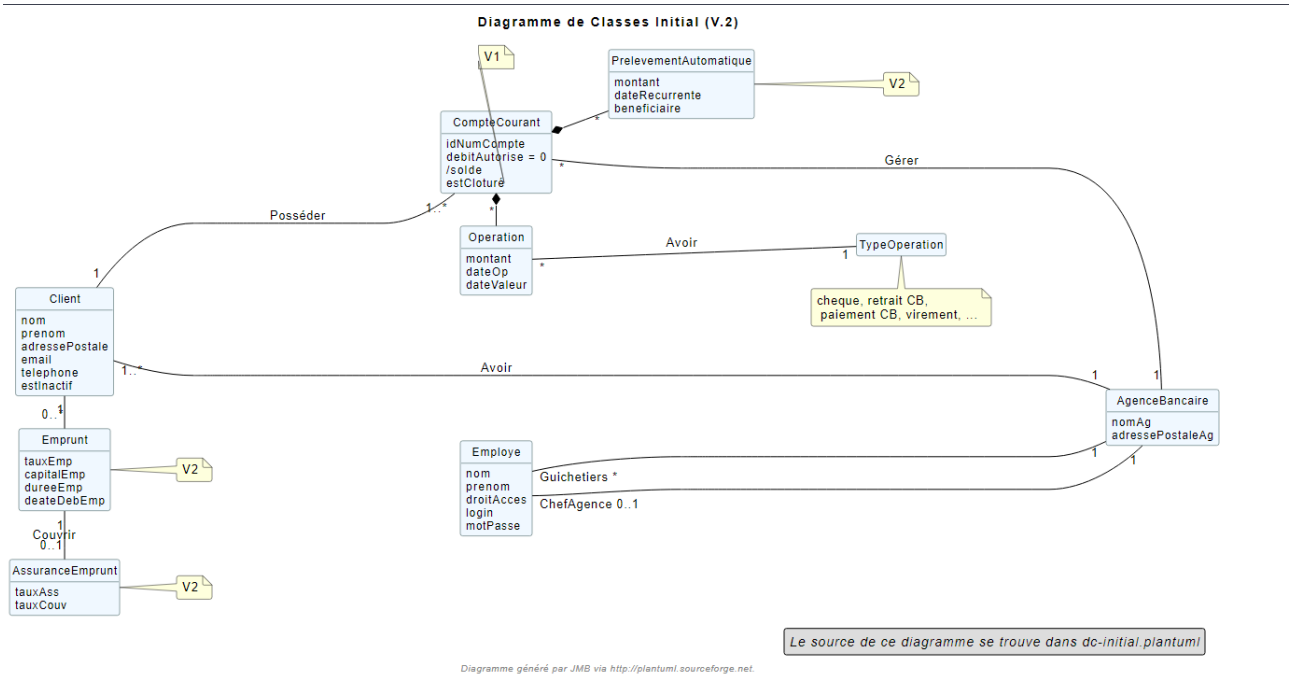
- Modifier les infos client
- Créer un nouveau client
- Consulter un compte
- Créditer/débiter un compte
- Créer un compte

- Effectuer un virement de compte à compte
- Clôturer un compte

Le chef d'agence en plus des fonctionnalités du guichetier, il peut :

- Gérer les employés
- Rendre inactif un client
- Effectuer un débit exceptionnelle
- Simuler un emprunt
- Simuler une assurance d'emprunt

4.2 Diagramme des Cas d'utilisation initial (V.2)



Une agence bancaire dispose de plusieurs employés

Une agence bancaire a un ou plusieurs clients

Une agence bancaire gère plusieurs comptes courants

Un employé peut être un chef d'agence ou un guichetier selon ses droits d'accès

Un client a une agence bancaire

Un client possède un ou plusieurs comptes courants

Un compte courant est géré par un agence bancaire

Un compte courant est composé d’opérations

Une opération a un type d’opération

Un client peut avoir 0 ou 1 emprunt et un emprunt peut couvrir une assurance d’emprunt

Un compte courant est composé de plusieurs prélèvements automatiques

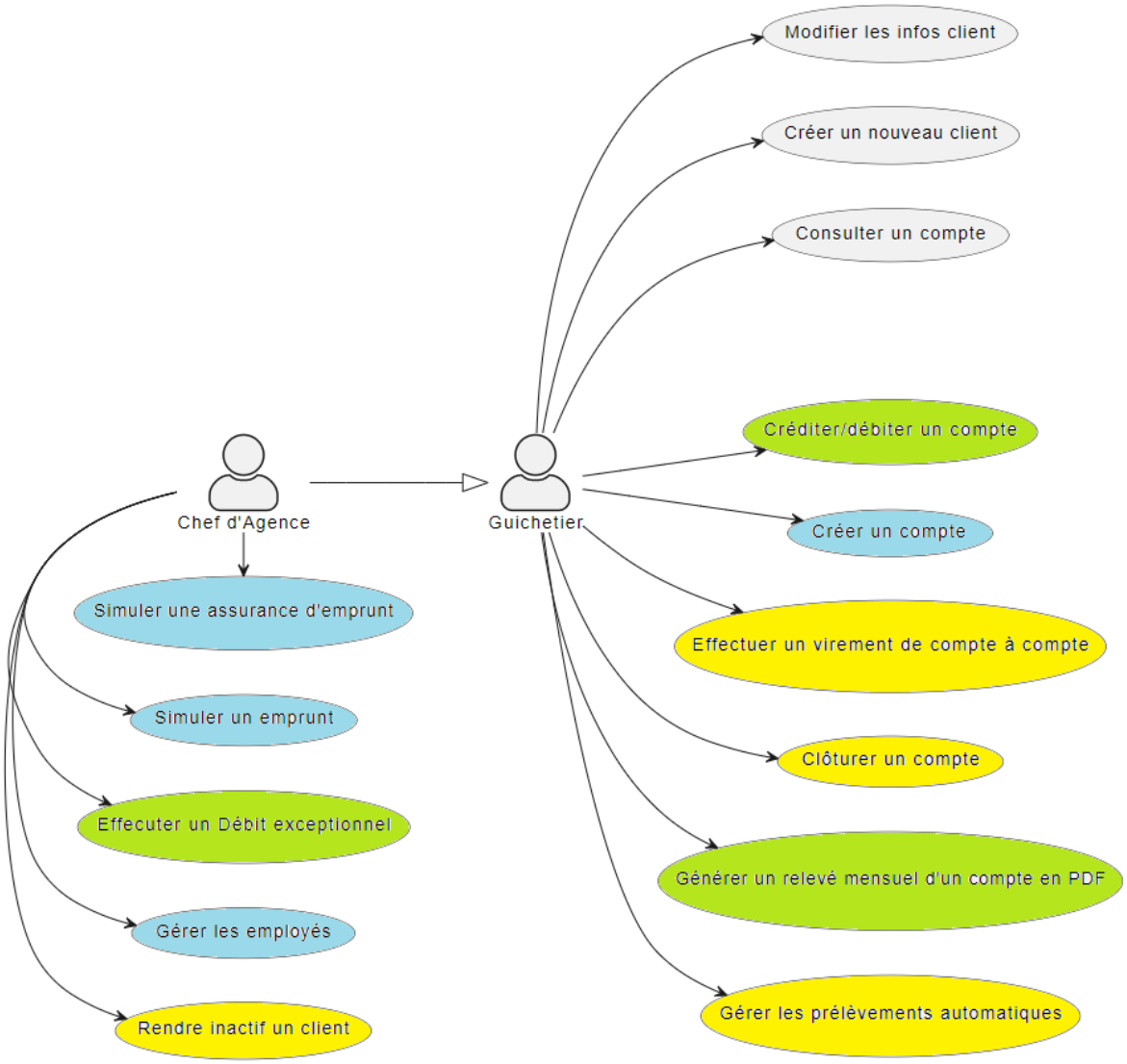
Nous pouvons identifié l’employé en fonction de ses droits d’accès (chef d’Agence ou guichetier)

Si le compte courant est inactif, nous pouvons pas accéder aux opérations et aux prélèvements

5. Fonctionnalités

Pour chaque fonctionnalité : en les expliquant Dans chaque partie : qui est le développeur responsable

- Partie de use case réalisé - scénarios éventuels
  - Les parties en bleu ont était réalisé par Egxon Zejnullahi
  - Les parties en vert ont était réalisé par Karim Zouli-Barrere
  - Les parties en jaune ont était réalisé par Andrew Phakeovilay
  - Le reste était déjà existant dans la v0



Andrew PHAKEOVILAY

V0

5.1 Rendre un client inactif

La fonctionnalité était déjà existant mais elle ne fonctionnai qu'avec le bouton modifier client donc il fallait coder un bouton permettant de désactiver un client directement.

Use case :

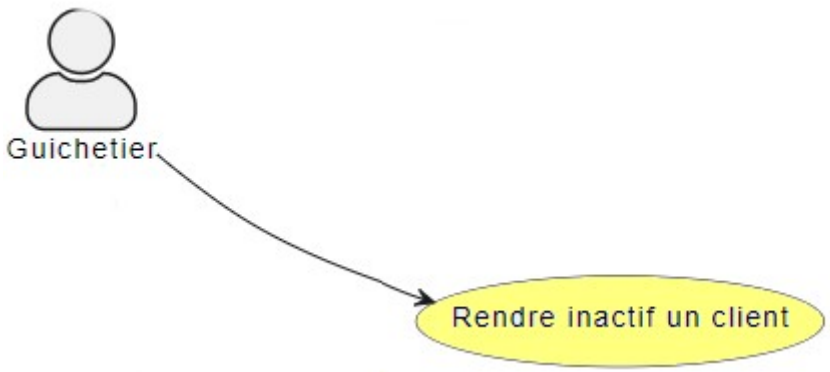
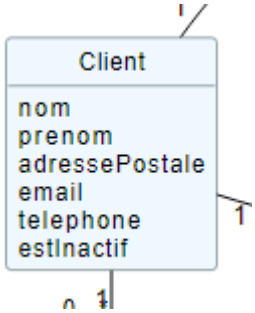


Diagramme de classes données nécessaires :



Classes impliquées dans chaque package :

Package application :
- DailyBankState.java

Package application.control :
- ClientsManagement.java

Package application.view :
- ClientsManagementController.java

Package model.orm :
- AccessClient.java
- LogToDatabase.java

Package model.data :
- Client.java

copies écrans cf. Doc Utilisateur V2

Extraits de code significatifs

```java
@FXML
private void doDesactiverClient() {
    int selectedIndice = this.lvClients.getSelectionModel().getSelectedIndex();
    if (selectedIndice >= 0) {
        Client client = this.olc.get(selectedIndice);
        Alert dialog = new Alert(AlertType.CONFIRMATION);
        dialog.setTitle("Confirmation");
        dialog.setContentText("Voulez-vous vraiment rendre inactif ce client ?");
        dialog.setHeaderText("Rendre inactif ?");
        dialog.initOwner(this.primaryStage);
        Optional<ButtonType> reponse = dialog.showAndWait();
        if (reponse.get() == ButtonType.OK) {
            this.cm.rendreInactif(client);
            this.btnDesactClient.setDisable(true);
            this.btnComptesClient.setDisable(true);
        }
    }
}
```

C'est l'action d'appuyer sur le bouton désactiver un client, ça affiche un pop-up de confirmation, si on appuit sur OK, le compte devient inactif et les boutons désactiver client et comptes client deviennent inactifs.

V1

5.2 Rendre compte inactif

La fonctionnalité rendre un compte inactif est de changer l'état du compte actif en inactif en mettant son solde à 0, nous débitons ou créditons afin de le mettre à 0. Lorsque le compte est inactif, nous ne pouvons pas accéder aux opérations.

Use case :

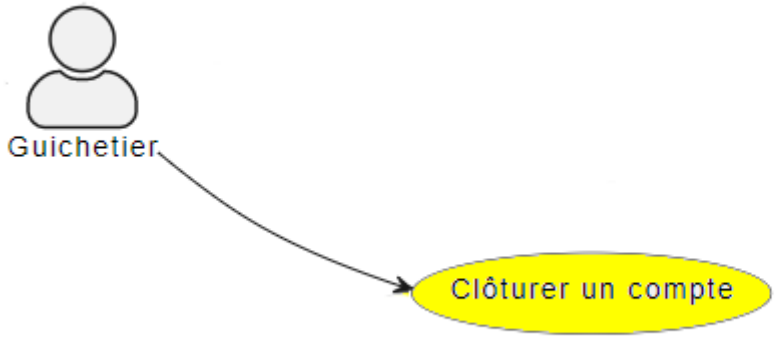
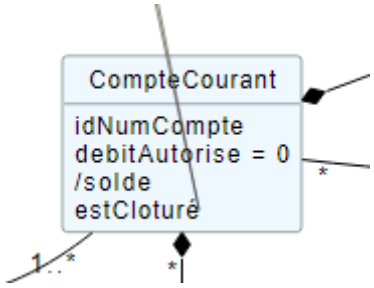


Diagramme de classes données nécessaires :



Classes impliquées dans chaque package :

Package application :
- DailyBankState.java

Package application.control :
- ComptesManagement.java

Package application.view :
- ComptesManagementController.java

Package model.orm :
- AccessCompteCourant.java

- AccessOperation.java
- LogToDatabase.java

Package model.data :

- CompteCourant.java
- Client.java

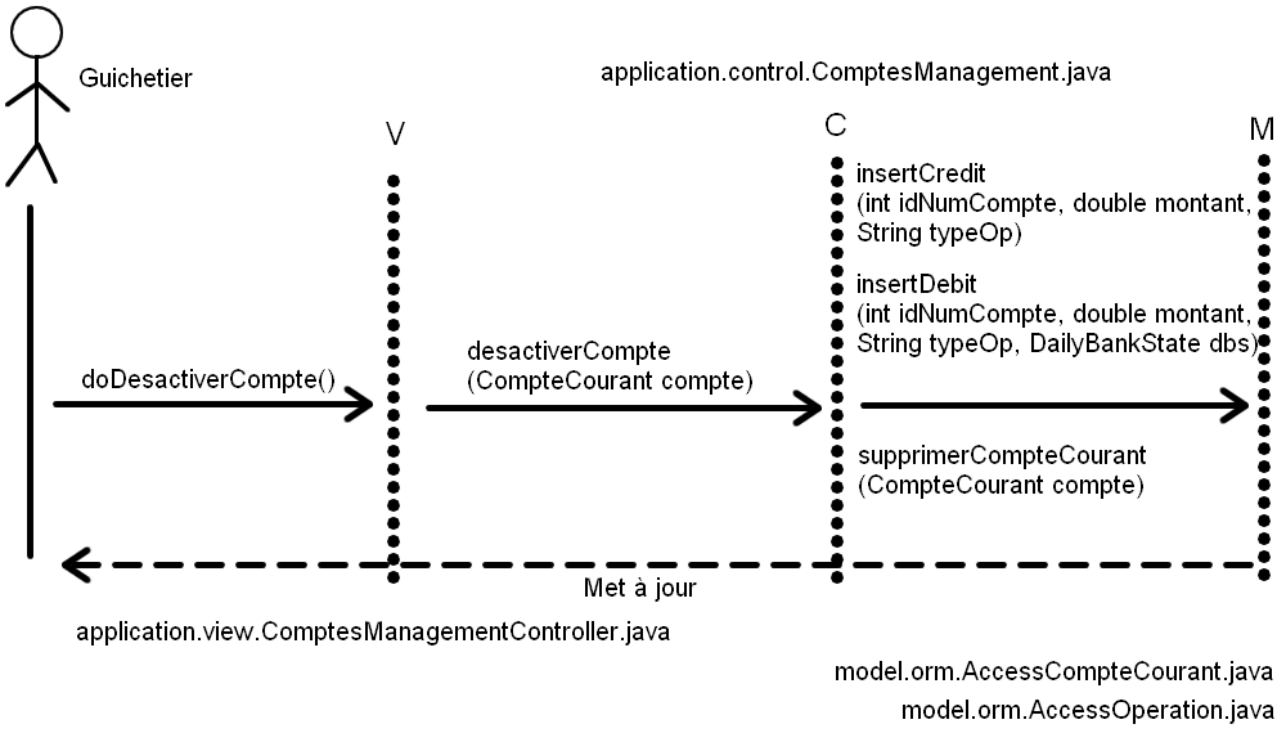
copies écrans cf. Doc Utilisateur V2

Extraits de code significatifs

```java
@FXML
private void doDesactiverCompte() {
    int selectedIndice = this.lvComptes.getSelectionModel().getSelectedIndex();
    if (selectedIndice >= 0) {
        CompteCourant cpt = this.olCompteCourant.get(selectedIndice);
        Alert dialog = new Alert(AlertType.CONFIRMATION);
        dialog.setTitle("Confirmation");
        dialog.setContentText("Voulez-vous vraiment clôturer le compte ?");
        dialog.setHeaderText("Clôturer le compte ?");
        dialog.initOwner(this.primaryStage);
        Optional<ButtonType> reponse = dialog.showAndWait();
        if (reponse.get() == ButtonType.OK) {
            this.cm.desactiverCompte(cpt);
            loadList();
        }
    }
}
```

C'est l'action d'appuyer sur le bouton désactiver compte d'un client, ça affiche un pop-up de confirmation, si on appuit sur OK, le compte devient inactif

Diagramme de séquence :



5.3 Virement compte à compte d'un client

Le virement compte à compte d'un client consiste à débiter un compte actif d'un client et de créditer vers un compte actif du même client.

Use case :

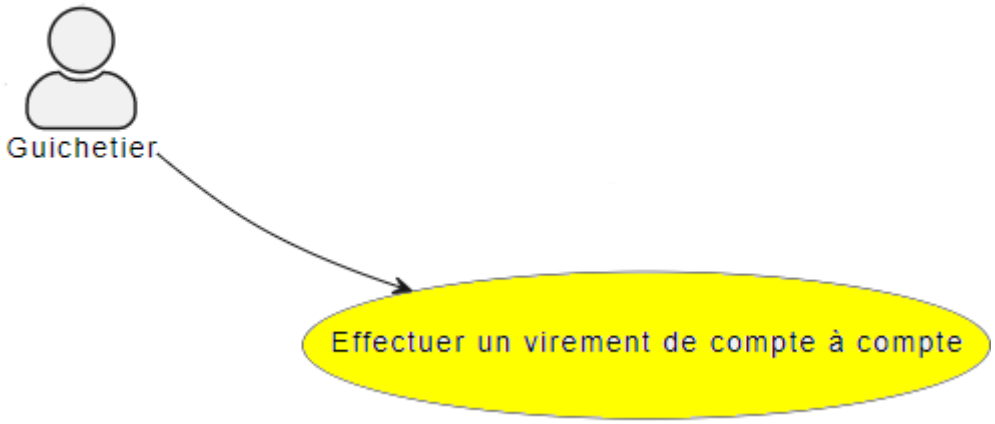
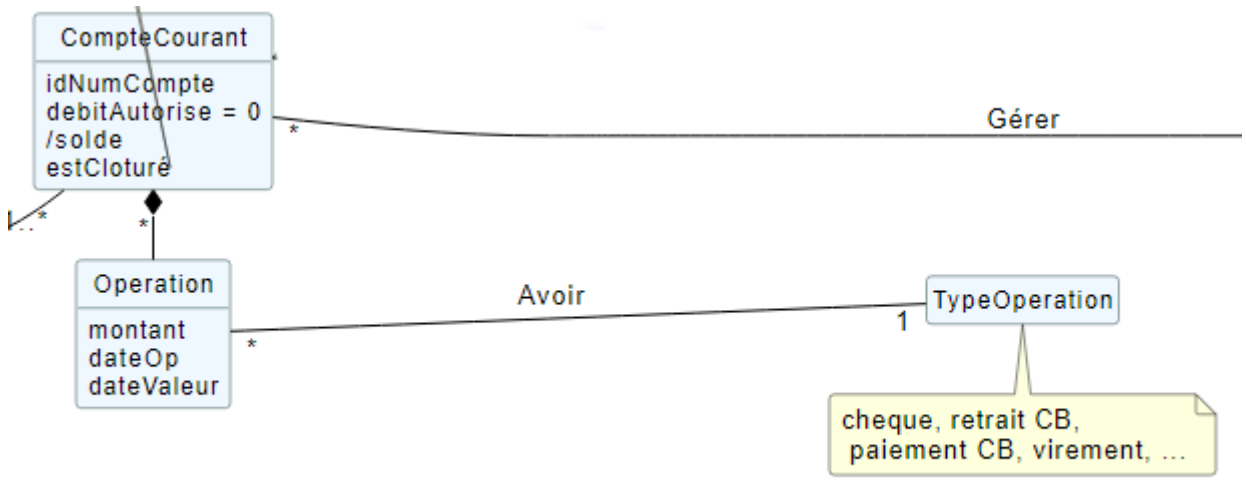


Diagramme de classes données nécessaires :



Classes impliquées dans chaque package :

Package application :

- DailyBankState.java

Package application.control :

- OperationsManagement.java
- ViremmentEditorPane.java
- ComptesManagement.java

Package application.view :

- ViremmentEditorPaneController.java

Package model.orm :

- AccessOperation.java
- LogToDatabase.java

Package model.data :

- Client.java
- CompteCourant.java

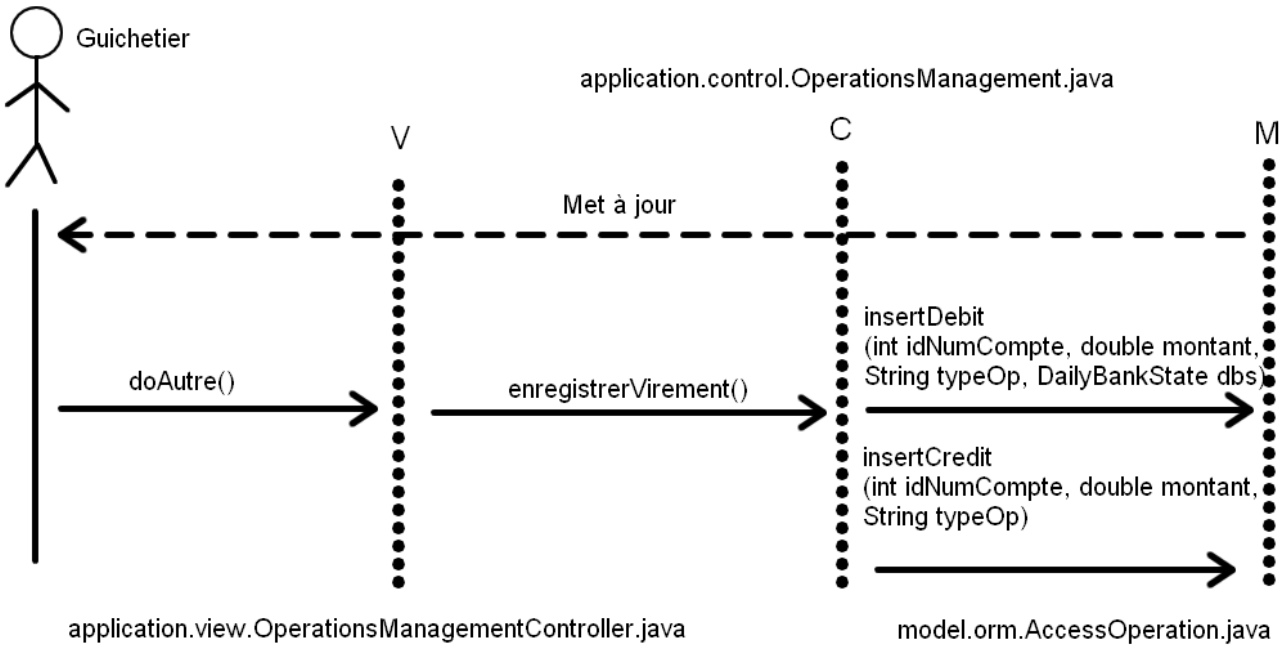
copies écrans cf. Doc Utilisateur V2

Extraits de code significatifs

```
    @FXML
    private void doVirement() {
        double montant;
        int selectedIndice = this.autreComptes.getSelectionModel().getSelectedIndex();
        if (selectedIndice >= 0) {
            this.compteChoisi = this.olCompteCourants.get(selectedIndice);
            try {
                montant = Double.parseDouble(this.textArgent.getText().trim());
                if (montant <= 0)
                    throw new NumberFormatException();
            } catch (NumberFormatException nfe) {
                this.textArgent.requestFocus();
                return;
            }
            if (this.compteConcerne.solde - montant < this.compteConcerne.debitAutorise) {
                Alert dialog = new Alert(AlertType.INFORMATION);
                dialog.setTitle("Erreur");
                dialog.setContentText("- Cpt. : " + this.compteConcerne.idNumCompte + "  " + String.format(Locale.ENGLISH, "%12.02f", this.compteConcerne.solde) + "  /  " + Strin
                dialog.setHeaderText("Dépassement du découvert !");
                dialog.initOwner(this.primaryStage);
                dialog.showAndWait();
                return;
            }
            this.operationResultat[0] = new Operation(-1, montant, null, null, this.compteConcerne.idNumCompte, "Virement Compte à Compte");
            this.operationResultat[1] = new Operation(-1, montant, null, null, this.compteChoisi.idNumCompte, "Virement Compte à Compte");
            this.primaryStage.close();
        }

    }
}
```

Cette fonction permet d'effectuer un virement qui a pour contrainte, un montant qui est numérique, qui ne dépasse pas le découvert du nouveau sinon pop-up information erreur et la contrainte d'un compte destinataire selectionné. Une fois les contraintes respectées, stocke dans la table operationResultat, le virement, l'un le débit du compte de départ et l'autre le crédit du compte de destination.

Diagramme de séquence :



V2

5.4 Le CRUD des prélèvements automatiques

Le crud des prélèvements s'agit de lire, écrire, modifier et supprimer les prélèvements automatiques en les assignant à un compte d'un client.

Use case :

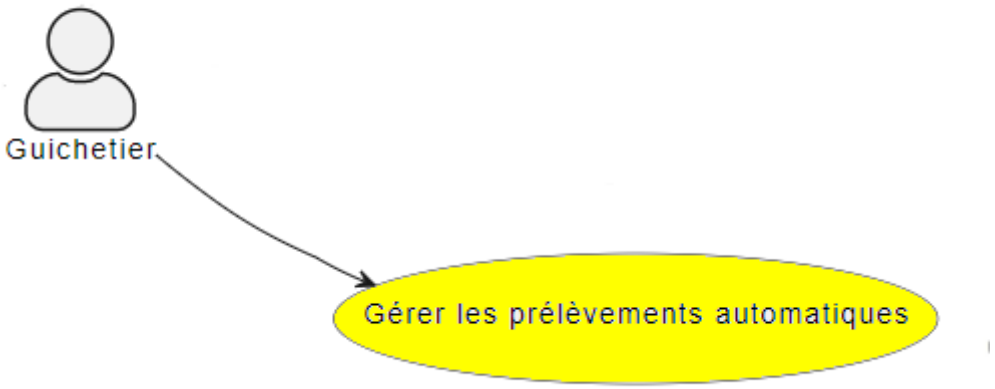
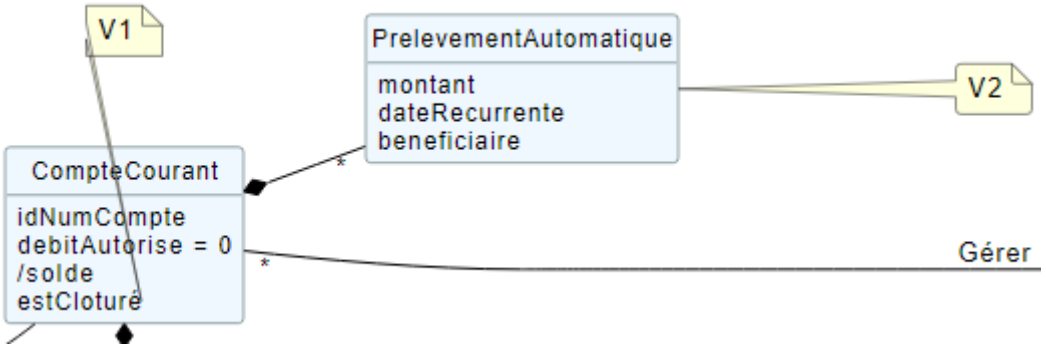


Diagramme de classes données nécessaires :



Classes impliquées dans chaque package :

Package application :
- DailyBankState.java

Package application.control :
- PrelevementsManagementController.java
- PrelevementEditorPaneController.java

Package application.tools :
- EditionMode.java

Package application.view :
- PrelevementEditorPaneController.java
- PrelevementsManagementController.java

Package model.orm :
- AccessPrevelement.java
- LogToDatabase.java

Package model.data :
- Prelevement.java
- Client.java
- CompteCourant.java

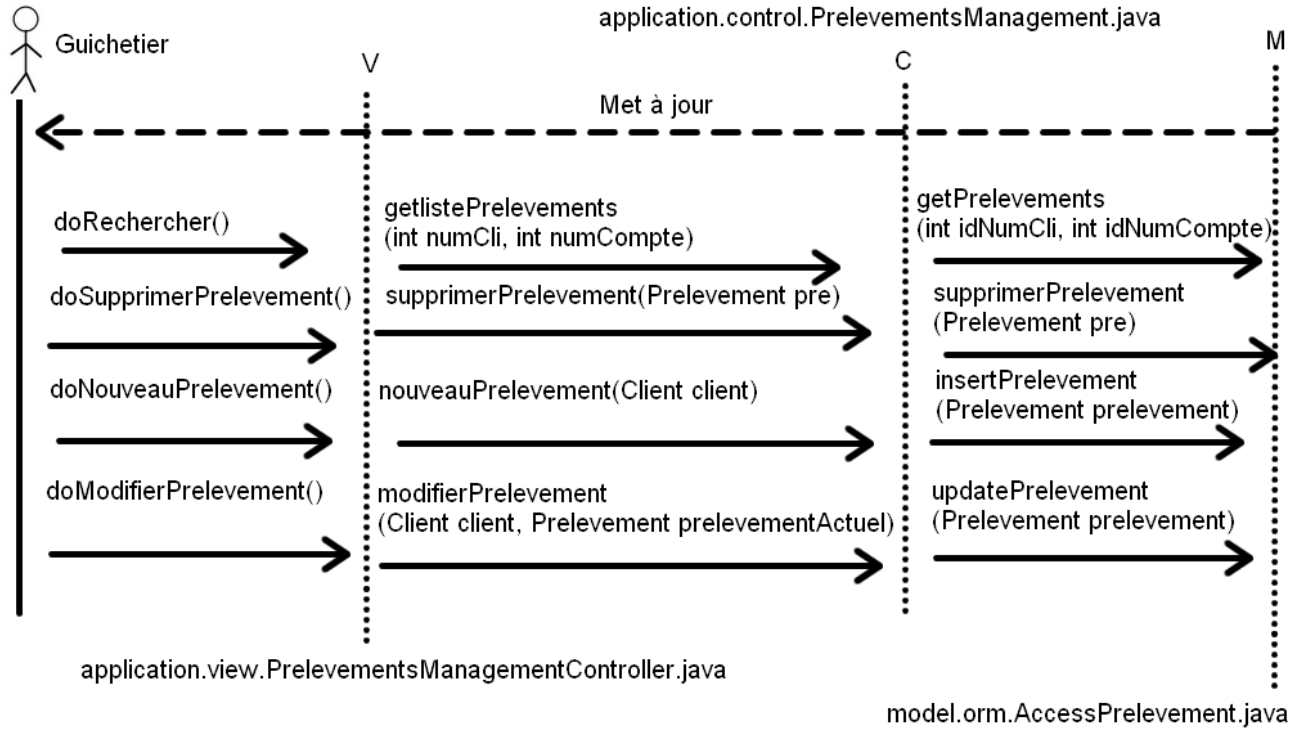
copies écrans cf. Doc Utilisateur V2

Extraits de code significatifs

```java
@FXML
private void doSupprimerPrelevement() {
    int selectedIndice = this.lvPrelevements.getSelectionModel().getSelectedIndex();
    if (selectedIndice >= 0) {
        Prelevement pre = this.olp.get(selectedIndice);
        Alert dialog = new Alert(AlertType.CONFIRMATION);
        dialog.setTitle("Confirmation");
        dialog.setContentText("Voulez-vous vraiment supprimer le prélèvement ?");
        dialog.setHeaderText("Supprimer le prélèvement ?");
        dialog.initOwner(this.primaryStage);
        Optional<ButtonType> reponse = dialog.showAndWait();
        if (reponse.get() == ButtonType.OK) {
            this.pm.supprimerPrelevement(pre);
            doRechercher();
        }
    }
}
```

Cette fonctionnalité permet de supprimer un prélèvement si le prélèvement est sélectionné, affiche une pop-up de confirmation, si la réponse est OK, alors supprime le prélèvement et rafraîchit la liste des prélèvements.

Diagramme de séquence :



Egxon Zejnullahi

V1

5.5 Le CRUD des employés

Le crud des employés s'agit de lire, écrire, modifier et désactiver/activer les employés. Le chef d'agence pourra donc:

- Crée des employés
- Modifier leurs informations
- Les désactiver
- Les activer

Use case :

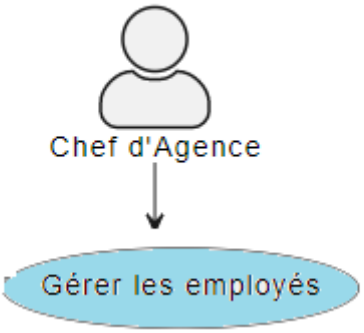
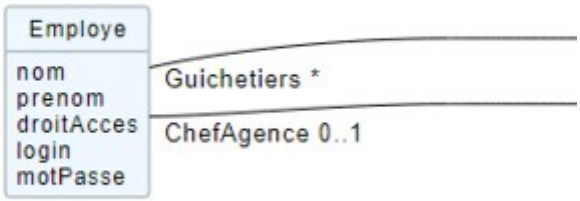


Diagramme de classes données nécessaires :



Classes impliquées dans chaque package :

Le CRUD Employé Package

application.view

- EmployeManagementController.java
- EmployeEditorPaneController.java

Package application.control

- EmployeManagement.java
- EmployeEditor.java

package model.orm

- AccessEmploye.java

package model.date

- Employe.java
- LogToDatabase.jav

copies écrans cf. Doc Utilisateur V2

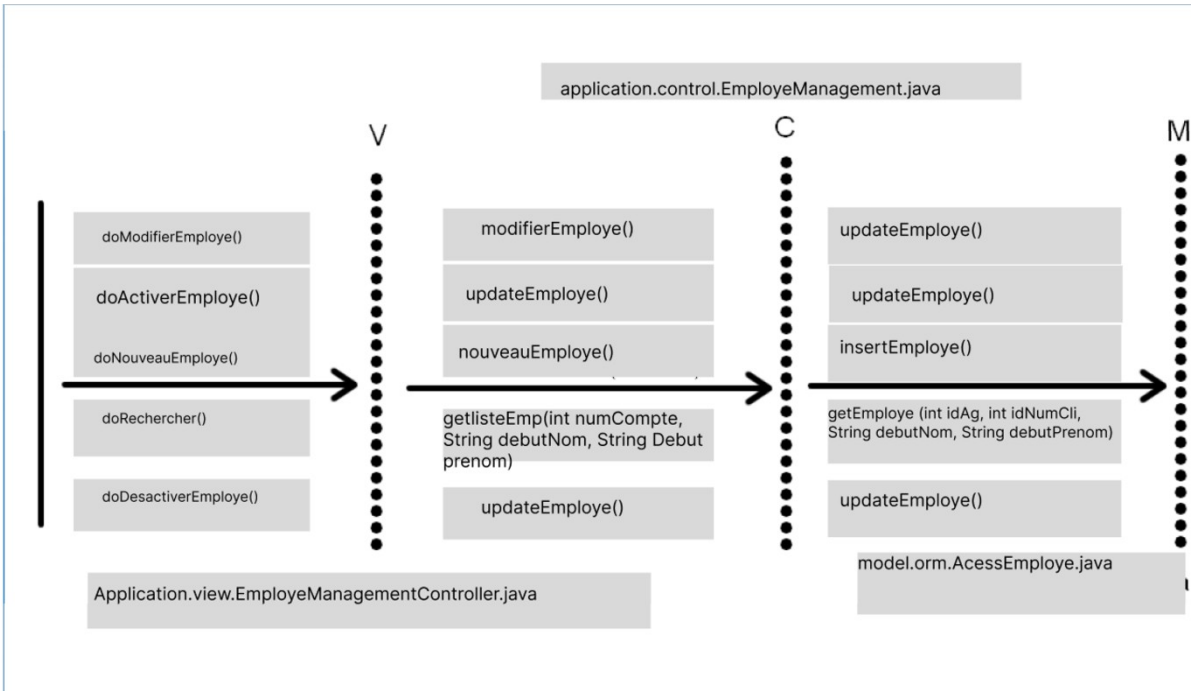
Extraits de code significatifs

```
@FXML
private void doDesactiverEmploye() {
    int selectedIndice = this.lvEmployes.getSelectionModel().getSelectedIndex();
    if (selectedIndice >= 0) {
        Employe empMod = this.ole.get(selectedIndice);
        empMod.motPasse = "/";
        this.em.updateEmploye(empMod);
        this.btnActiverEmp.setDisable(false);
        this.btnDesactEmp.setDisable(true);
    }
}

@FXML
private void doActiverEmploye() {
    int selectedIndice = this.lvEmployes.getSelectionModel().getSelectedIndex();
    if (selectedIndice >= 0) {
        Employe empMod = this.ole.get(selectedIndice);
        empMod.motPasse = "//";
        this.em.updateEmploye(empMod);
        this.btnActiverEmp.setDisable(true);
        this.btnDesactEmp.setDisable(false);
    }
}
```

Pour ces fonctionnalités, on modifie juste le mot de passe de l'employé a un mot de passe de caractère (empMod.length == 1). Puis pour se connecter à l'application, il faut un mot de passe de minimum 2 caractères, de ce fait, il est donc impossible pour l'employé de se connecter. Quand le mot de passe est égal à un caractère, on ne peut pas modifier ces informations mais juste les regarder.

Quand le mot de passe est supérieur à 1 seul caractère, le chef d'agence aura donc la possibilité de modifier les informations du compte et donner un nouveau mot de passe à l'employé (ou alors le mot de passe reste "//" mais c'est c'est dangereux) .

Diagramme de séquence :



5.6 Crée un compte

La création d'un compte permet de crée un compte courant chez un client. Avec ce compte, le client pourra effectuer des débits et des crédits. Tous les comptes ont un découvert autorisé et un solde de départ. Cette fonctionnalité peut être effectué par les guichetiers et les chefs d'agences.

Use case :

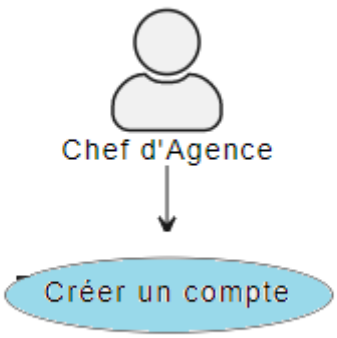
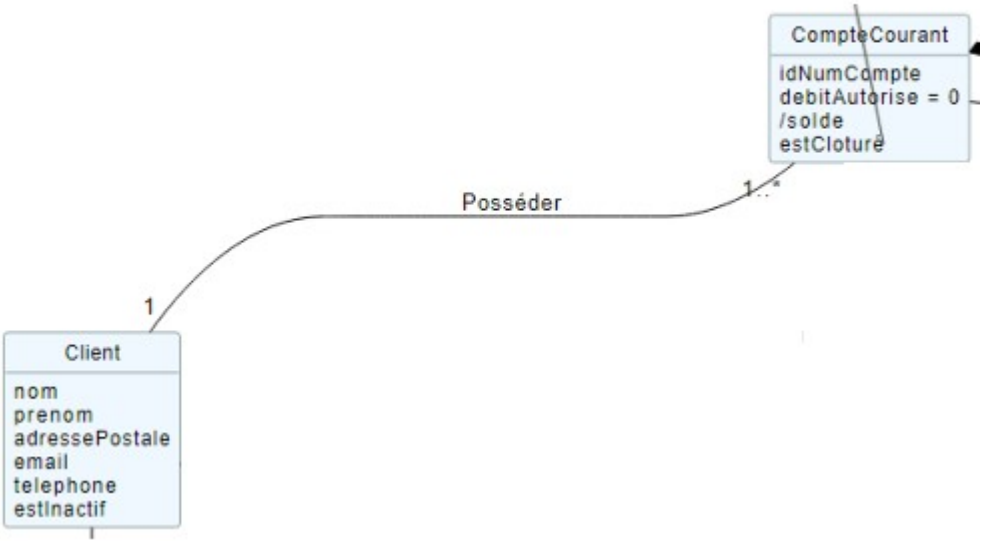


Diagramme de classes données nécessaires :



Crée un compte

Package application.control

- CompteManagement.java
- ClientsEditorPane.java
- ClientsManagement.java
- CompteEditorPane.java

Package application.view

- CompteManagementController.java
- CompteEditorPane.java
- CompteManagementController.java
- CompteEditorPaneController.java

package model.orm

- AccessCompteCourant.java
- LogToDatabase.java
- AccessClients.java

package model.date

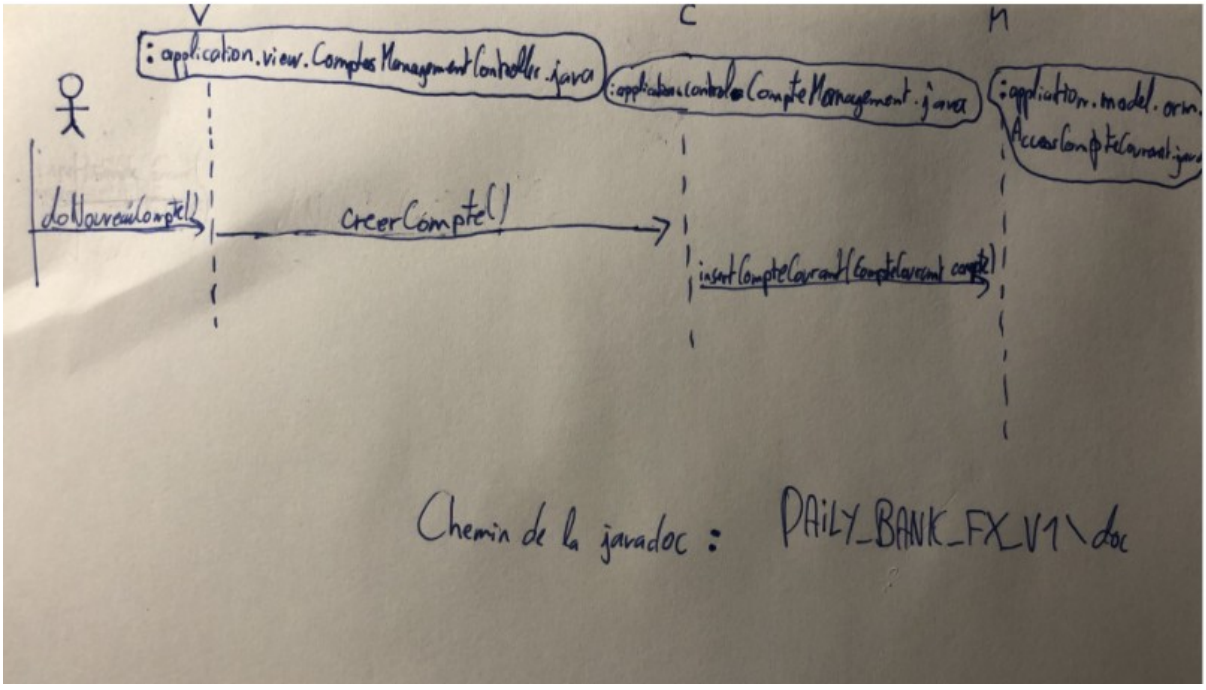
- CompteCourant.java
- Clients.java

copies écrans cf. Doc Utilisateur V2

Extraits de code significatifs

```java
/**
 * Creer un compte
 * @return Le compte crée
 */
public CompteCourant creerCompte() {
    CompteCourant compte;
    CompteEditorPane cep = new CompteEditorPane(this.primaryStage, this.dbs);

    compte = cep.doCompteEditorDialog(this.clientDesComptes, null, EditionMode.CREATION);

    AccessCompteCourant ac = new AccessCompteCourant();

    if (compte != null) {
        try {

            ac.insertCompteCourant(compte);

            if (Math.random() < -1) {
                throw new ApplicationException(Table.CompteCourant, Order.INSERT, "todo : test exceptions", null);
            }
        } catch (DatabaseConnexionException e) {
            ExceptionDialog ed = new ExceptionDialog(this.primaryStage, this.dbs, e);
            ed.doExceptionDialog();
            this.primaryStage.close();
        } catch (ApplicationException ae) {
            ExceptionDialog ed = new ExceptionDialog(this.primaryStage, this.dbs, ae);
            ed.doExceptionDialog();
        }
    }
    return compte;
}
```

Diagramme de séquence :



5.7 Simuler emprunt et simuler assurance

La simulation d'emprunt et la simulation d'asurance permet de faire une simulation et d'emprunt ou une simulation d'assurance.

Use case :

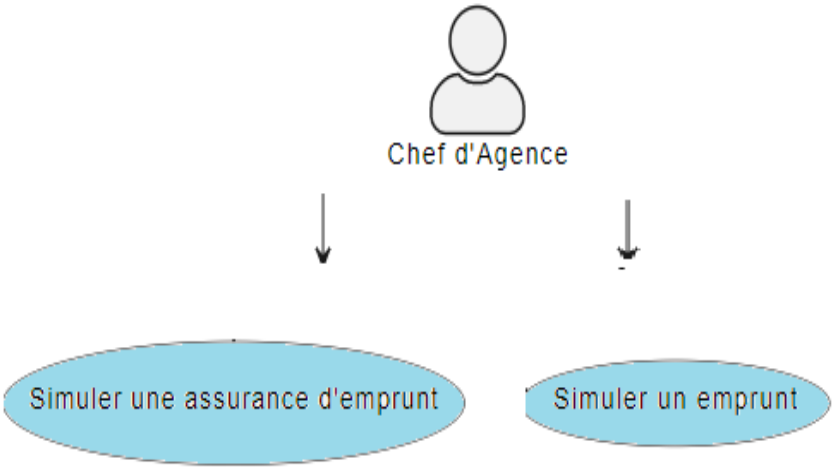
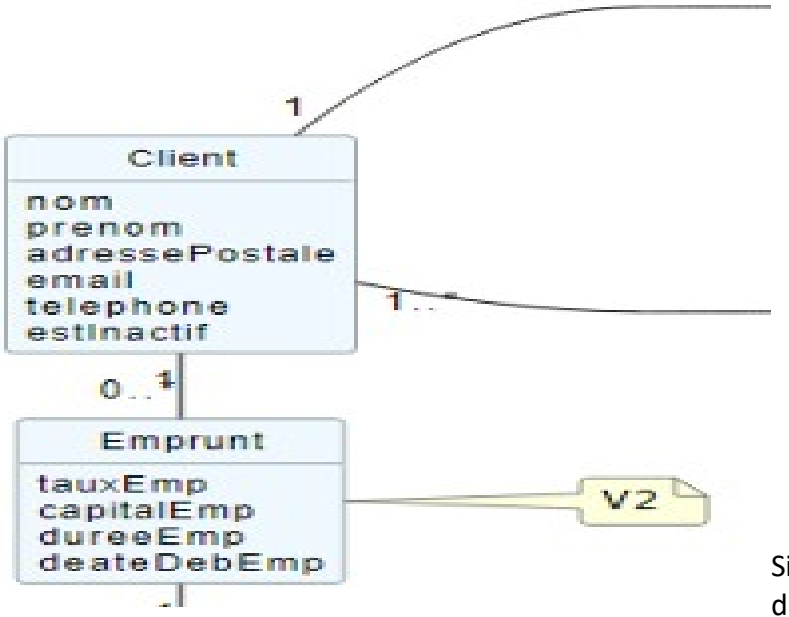


Diagramme de classes données nécessaires :



Simulation d'emprunt et simulation

Package application.view

- SimulerEditorPaneController,java

Package application.control

- SimulerEditorPane.java

copies écrans cf. Doc Utilisateur V2

Extraits de code significatifs

```
private void doSimul() {

    String aff = "";

    if (!montant.getText().isEmpty() && isNumber(this.montant) && !annee.getText().isEmpty() && isNumber(annee) && !TA.getText().isEmpty() && isNumber(TA))

        int numTA= Integer.parseInt(TA.getText());
        int numA= Integer.parseInt(annee.getText());
        int numMontant= Integer.parseInt(montant.getText());

        aff ="Année | Capital restant dû |Intérêts | Amortissement du capital | Annuité\n";

        int Capital=numMontant;
        int interet=(Capital/100)*numTA;
        int amor= (Capital/numA);
        int annuite= amor+interet;


        int totI = 0;
        int totC = 0;
        int totA = 0;
        for (int i =0 ; i<numA; i++) {
            int bona =i+1;
            totI += interet;
            totC += amor;
            totA += annuite;
            aff = aff + "    "+ bona +"            |   " + Capital + "    |     " + interet + "     |              " + amor +"                |   " + an

            Capital = Capital-amor;
            interet = (Capital/100)*numTA;
            annuite = amor+interet;


        }
```

La simulation d'emprunt est une fonction simple juste reprenant les données mit par l'utilisateur et ensuite traité avec des calculs de gestion.

Pour les fonctions de simulations, ils ne faisaient appel à aucune fonction.

Karim Zouli-Barrere
V1

5.8 Créditer/débiter un compte

La fonctionnalité créditer/débiter un compte permet de rajouter ou de retirer de l'argent dans un compte. Lorsque le compte est inactif, nous ne pouvons pas accéder aux opérations.

Use case :

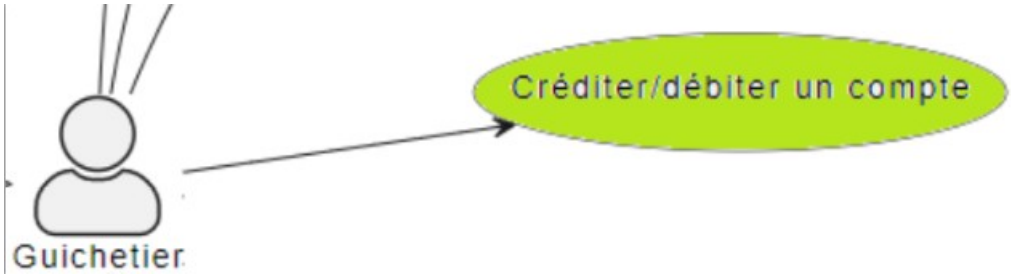
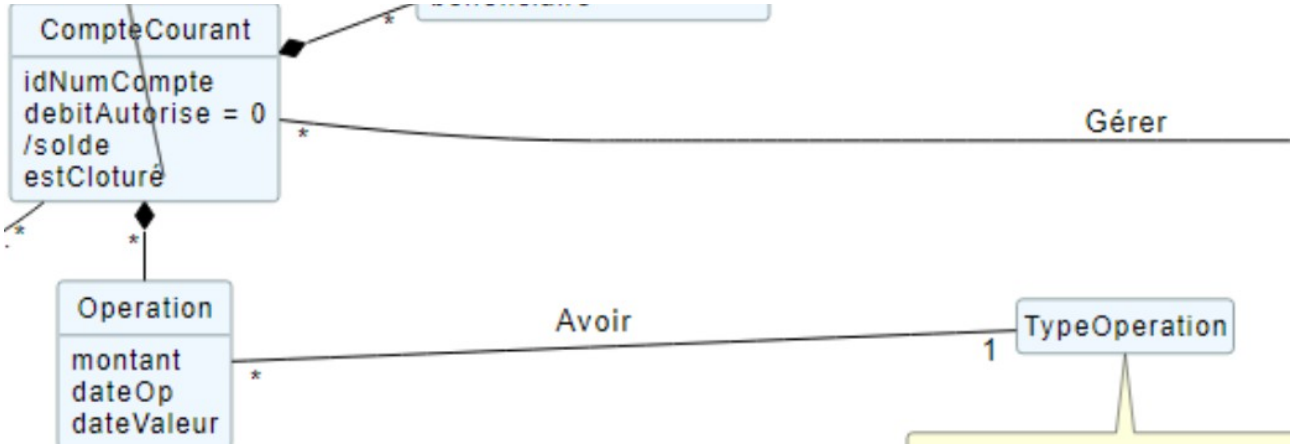


Diagramme de classes données nécessaires :



Classes impliquées dans chaque package :

Package application.control :

- OperationsManagement.java

Package model.orm :

- AccessOperation.java

Package application.view :

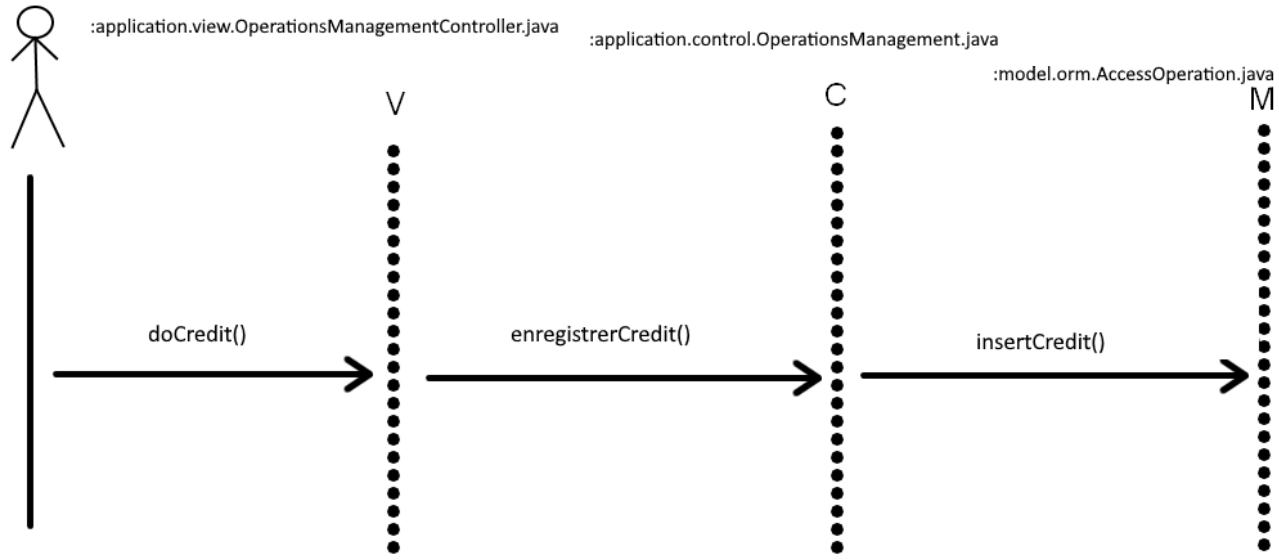
- OperationsManagementController.java

Extraits de code significatifs

```java
/**
 * Retourne le credit enregistré
 * @return l'opération
 */
public Operation enregistrerCredit() {

    OperationEditorPane oep = new OperationEditorPane(this.primaryStage, this.dbs);
    Operation op = oep.doOperationEditorDialog(this.compteConcerne, CategorieOperation.CREDIT);
    if (op != null) {
        try {
            AccessOperation ao = new AccessOperation();

            ao.insertCredit(this.compteConcerne.idNumCompte, op.montant, op.idTypeOp);

        } catch (DatabaseConnexionException e) {
            ExceptionDialog ed = new ExceptionDialog(this.primaryStage, this.dbs, e);
            ed.doExceptionDialog();
            this.primaryStage.close();
            op = null;
        } catch (ApplicationException ae) {
            ExceptionDialog ed = new ExceptionDialog(this.primaryStage, this.dbs, ae);
            ed.doExceptionDialog();
            op = null;
        }
    }
    return op;
}
```

Lorsque l'on clique sur le bouton crédit, cela effectue la fonction doDebit() de la classe OperationsManagementController.java qui appelle ensuite cette fonction ci-dessus qui enregistre un crédit sur un compte précis.

Diagramme de séquence :



V2

5.9 Relever mensuel pdf

La fonctionnalité générer un relevé mensuel d'un compte en PDF permet tel que son nom l'indique, de générer un pdf dans lequel on trouvera le relevé mensuel d'un compte.

Use case :

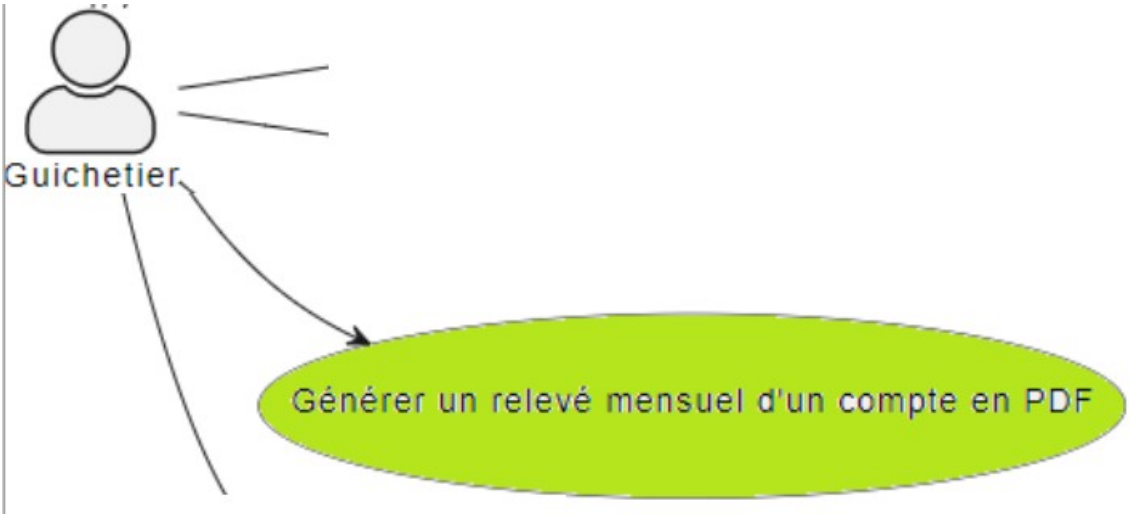
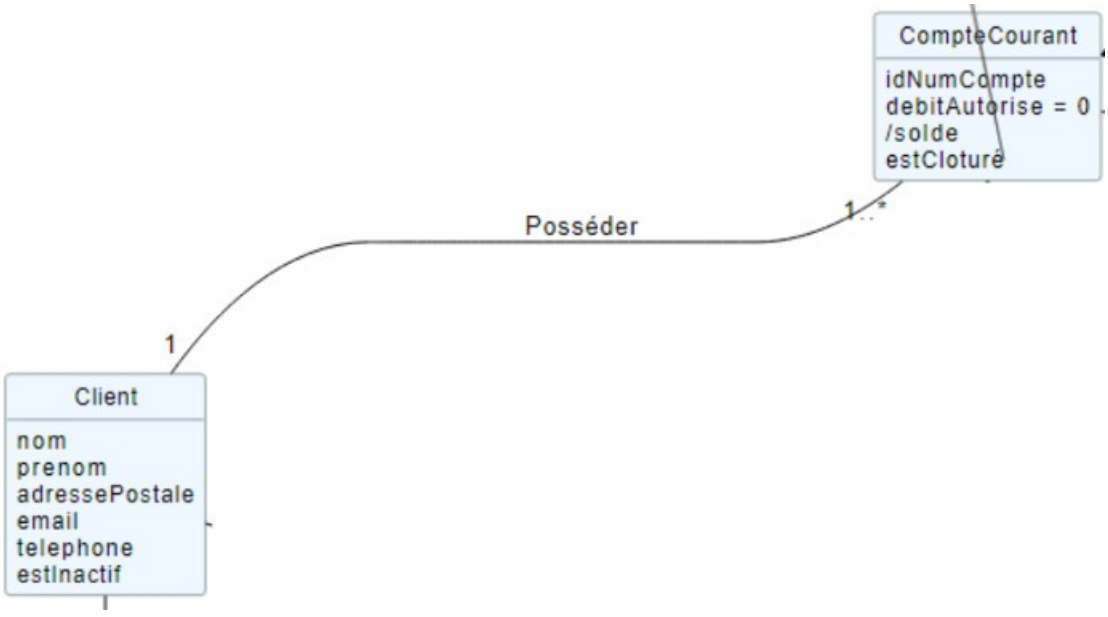


Diagramme de classes données nécessaires :



Classes impliquées dans chaque package :

Package application.view :

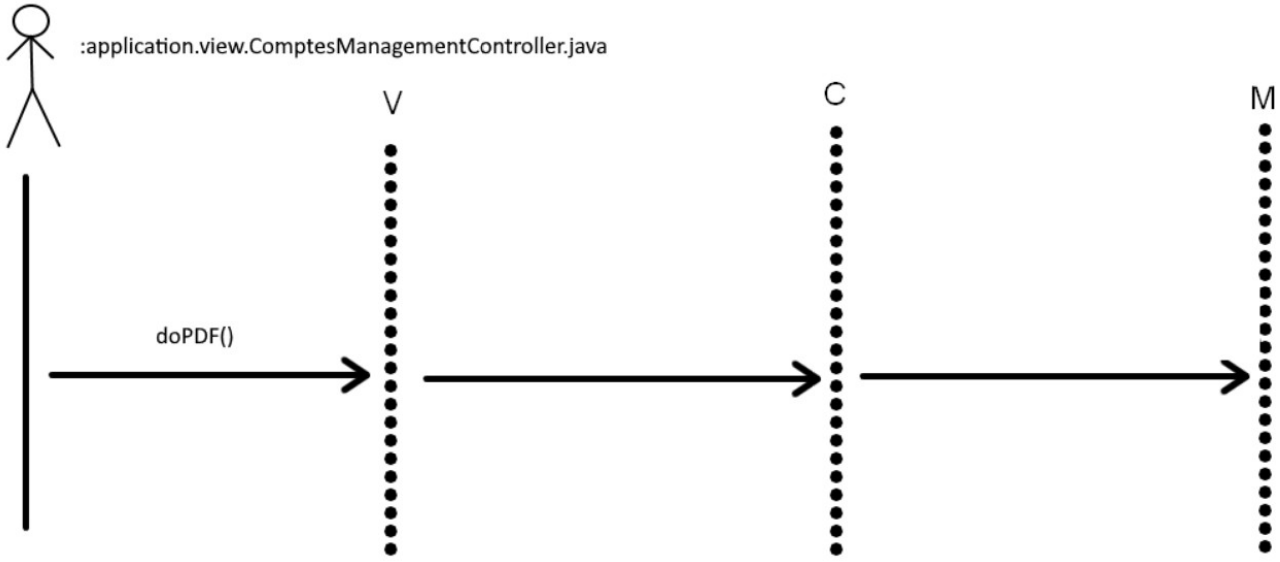
- OperationsManagementController.java

Extraits de code significatifs

```java
@FXML
private void doPDF() {

    Document doc = new Document();

    try {

        PdfWriter.getInstance(doc, new FileOutputStream("releve_mensuel_" + this.clientDesComptes.nom + this.clientDesComptes.prenom + ".pdf"));
        doc.open();

        doc.add(new Paragraph("Relevé mensuel du client " + this.clientDesComptes.nom + " " + this.clientDesComptes.prenom));

        doc.add(new Paragraph("-----------------------------------------------------"));


        for(int i = 0 ; i < this.olCompteCourant.size() ; i++) {
            doc.add(new Paragraph(this.olCompteCourant.get(i).toString()));
            doc.add(new Paragraph(" "));

            OperationsManagement ops = new OperationsManagement(primaryStage, dbs, clientDesComptes, this.olCompteCourant.get(i));

            ArrayList<Operation> listOp = ops.operationsEtSoldeDunCompte().getRight();

            for(int j = 0 ; j < listOp.size() ; j ++) {
                doc.add(new Paragraph(listOp.get(j).toString()));
            }
            doc.add(new Paragraph(" "));
        }
```

Lorsque l'on clique sur le bouton relever mensuel en pdf, cela va créer un fichier pdf avec le nom et prénom du client.
Ensuite on ajoutera le titre du document avec le nom et prénom du client puis les informations de son compte.

Pour cette fonctionnalité on n'a pas utilisé le modèle MVC. Si l'on avait voulu le schématiser par un diagramme de séquence cela aurait donné ceci :



5.10 Débit exceptionnelle

La fonctionnalité débit exceptionnelle permet au chef d'agence de dépasser la limite de débit pour un compte client. Cette fonction ne concerne que le chef d'agence, si un guichetier essaye de faire un débit supérieur au plafond possible, cela ne marchera pas.

Use case :

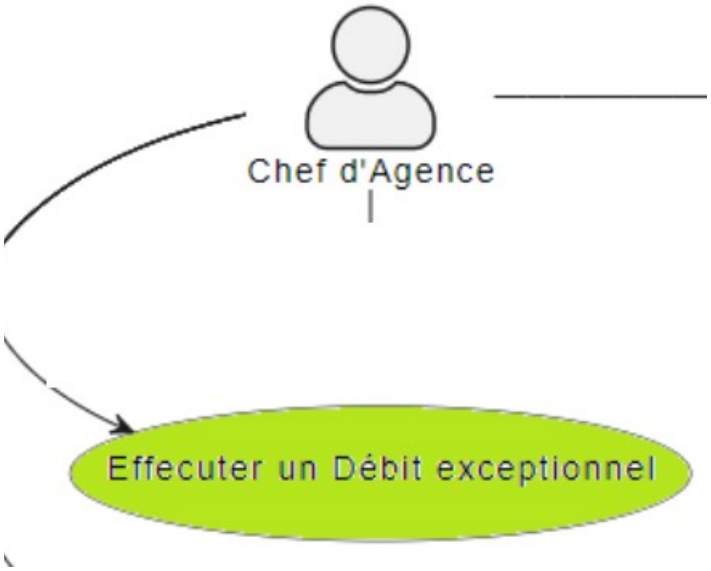
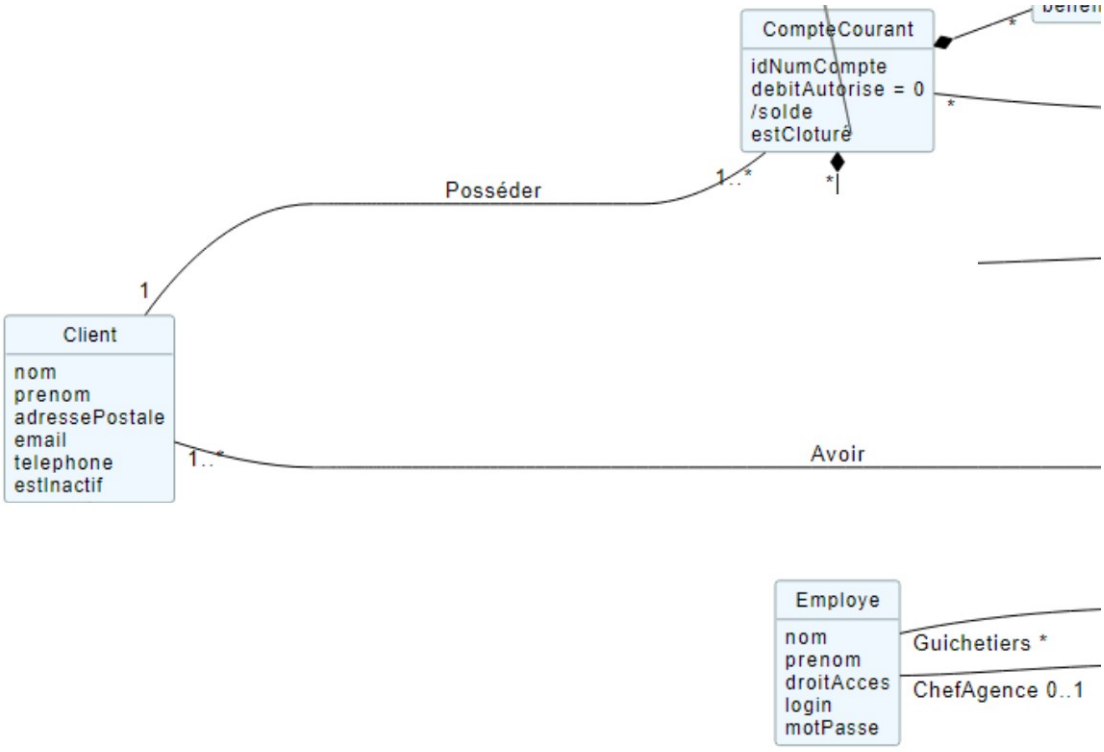


Diagramme de classes données nécessaires :



Classes impliquées dans chaque package :

Package application.control :

- OperationsManagement.java

Package model.orm :
- AccessOperation.java

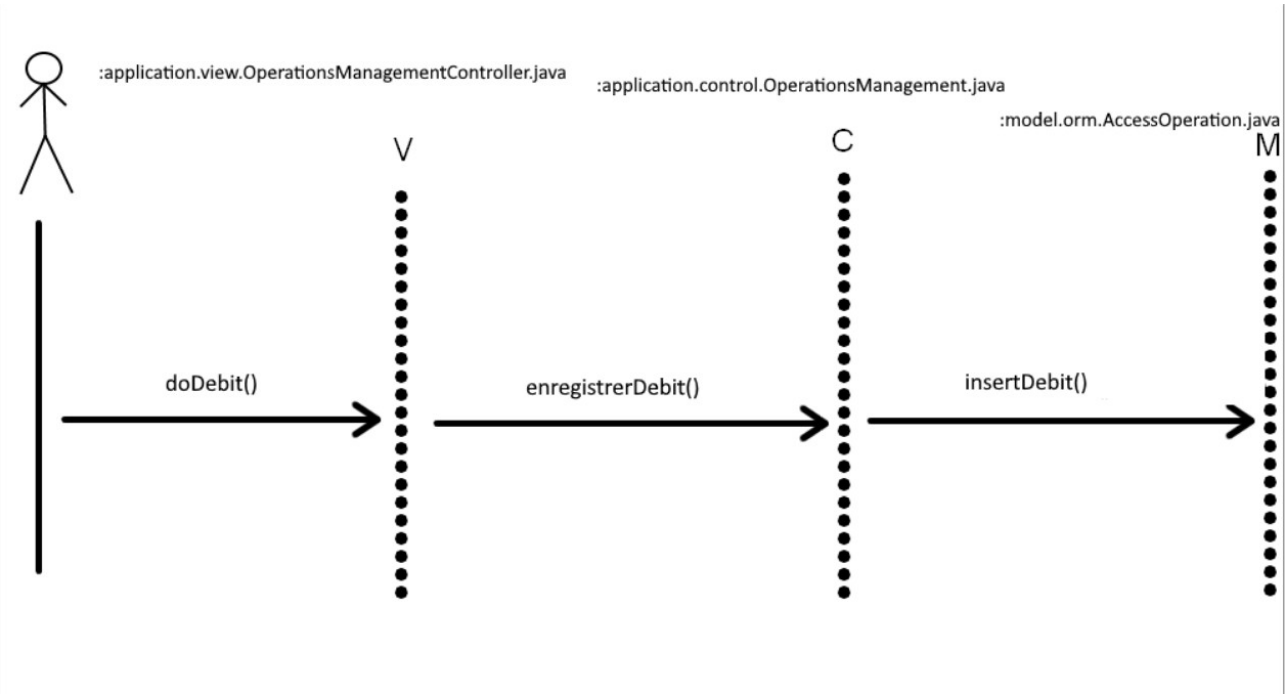
Package application.view :
- OperationsManagementController.java

Extraits de code significatifs

```java
public void insertDebit(int idNumCompte, double montant, String typeOp, DailyBankState dbs)
        throws DatabaseConnexionException, ManagementRuleViolation, DataAccessException {
    try {
        Connection con = LogToDatabase.getConnexion();
        CallableStatement call;

        String q = "{call Debiter (?, ?, ?, ?, ?)}";
        // les ? correspondent aux paramètres : cf. déf procédure (4 paramètres)
        call = con.prepareCall(q);
        // Paramètres in
        call.setInt(1, idNumCompte);
        // 1 -> valeur du premier paramètre, cf. déf procédure
        call.setDouble(2, montant);
        call.setString(3, typeOp);
        // Paramètres out
        call.registerOutParameter(4, java.sql.Types.INTEGER);
        // 4 type du quatrième paramètre qui est déclaré en OUT, cf. déf procédure
        call.setString(5, dbs.getEmpAct().droitsAccess);

        call.execute();

        int res = call.getInt(4);

        if (res != 0 ) { // Erreur applicative
            throw new ManagementRuleViolation(Table.Operation, Order.INSERT,
                    "Erreur de règle de gestion : découvert autorisé dépassé", null);
        }
    } catch (SQLException e) {
        throw new DataAccessException(Table.Operation, Order.INSERT, "Erreur accès", e);
    }
}
```

Lorsque l'on clique sur le bouton débit, cela effectue la fonction doDebit() de la classe OperationsManagementController.java qui appelle ensuite la fonction enregistrerDebit() qui appelle la fonction insertDebit() de la classe OperationsManagement.java. Cette fonction permet d'interagir avec la base de donnée. Un 5ème paramètre a été ajouté afin de permettre d'identifier le type d'utilisateur (Guichetier, Chef D'Agence).

Diagramme de séquence :



6. Installation de l'application

Pour installer l'application, il faudra créer un fichier jar dans lequel on y mettra ojdbc6.jar afin de disposer des pilotes oracle au runtime.

Pour ce faire, cliquer sur Export > Java > Runnable jar files puis ensuite launch configuration et prendre la bonne. Sélectionner "Package required libraries into generated JAR" pour inclure dans le jar le fichier ojdbc6.jar et choisir un fichier de destination .jar

7. Lancer le jar

Pour lancer le jar, vérifiez que vous êtes dans la bonne version de java, c'est-à-dire java 8.

Afin de vérifier ceci, vous pouvez le regarder dans votre invite de commande, en faisant la commande java -version.

Vérifiez d'avoir bien ce résultat :

```
C:\Users\zouli>java -version
java version "1.8.0_261"
Java(TM) SE Runtime Environment (build 1.8.0_261-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.261-b12, mixed mode)
```

Et ensuite double cliquer sur l'application afin de la lancer.