Rapport projet système distribués

Rémy Auloy

Sujet 7 : Gestion de comptes bancaire

Année 2022 - 2023

**TABLE DES MATIERES**

[I. Introduction 2](#_Toc122294965)

[1.1 Présentation 2](#_Toc122294966)

[1.2 Objectifs du projet et justification de l’utilisation d’akka et de java 3](#_Toc122294967)

[II. Analyse du sujet 4](#_Toc122294968)

[1.1 Structure de données utilisées 4](#_Toc122294969)

[a. Couche métier 4](#_Toc122294970)

[b. Couche base de données 6](#_Toc122294971)

[1.2 Algorithmes utilisés 7](#_Toc122294972)

[1.3 Paradigme retenu et justification 12](#_Toc122294973)

[1.4 Décomposition du projet en sous-problème 14](#_Toc122294974)

[1.5 Problèmes rencontrés durant le projet 14](#_Toc122294975)

[III. Architecture logicielle détaillée 16](#_Toc122294976)

[IV. SpÉcification des classes 21](#_Toc122294977)

[V. documentation de compilation 21](#_Toc122294978)

[1.1 Prérequis 21](#_Toc122294979)

[1.2 Comment créer la base de données 21](#_Toc122294980)

[1.3 Comment compiler le projet 23](#_Toc122294981)

[VI. jeu de tests 23](#_Toc122294982)

[1.1 Les donnée de ma base 23](#_Toc122294983)

[VII. conclusion 27](#_Toc122294984)

**TABLE DES ILLUSTRATIONS**

[1 : Diagramme des structures de données 4](#_Toc122294985)

[2 : Diagramme de la base de données 6](#_Toc122294986)

[3 : Algorithme de recherche du banquier du client 7](file:///C:\Users\Tarja\Desktop\gestionCompteBancaire\Docs\Auloy_Rapport.docx#_Toc122294987)

[4 : Algorithme de vérification de la demande du client 9](file:///C:\Users\Tarja\Desktop\gestionCompteBancaire\Docs\Auloy_Rapport.docx#_Toc122294988)

[5 : L’algorithme de création d’un jeu de test aléatoire 11](file:///C:\Users\Tarja\Desktop\gestionCompteBancaire\Docs\Auloy_Rapport.docx#_Toc122294989)

[6 : Liste des packages avec leurs contenus 16](file:///C:\Users\Tarja\Desktop\gestionCompteBancaire\Docs\Auloy_Rapport.docx#_Toc122294990)

[7 : Diagramme de classe avec dépendances 18](file:///C:\Users\Tarja\Desktop\gestionCompteBancaire\Docs\Auloy_Rapport.docx#_Toc122294991)

[8 : Diagramme de classe avec dépendances 18](#_Toc122294992)

[9 : exemple de d'instanciation 19](#_Toc122294993)

[10 : Exemple de composition 19](#_Toc122294994)

[11 : creation de la base de données réussi 22](#_Toc122294995)

[12 : Capture d'écran du succès de la compilation 23](#_Toc122294996)

[13 : clients 24](file:///C:\Users\Tarja\Desktop\gestionCompteBancaire\Docs\Auloy_Rapport.docx#_Toc122294997)

[14 : comptes 24](#_Toc122294998)

[15 : banquiers 24](#_Toc122294999)

# Introduction

## Présentation

Pour ce projet, j’ai choisi de traiter le sujet 7, intitulé, « Gestion de comptes bancaires ». Ce sujet, doit permettre de mettre en place une simulation de gestion de comptes bancaires à l’aide de la technologie « Akka ». Nous devrons également mettre en place un système de gestion de base de données afin d’obtenir une persistance dans les données des clients.

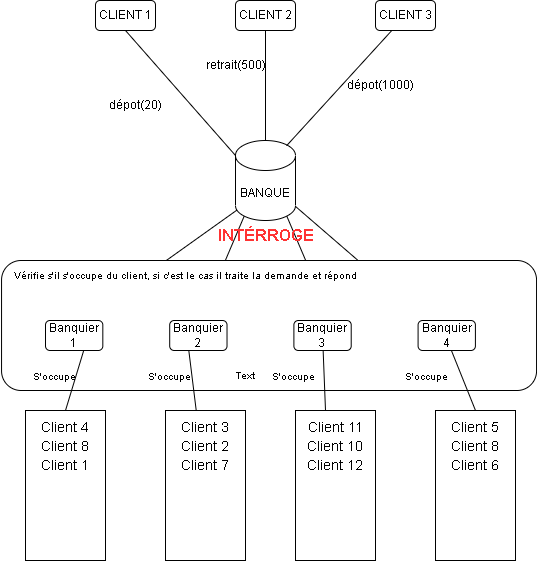
Pour informations : « Akka » est une bibliothèque de programmation concurrente pour Java qui utilise le modèle d'acteurs pour faciliter le développement d'applications distribuées. Le modèle d'acteurs est un modèle de programmation concurrente qui se concentre sur la communication entre objets logiciels plutôt que sur l'héritage. Akka permet également de gérer la concurrence et les transactions de manière transparente et offre une tolérance aux pannes et une résilience avancée pour les applications critiques qui doivent être disponibles en permanence. En résumé, Akka est une bibliothèque qui simplifie le développement d'applications distribuées en utilisant un modèle d'acteurs et offre une gestion avancée de la concurrence, des transactions et de la tolérance aux pannes.

Contextualisons la situation, nous avons une banque qui emploi des banquiers. Ces banquiers, eux, ont la charge de plusieurs compte client.

À présent, un client se présente à la banque pour déposer ou retirer de l’argent sur un de ces comptes. Il envoi donc une demande à la banque, la banque, elle, avant de demander au banquier, va chercher quelle est le banquier qui s’occupe du compte du client qui fait la demande. Une fois le banquier trouvé, la banque va interroger le banquier afin de savoir si la demande du client est possible.

Rajoutons, qu’il est totalement possible d’avoir plusieurs clients simultanément. Tout comme il peut y avoir plusieurs banquiers employés. En revanche, nous avons une seule banque, ce qui portera problème lorsque nous parlerons des problèmes éventuels.   
D’autres problèmes potentiels peuvent apparaitre mais comme je l’ai dit nous en reparlerons plus tard.

Afin de mieux visualiser ceci, voici un schéma :



## Objectifs du projet et justification de l’utilisation d’akka et de java

Pour réaliser ce projet, j'ai choisi d'utiliser la bibliothèque Akka pour gérer les acteurs et le langage de programmation Java. Les objectifs que je me suis fixés en réalisant ce projet étaient de mettre en place une simulation de gestionnaire de compte bancaire afin de comprendre comment fonctionne un tel système et d’essayer de me familiariser avec les concepts de concurrence et de gestion des acteurs.

J'ai choisi d'utiliser Akka car c'est une bibliothèque de gestion des acteurs très complète qui offre de nombreux outils pour gérer la concurrence et la distribution des tâches. En utilisant Akka, j'ai pu facilement mettre en place les différents acteurs du système et gérer leurs interactions de manière efficace. D’autant plus que nous avions découvert Akka avec java en TP

J'ai également choisi d'utiliser Java car c'est un langage de programmation populaire et largement utilisé dans l'industrie. En utilisant Java, j'ai pu bénéficier de la prise en charge de ce langage par de nombreux environnements de développement et me familiariser avec les bonnes pratiques de développement. De plus, Java offre une grande flexibilité et une grande richesse de bibliothèques pour réaliser des projets de ce type. J’ajouterais que je préfère Java aux deux autres langages possiblement choisissable C/C++ et Scala. Le C/C++ est un langage avec lequel j’ai du mal et ce qui concerne Scala, c’est un langage très puissant également et qui aurait pu parfaitement convenir a ce projet, en revanche je n’ai pas assez de connaissance dans ce langage pour me permettre de le choisir.

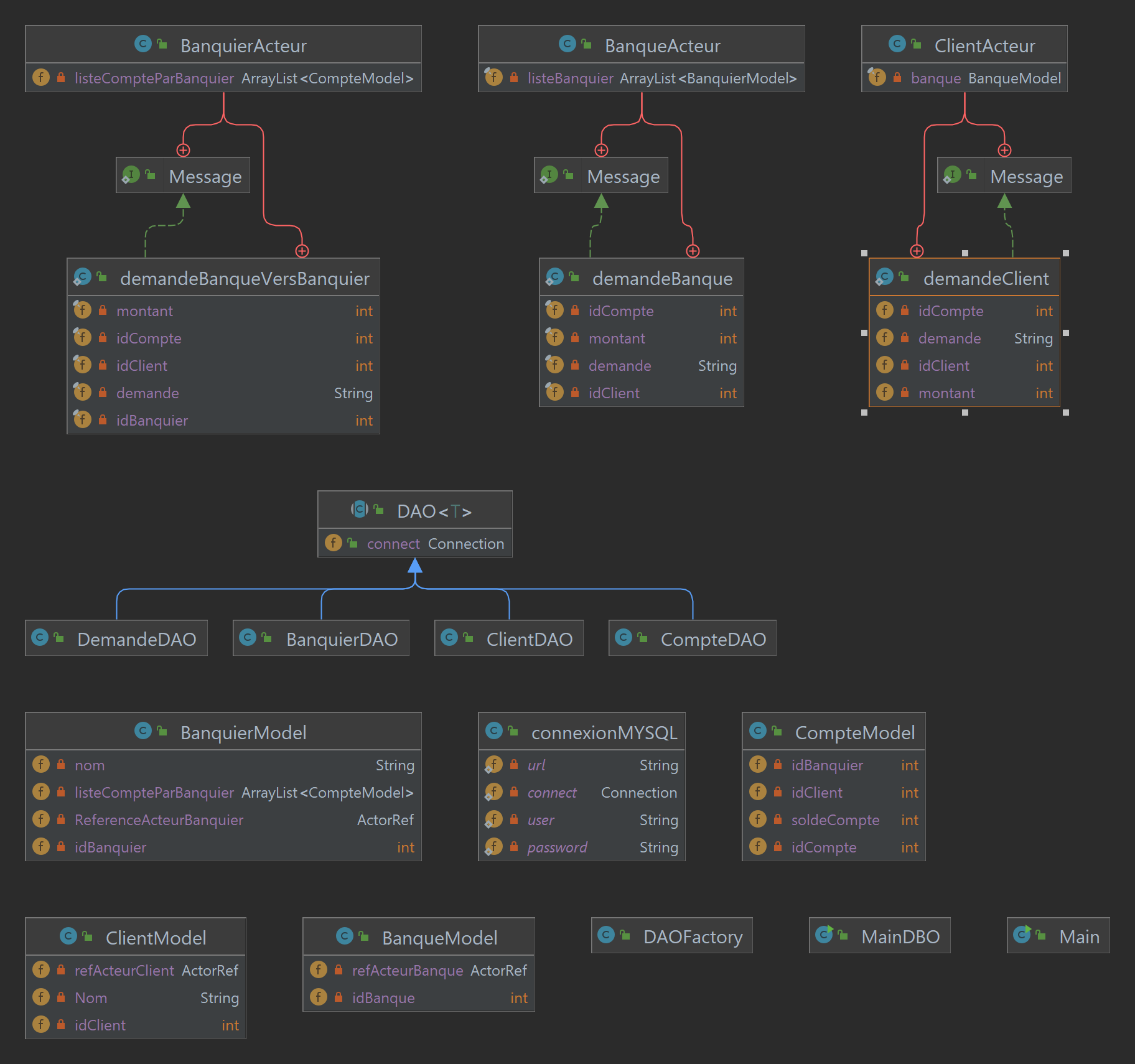
De plus, j’aimerais préciser que je n’ai pas développer d’interface graphique a mon application, tout se passe dans le terminal. Mais nous reviendrons sur cela plus tard.

# Analyse du sujet

## Structure de données utilisées

### Couche métier

Pour vous aider à visualiser l’ensemble de mes structures de données, voici un diagramme des classes sans les dépendances, n’y les méthodes. Nous verrons le diagramme de classe plus complet, lorsque nous parlerons de l’architecture du projet.



1 : Diagramme des structures de données

Concernant les structures de données que j’ai utilisées dans mon projet pour stocker et manipuler mes données de la couche métier.

Pour tout ce qui concerne les « id », du solde du compte et des montants des demandes, j’ai décidé de plutôt choisir des « int » plutôt que d’autre type comme « long ».

Il faut savoir que

* « Int » permet de stocker des valeurs entières comprises entre -2147483648 et 2147483647.
* « Long » permet de stocker des valeurs entières comprises entre -9223372036854775808 et 9223372036854775807.

J’ai fait ce choix car dans notre cas il s’agit d’un projet universitaire, et non d’une réelle banque. En sachant que « long » est plus gourmand en mémoire que « int », car il occupe 64 bits dans la mémoire de l'ordinateur, contre 32 bits pour « int », j’ai décidé de choisir int pour gagner en performance.  
Si le projet aurait été utilisé par une vraie banque, dans ce cas. Il faudrait utiliser « long » car les identifiants peuvent être très large et dépasser la plage de valeurs autorisée par « int ». De même que pour le solde du compte.

Concernant, mes chaines de caractères, j’ai utilisé des « String » car c’est une classe de la bibliothèque standard de Java qui permet de manipuler des chaînes de caractères. Elle est très pratique et facile à utiliser, elle offre de nombreuses méthodes pratiques pour manipuler les chaînes de caractères, comme la possibilité de concaténer des chaînes, de remplacer des sous-chaînes, de transformer la casse des caractères, etc. D’où mon choix.

Pour ce qui est du stockage de la liste des comptes, ou encore de la liste des banquiers. En d’autres termes, dès qu’il faut stocker une liste de quelque chose, j’ai utilisé la structure de donné « ArrayList » pour la raison suivante :

* « ArrayList » est une structure de données très flexible et facile à utiliser. Elle permet de stocker des données de manière dynamique et de les manipuler facilement grâce aux méthodes de la classe « ArrayList ».
* C’est une structure très performante pour les opérations courantes, comme l’ajout ou la suppression d’éléments dans la liste (ici, nos comptes ou nos banquiers). Le parcourt des listes est également rapide avec une « ArrayList »
* De plus, « ArrayList » est une structure de donnée dites adaptative, ce qui en d’autres termes signifie qu’elle s’agrandit et se rétrécit automatique en fonction du nombre d’éléments qu’elle contient. Cela peut être un avantage par rapport aux tableaux, qui ont une taille fixe et nécessitent souvent d'être redimensionnés manuellement.

Comme je voulais manipuler mes données facilement et les stocker dynamiquement mon choix c’est porter sur les « ArrayList » plutôt que d’autres structure comme les listes chainées.

Dans le code métier, vous ne trouverez pas d’autres structure de données si ce n’est des structures de type « ActorRef » pour stocker des références aux acteurs dans les classes métiers. Ou encore une structure « Connection » dans la classe « connexionMYSQl » mais ce choix n’a pas besoin d’être justifié car il est indispensable afin d’assurer la connexion à la base de données.

### Couche base de données

Pour vous aider à visualiser mes structures de données sur la couche de la base de données, voici le diagramme de mon modèle conceptuel de données (MCD).

Une image contenant texte

Description générée automatiquement

2 : Diagramme de la base de données

Comme nous décidons d’avoir uniquement une seule banque, nous n’avons pas besoin de modéliser la table « Banque », si nous le faisions elle serait inutile.

En revanche j’ai choisi de stocker les demandes des clients faites dans la classe ClientActeur, cela correspond à son inner classe nommé « demandeClient ». Cela peut être utile afin d’avoir un historique des demandes, si un client veut connaitre ses demandes, la banque est mesure de lui demander, d’où mon choix de vouloir les stocker.

Comme vous pouvez le remarquer, pour choisir mes structures de données ici, je me suis aligné sur les structures de la couche métier, les « int » restent des « int ». En ce qui concerne les chaines de caractères, j’a choisis d’utiliser des char (50) pour stocker les noms des clients. Vous verrez plus tard que dans mes tests, je n’affiche jamais les noms dans la console mais plutôt les noms, en revanche j’ai trouvé plus pratique de les stocker en base de données afin de visualiser plus facilement à qui correspond ces id.

Pour finir, on remarque que le type de la demande est une « énum », pourquoi ? Car un client à seulement 2 possibilités de demande. Soit il fait un dépôt d’argent sur son compte, soit il fait un retrait d’argent de son compte. D’où la structure « ENUM ».

Je détaillerais les cardinalités et donc la justification des clés étrangère dans la partie architecture du rapport.

## Algorithmes utilisés

Dans mon projet j’ai plusieurs algorithmes que j’ai codé :

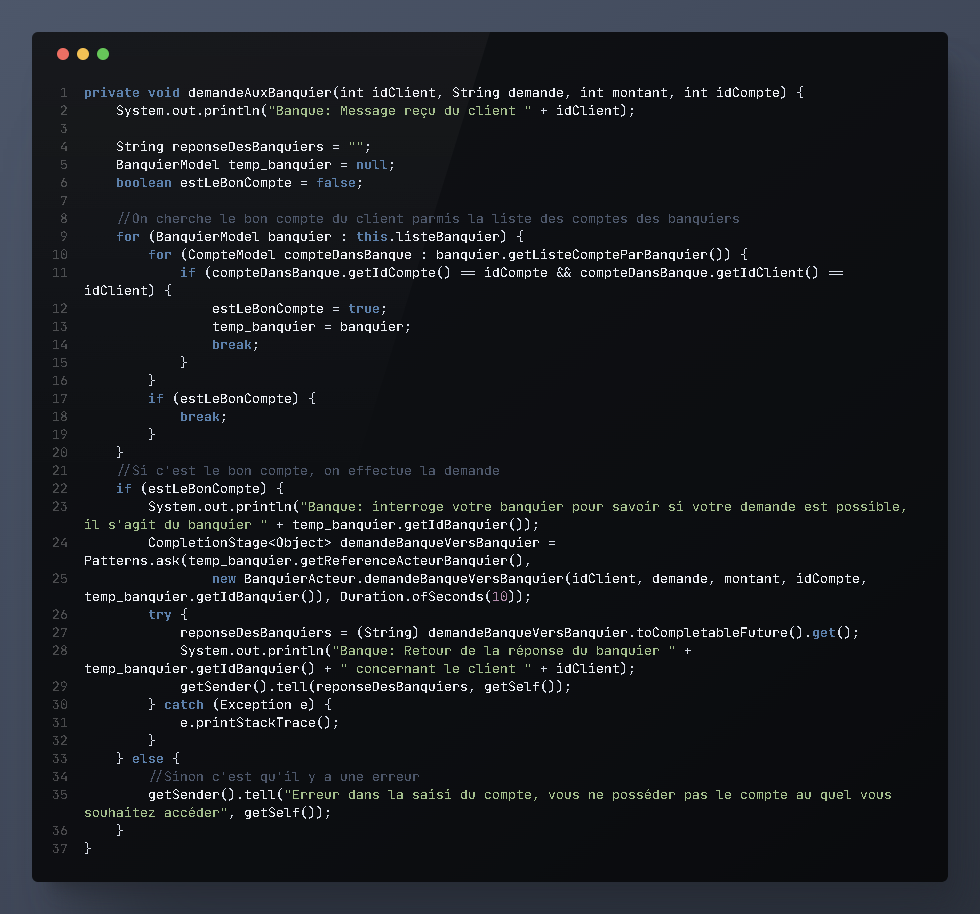
* L’algorithme de recherche du banquier du client (classe « BanquierActeur »)
* L’algorithme de vérification de la demande du client par le banquier (classe « BanquierActeur »)
* L’algorithme de création d’un jeu de test aléatoire (classe « MainDBO »)

Aucun n’algorithme a proprement parlé dans la classe du client, car celui-ci fait simplement une demande (« ask ») a la banque.

Détaillons ces algorithmes :

Algorithme de recherche du banquier du client faisant la demande :

Si vous souhaitez voir le code, il est disponible ici :

« gestionCompteBancaire\src\main\java\org\example\actors\BanqueActor.java »

3 : Algorithme de recherche du banquier du client

Expliquons cet algorithme, que fais ce code ?

La première chose que fait le code est d'afficher un message indiquant qu'il a reçu un message du client avec l'identifiant donné. Ensuite, il déclare une variable de type "String" appelée "reponseDesBanquiers" qui sera utilisée plus tard pour stocker la réponse des banquiers à la demande du client. Il déclare également une variable de type "BanquierModel" appelée "temp\_banquier" qui sera utilisée pour stocker un objet "BanquierModel" temporairement pendant l'exécution du code. Enfin, il déclare une variable booléenne appelée "estLeBonCompte" qui sera utilisée pour indiquer si le compte indiqué par le client est valide ou non.

Ensuite, le code commence une boucle "for" qui parcoure la liste des banquiers. Pour chaque banquier de la liste, le code démarre une deuxième boucle "for" qui parcoure la liste des comptes de ce banquier. Pour chaque compte de la liste, le code vérifie si l'identifiant du compte correspond à celui donné par le client et si l'identifiant du client correspond à celui associé au compte. Si ces deux conditions sont vraies, cela signifie que le compte appartient bien au client et le code met à jour la variable "estLeBonCompte" en lui donnant la valeur "true" et stocke l'objet "banquier" dans la variable "temp\_banquier". Si ces conditions ne sont pas vraies, le code passe au compte suivant de la liste. Si la boucle "for" parcourant les comptes du banquier en cours se termine sans avoir trouvé le compte correspondant, elle passe au banquier suivant de la liste. Si aucun banquier ne possède le compte correspondant, la variable "estLeBonCompte" reste à sa valeur initiale de "false".

Une fois la boucle "for" terminée, le code vérifie la valeur de la variable "estLeBonCompte". Si elle est "true", cela signifie que le compte correspondant a été trouvé et que la demande du client peut être traitée. Le code affiche alors un message indiquant qu'il interroge le banquier responsable du compte du client et envoie un message au banquier en question en utilisant la méthode "ask" de la classe "Patterns". Cette méthode permet d’envoyer un message à un acteur en spécifiant la référence de l'acteur cible, le message à envoyer et une durée maximale d'attente de la réponse. Dans ce cas, le message envoyé est une instance de la classe "BanquierActeur.demandeBanqueVersBanquier", qui contient l'identifiant du client, le type de demande, le montant, l'identifiant de compte et l'identifiant du banquier.

Le code utilise également la méthode "toCompletableFuture" sur le résultat de l'appel à "ask" pour obtenir un objet "CompletableFuture" qui permettra de récupérer la réponse du banquier de manière asynchrone. Ensuite, il utilise la méthode "get" de l'objet "CompletableFuture" pour attendre la réponse du banquier et la stocker dans la variable "reponseDesBanquiers". Si une exception est levée pendant l'exécution de ce code, elle est affichée à l'aide de la méthode "printStackTrace" de la classe "Exception".

Une fois la réponse du banquier reçue, le code affiche un message indiquant qu'il a reçu la réponse du banquier et envoie la réponse au client en utilisant la méthode "tell" de l'objet "ActorRef" représentant le client. Si, au contraire, la variable "estLeBonCompte" est "false", cela signifie que le compte indiqué par le client n'a pas été trouvé et le code envoie un message au client indiquant qu'il y a une erreur dans la saisie du compte.

Algorithme de vérification de la demande du client par le banquier :

Si vous souhaitez voir le code, il est disponible ici :

« gestionCompteBancaire\src\main\java\org\example\actors\BanquierActor.java »

4 : Algorithme de vérification de la demande du client

Expliquons cet algorithme, que fais ce code ?

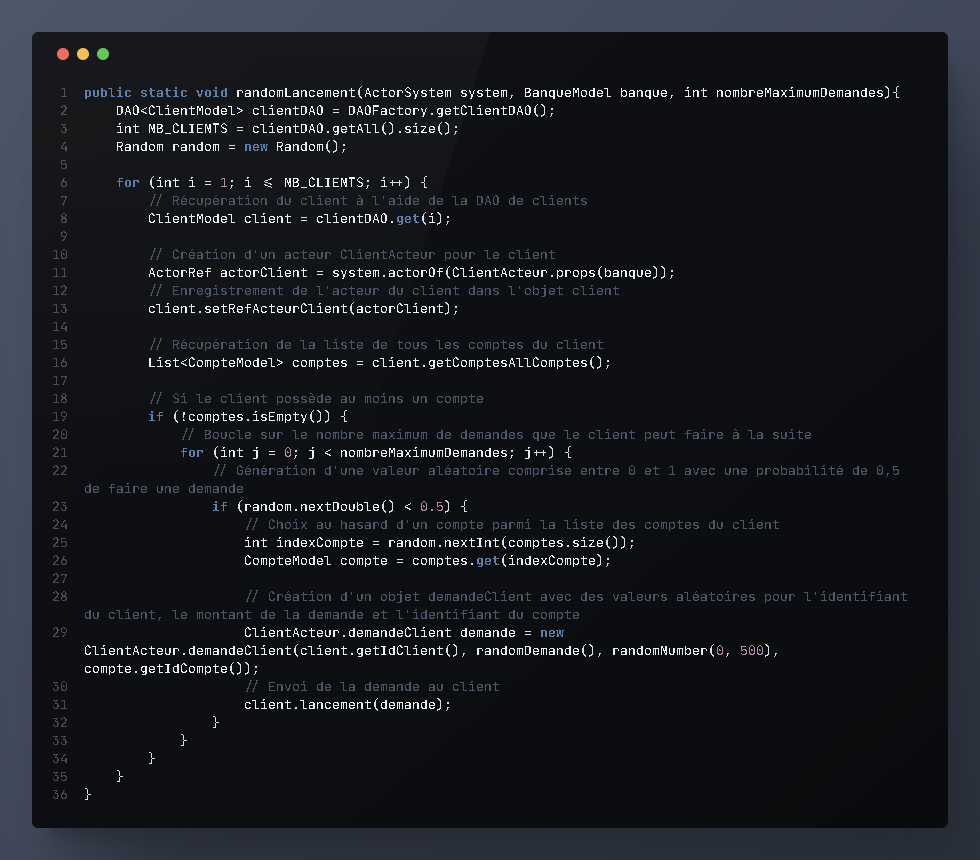
La première chose que fait le code est d'afficher un message indiquant qu'il a reçu une demande du client avec l'identifiant donné. Ensuite, il déclare des variables de type "DAO" pour accéder à la base de données et une variable de type "ClientActeur.demandeClient" qui sera utilisée pour enregistrer la demande du client dans la base de données. Il déclare également une variable de type "CompteModel" appelée "compte" qui sera utilisée pour stocker un objet "CompteModel" correspondant au compte du client.

Le code commence ensuite une boucle "for" qui parcoure la liste des comptes du banquier. Pour chaque compte de la liste, le code vérifie si l'identifiant du compte correspond à celui donné par le client et si l'identifiant du client correspond à celui associé au compte. Si ces deux conditions sont vraies, cela signifie que le compte appartient bien au client et le code stocke l'objet "compte" dans la variable "compte" et sort de la boucle. Si ces conditions ne sont pas vraies, le code passe au compte suivant de la liste.

Une fois la boucle terminée, le code vérifie la valeur de la variable "compte". Si elle est non-nulle, cela signifie que le compte correspondant a été trouvé et que la demande du client peut être traitée. Le code vérifie alors le type de demande en utilisant l'instruction "if". Si la demande est un "retrait", le code vérifie si le solde du compte est suffisant pour effectuer la transaction. Si le solde est insuffisant, le code envoie un message d'erreur au client en utilisant la méthode "tell" de l'objet "ActorRef" représentant le client. Si le solde est suffisant, le code effectue la transaction en mettant à jour le solde du compte, en enregistrant la demande du client dans la base de données et en envoyant un message de confirmation au client. Si la demande est un "dépôt", le code effectue la transaction de manière similaire. Si la variable "compte" est nulle, cela signifie que le compte indiqué par le client n'a pas été trouvé et le code envoie un message d'erreur au client.

L’algorithme de création d’un jeu de test aléatoire :

Si vous souhaitez voir le code, il est disponible ici :

« gestionCompteBancaire\src\main\java\org\example\MainDBO .java»

5 : L’algorithme de création d’un jeu de test aléatoire

Expliquons ce dernier algorithme ensemble, que fait-il ?

Tout d'abord, la méthode crée un objet "DAO" pour accéder aux clients dans la base de données en utilisant la classe "DAOFactory". Elle récupère également la liste de tous les clients en utilisant la méthode "getAll" de cet objet "DAO".

Ensuite, la méthode crée un objet "Random" qui va être utilisé pour générer des valeurs aléatoires.

Pour chaque client de la banque, la méthode crée un acteur "ClientActeur" en utilisant la méthode "actorOf" de l'objet "ActorSystem" passé en paramètre. Elle enregistre également l'acteur du client dans l'objet "ClientModel" correspondant en utilisant la méthode "setRefActeurClient".

La méthode récupère également la liste de tous les comptes du client en utilisant la méthode "getComptesAllComptes" de l'objet "ClientModel". Si le client possède au moins un compte, la méthode entre dans une boucle qui va générer jusqu'à "nombreMaximumDemandes" demandes pour le client.

A chaque itération de la boucle, la méthode génère une valeur aléatoire avec une probabilité de 1/2 de faire une demande. Si une demande doit être faite, la méthode choisit au hasard un compte parmi la liste des comptes du client en utilisant l'objet "Random" et crée un objet "ClientActeur.demandeClient" avec des valeurs aléatoires pour l'identifiant du client, le montant de la demande et l'identifiant du compte. La demande est ensuite envoyée au client en utilisant la méthode "lancement" de l'objet "ClientModel" correspondant.

On peut donc créer un grand nombre de demande et simuler des probabilités que les clients fassent oui ou non une demande.

## Paradigme retenu et justification

Dans mon cas, en choisissant le projet sur « Akka », le choix du paradigme n’est pas très compliqué à choisir.

Pour réaliser ce projet de gestion de comptes bancaires, j'ai choisi de m'appuyer sur deux paradigmes de programmation : l'orienté aux acteurs et l’orienté à l'objet.

Le paradigme orienté aux acteurs m'a permis de découper mon application en différents acteurs indépendants qui communiquent entre eux en envoyant et en recevant des messages. Cela a été particulièrement adapté pour modéliser les différents éléments de l'application (banque, banquiers, clients) et gérer les communications entre eux de manière efficace. J'ai également pu profiter des avantages de la programmation orientée aux acteurs en termes de parallélisation et de performance. En effet, le paradigme orienté aux acteurs est particulièrement adapté pour modéliser les systèmes distribués et asynchrones, comme c'est le cas ici avec notre projet. Le paradigme acteur offre également une grande flexibilité et une facilité de maintenance. En découpant l'application en acteurs indépendants, je peux facilement ajouter ou supprimer des fonctionnalités sans affecter le reste de l'application. De plus, en utilisant des messages pour la communication entre acteurs, je peux facilement changer la manière dont les acteurs interagissent entre eux sans avoir à modifier leur code.

Cependant, je n’ai pas seulement utilisé le paradigme acteur, en effet, celui-ci est souvent utilisé avec d’autres paradigmes comme notamment le paradigme objet, ce que j’ai fait.

Le paradigme orienté objet est un style de programmation qui met l'accent sur la manipulation de données structurées sous la forme d'objets. Ces objets sont des instances de classes qui définissent leurs propriétés (appelées attributs) et leurs comportements (appelés méthodes).

Dans notre cas, j’ai choisi d’utiliser ce paradigme en créant des classes de modèle telles que BanqueModel, BanquierModel, ClientModel et CompteModel. Ces classes définissent les attributs et les méthodes associées à chaque type d'objet dans votre application, permettant ainsi de manipuler et de traiter les données de manière structurée.

Tout cela se justifie car cela me permet de modéliser de manière naturelle et intuitive les entités du domaine métier de mon application, à savoir la banque, les banquiers, les clients et les comptes.

De plus, en utilisant des classes objets dans mon projet de gestionnaire de compte bancaire, j'ai pu bénéficier de l'avantage de la modularité en créant des objets à partir de mes classes de modèle et en les utilisant de manière indépendante dans différentes parties de mon application. Cela m'a permis de diviser mon code en modules indépendants qui peuvent être assemblés de manière à créer un programme complet.

De plus, en utilisant des classes de modèle réutilisables, j'ai également profité de l'avantage de la réutilisabilité. Par exemple, si j'avais besoin de manipuler des données de compte bancaire dans plusieurs parties de mon application, je pouvais utiliser la classe CompteModel de manière réutilisable pour gérer ces données. Cela m'a permis d'éviter de devoir réécrire du code chaque fois que j'avais besoin de manipuler des données de compte, et de réutiliser mon code de manière efficace dans différents contextes.

En résumé, en utilisant le paradigme objet dans mon projet de gestionnaire de compte bancaire, j'ai pu profiter de l'avantage de la modularité en divisant mon code en modules indépendants, ainsi que de l'avantage de la réutilisabilité en utilisant des classes de modèle réutilisables. Ces avantages ont contribué à rendre mon code plus lisible et plus facile à maintenir, et ont permis d'améliorer les performances de mon application.

En ce qui concerne la gestion de donnée, j’ai décider ici, de mettre en place le pattern DAO et le pattern Factory.

Tout d'abord, ces deux patterns permettent de séparer la logique de gestion des données de la logique métier de mon application. En utilisant le pattern DAO, j'ai pu créer des objets spécifiques qui se chargent de récupérer et de mettre à jour les données de ma base de données de manière indépendante de mon application. Cela m'a permis de simplifier la gestion de mes données et de faciliter la maintenance de mon code.

De plus, en utilisant le pattern Factory, j'ai pu centraliser la création de mes objets DAO dans une seule et même classe, ce qui m'a permis de faciliter l'accès aux différents DAO de mon application et de rendre mon code plus lisible et plus maintenable. Cela m'a également permis de faciliter l'ajout ou la modification de mes DAO, en ne modifiant qu'une seule classe au lieu de devoir modifier chaque partie de mon code qui utilise ces DAO.

Je précise que pour ce projet, dans la classe DAO de chaque objetDAO (BanquierDAO, Client DAO,CompteDAO et DemandeDAO) je n’ai pas coder chaque opération CRUD car je n’en avait pas besoin pour ce projet. En effet, il n’est pas indiqué dans l’énnoncé que nous devons gérer la création de compte, de banquier ou encore de client. C’est pourquoi je les inserts directement dans mon script SQL à la creation de la base de données. Une seule fonction est codée dans chaque classe, la fonction get(id). Cependant j’ai quand même implémenté chaque méthode de l’interface DAO dans chaque classe des objets afin de respecter le pattern DAO. (Imaginons que je souhaite améliorer mon projet par la suite, cela me facilitera la tache car tout est déjà prêt.)

## Décomposition du projet en sous-problème

Dans cette partie je vais expliquer comment j’ai décomposé le projet en sous-problème afin de le rentre plus facile à gérer et à résoudre. En d’autres termes, je vais vous expliquer comment j’ai résonné afin de parvenir à la fin du projet.

Tout d’abord, j'ai créé les classes de modèle qui définissent les attributs et les méthodes associées à chaque type d'objet dans mon application, comme la banque, les banquiers, les clients et les comptes. Ensuite, j'ai créé les acteurs qui utiliseront ces classes pour communiquer entre eux et gérer les interactions entre les différentes parties de mon application.

Ensuite, j'ai mis en place une fonction « main » fonctionnel sans base de données, afin de pouvoir tester les différentes fonctionnalités de mon application sans dépendre d'une base de données.

Par la suite, j’ai utilisé le pattern DAO et Factory pour gérer la communication avec la base de données et permettre de stocker et de récupérer les données de manière efficace. J'ai également créé un script de base de données et inséré des comptes de banquiers et de clients dans la base de données.

Ensuite, j'ai modifié l'acteur ActeurBanque pour qu'il puisse stocker les demandes des clients à l'aide d'un objet demandeDAO, et j'ai modifié l'acteur ActeurBanquier pour qu'il puisse mettre à jour la table "compte" de la base de données avec l'objet compteDAO. Enfin, j'ai créé une fonction main utilisable avec la base de données, ainsi qu'un algorithme de création d'un jeu de tests aléatoire.

Grâce à cette décomposition en sous-problèmes, j'ai pu traiter chaque partie de mon projet de manière indépendante et ainsi le rendre plus facile à gérer et à résoudre.

## Problèmes rencontrés durant le projet

Avant de commencer à coder, je visualisais un problème majeur, problème que j’ai eu du mal à résoudre, mais qui m’a permis de bien comprendre le fonctionnement du projet.

C’est un problème de concurrence des données, prenons un exemple :

* Le client « Rémy » possède le compte 1 géré par le banquier 1
* Le client « Tom » possède le compte 3 géré par le banquier 1

Si ces deux clients font plusieurs demandes simultanément imaginons 50 demandes. Ce qui dans la vraie vie est peu probable, mais passons. Cela signifie que plusieurs clients font de nombreuses demandes simultanées sur des comptes gérés par le même banquier, cela peut donc causer des problèmes de concurrence et entraîner des retards dans le traitement des demandes.  
Par conséquent « un askTimeout » peut apparaitre.

Il s ‘avère que j’ai fait des tests, et que le « askTimeout » se levait dès qu’un banquier devait traiter strictement plus de 4 demandes simultanément.

Après plusieurs heures de recherche, je pensais que mon code et mes algorithmes étaient incorrect. J’ai donc réfléchi à implémenter un système de file d’attente dans la classe qui vérifie les demandes de mes clients. J’ai donc voulu utiliser, une liste chainée. Pourquoi ? Car une file d’attente respecte le principe de « premier arrivé, premier servi », et une liste chainée peut facilement respecter cette règle en ajoutant de nouveaux éléments à la fin de la liste et en retirant les éléments du début de liste.

Au cours de mon développement, je me suis dis que ce que je faisais était contraire au principe de Akka », en effet, je cherchais à ce que les banquiers traites de manière séquentielle les demandes. Or, le principe de Akka est de justement, pouvoir traiter plusieurs choses en parallèle, de manière asynchrone. J’ai donc arrêté ceci, et je me suis dis que j’allais essayer autre chose.

D’autant plus que Akka gère déjà une file d'attente interne pour les messages envoyés à un acteur. L'ajout d'une file d'attente supplémentaire pourrait être considéré comme une duplication inutile des fonctionnalités offertes par Akka.

C’est alors, que j’ai décider de tester de mettre cette ligne juste après les demandes des clients dans ma fonction main :

Thread.sleep(500);

C’est ainsi, que j’ai réussi à régler le problème majeur que j’ai eu durant ce projet. Mais alors pourquoi cette ligne permet de ne pas lever de « askTimeout » lorsque nous effectuons beaucoup de demande ? Et bien, derrière Akka ce sont des threads qui fonctionnent en parallèle, cette ligne permet de mettre en pause pendant 500 millisecondes le thread courant. C’est utile afin de réduire la pression sur mon acteur banquier qui est beaucoup demandé.

Même si, ce n’est pas très recommander d’utiliser cette méthode, je n’ai pas trouvé d’autres façon que de faire ainsi.

Une autre solution éventuelle, aurait été de mettre en place une répartition équilibré de la charge de travail des banquiers. Cependant dans l’énoncé du projet, il est dit qu’un banquier s’occupe d’un compte bancaire fixe. Cela serait donc pas possible dans ce cas la si nous décidions d’implémenter une répartition équilibrée de la charge de travail entre les banquiers.

Une autre solution éventuelle, aurait été de mettre en place un système de mise en cache, qui aurait permit de stocker temporairement les données fréquemment demandées afin de réduire la pression sur les banquiers et donc d’améliorer le temps de réponse.

N’ayant pas réussi à implémenter ceci, j’ai donc choisi d’utiliser la solution évoquer de Thread.sleep() évoquer précédemment.

# Architecture logicielle détaillée

L'architecture logicielle de mon application de gestion de comptes bancaires est basée sur l'utilisation d'Akka en Java. J'ai découpé mon projet en trois packages principaux :

Une image contenant texte

Description générée automatiquement

6 : Liste des packages avec leurs contenus

* Le package « actors » contient les classes Actor d'Akka qui représentent les différents acteurs de l'application (banque, banquiers, clients). Ces classes sont responsables de la gestion des messages et de la communication entre les différents acteurs de l'application.
* Le package « DAO » (Data Access Object) contient les classes qui gèrent la connexion à la base de données et l'accès aux données. Ces classes utilisent les patterns DAO et DAO Factory pour offrir une interface uniforme pour accéder aux différentes tables de la base de données (banquiers, clients, comptes, demandes).
* Le package « model » contient les classes métier qui représentent les objets de l'application (banque, banquiers, clients, comptes). Ces classes sont utilisées par les classes d'acteurs pour gérer les données de l'application et effectuer les opérations métier.

En plus de ces packages, j'ai également une classe Main qui est responsable de démarrer l'application et de créer les acteurs principaux.

On remarque donc que 2 couches se distingue, la première la couche métier et la couche de données.

La couche métier est responsable de la logique métier de l'application, c'est-à-dire la gestion des comptes bancaires, des banquiers et des clients. Cette couche est composée des acteurs qui représentent les différents éléments de l'application et des modèles qui représentent les données de l'application.

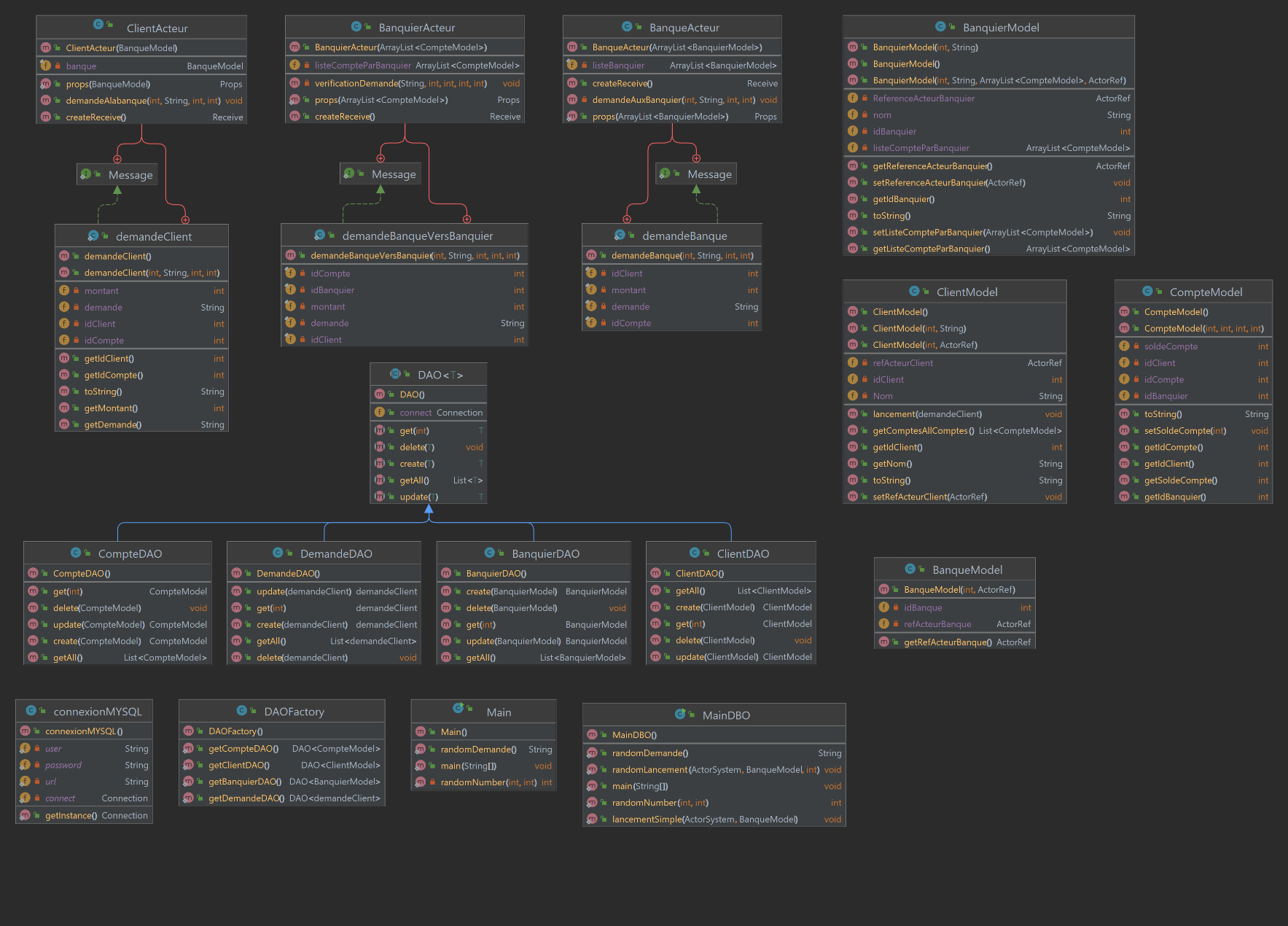
La couche de données, quant à elle, est responsable de l'accès et de la gestion des données de l'application. Elle est composée des classes DAO qui permettent de réaliser les opérations CRUD (create, read, update, delete) sur les données de l'application.

Le choix de cette architecture en deux couches permet de séparer clairement les responsabilités de chaque partie de l'application, ce qui rend le code plus lisible et plus facile à maintenir. Cela permet également de rendre l'application plus flexible en facilitant l'ajout ou la modification de fonctionnalités sans avoir à toucher au code de l'autre couche. Comme je l’avais dis précédemment.

En utilisant cette architecture, il est également plus facile de changer la manière dont les données sont stockées, par exemple en passant d'une base de données relationnelle à un autre type de stockage de données, sans avoir à modifier le code de la couche métier.

Donc imaginons que si je voulais tout stocker dans des fichiers XML , cela serait possible sans modifié tous le code.

Précédemment, nous avions vus un premier aperçu des classes de ce projets cependant nous n’avions aucun aperçu des méthodes et dépendances de ces classes, regardons et commentons cela à l’aide d’un diagramme de classe (sans les dépendances entre classe pour l’instant pour des questions de lisibilité :

Pour plus de détails, l’image est visible dans le dossier « [gestionCompteBancaire/Docs/diagrammeClasse/diagramme\_de\_classe\_sans\_dependences.png](diagrammeClasse/diagramme_de_classe_sans_dependences.png)»

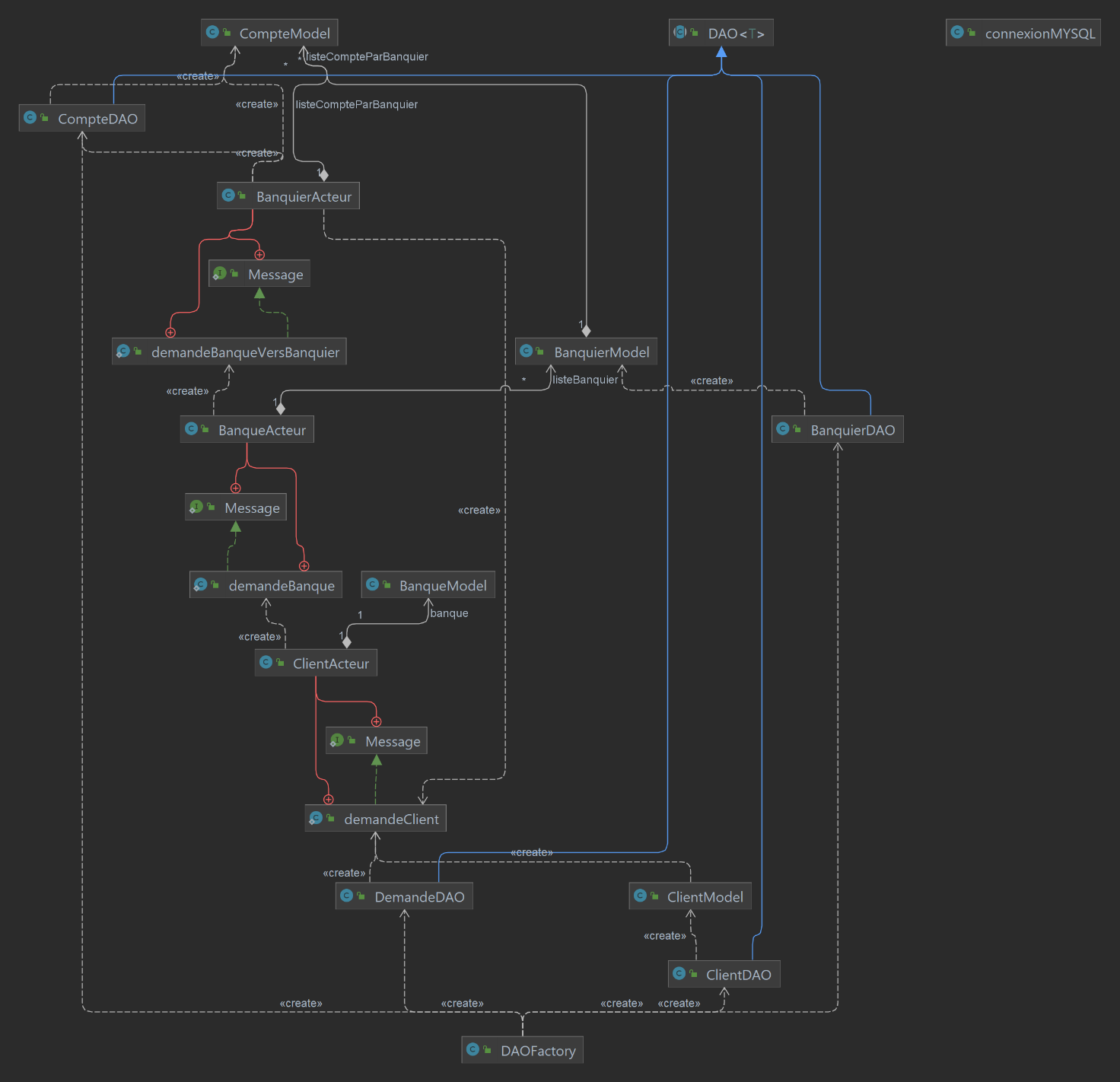
Je ne vais pas décrire attributs par attributs, méthode par méthode ce diagramme car dans la partie « Spécification des classes ou fonctions principales » je vous renverrai vers la javadoc de mon projet où la vous pourrez consulter en détail chaque classe de mon projet.

Enfin voici les dépendances entre chaque classe de mon diagramme de classe, j’ai fais au mieux pour la lisibilité du diagramme, j’ai également pas afficher les attributs, propriété et méthode des classes mais ce sont les mêmes que sur la capture d’écran précédente. Si vous souhaitez visualiser le diagramme de classe complet, j’ai laissé une image pour ça dans le répertoire :

« [gestionCompteBancaire/Docs/diagrammeClasse/diagramme\_de\_classe\_complet.png](diagrammeClasse/diagramme_de_classe_complet.png)»

7 : Diagramme de classe avec dépendances

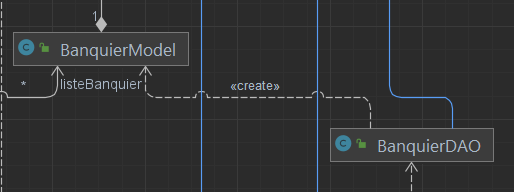
Revenons à notre diagramme de classe avec seulement nos dépendances. Je n’ai pas ajouté les classes mains car ici ce n’est pas très intéressant de savoir que la classe main créer tels instance. Nous savons très bien que la classe main va créer une instance de la plupart des objets.



8 : Diagramme de classe avec dépendances

Comment interagissent donc nos classes entre elles ?

Tout d’abord, partout où l’ont voit le mot «create » cela veut dire qu’on créer une instance de l’objet destinataire à partir de la classe expéditrice. Prenons un exemple :

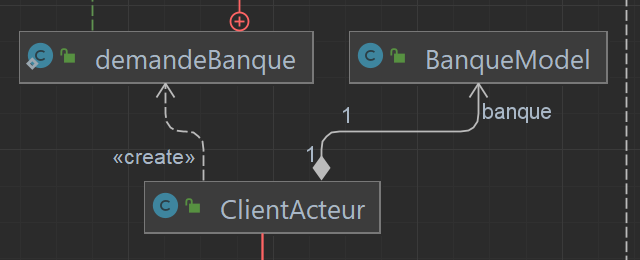


9 : exemple de d'instanciation

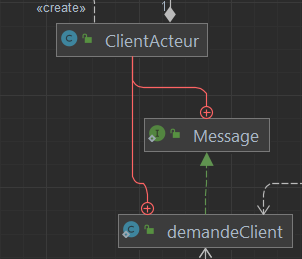
Dans banquierDAO, nous instancions un objet de type « BanquierModel ». De cette manière :

BanquierModel banquier = new BanquierModel();

Voilà ce que signifie le mot « create ».

On remarque également qu’on a plusieurs compositions, illustrons ce que ça signifie avec un exemple :   


10 : Exemple de composition

On remarque ici, une composition entre ClientActeur et BanqueModel, cela veut dire que dans client acteur, nous aurons un attribut banque de type BanqueModel.

Si l’on regarde la capture d’écran ci-contre, on remarque deux types de flèche :

* Les « flèches »rouge symbolise une inner classe, cela veut dire que dans la classe Client Acteur, il y a une classe interne nommé « demandeClient » et une interface nommé « Message ».
* La flèche verte elle symbolise, l’imprémentation de l’interface, cela signifie que la classe « demandeClient » implémente l’interface « Message ».

Si l’on se refie à cette capture d’écran (8 : Diagramme de classe avec dépendances), on remarque des flèches bleues. Les flèches bleues représentent l’héritage, par exemple, « demandeDAO », hérite de la classe « DAO ». Plus simplement, toutes les classes liées à la couche de la base de données hérite de la classe « DAO ». Pourquoi ? Pour respecter le pattern DAO. En effet, DAO est une classe abstraite, qui va regrouper toutes les méthodes que les objets DAO devront implémenter.

Ainsi, j’ai défini tous les types de relations qui existe entre chacune de mes classes.

Avant de passer à la partie présentant chacune Spécification des classes, je souhaiterais revenir sur l’architecture de ma base de données :

Pour rappel, voici le diagramme de ma base de données : (2 : Diagramme de la base de données).

Je vais définir les relations entre les tables ce qui justifiera les clés étrangères :

* La relation client/compte :

Un client peut avoir 0 ou plusieurs comptes et un compte est possédé par 1 seul et même client. D’où la clé étrangère idClient dans Compte. C’est car nous avons une relation de type (1,N) .

* La relation client/demande :

Un client fait 0 ou plusieurs demandes et une demande peut être faite par 1 seul et même client. Cela explique la clé étrangère idClient dans demande, pour la même raison que précédemment, nous avons une relation de type(1,N)

* La relation banquier/compte :

Un banquier gère 0 ou plusieurs demandes et une demande peut être géré par un seul banquier. Encore une fois ici, nous avons une relation de type(1,N), ce qui explique la clé étrangère idBanquier dans la table compte

* La relation compte/demande

Une demande concerne 1 seul compte et un compte peut être concerné par plusieurs demandes. Encore une fois, on a une relation de type(1,N) ce qui explique la clé étrangère idCompte dans demande.

Ainsi, j’espère vous avoir bien fais compris le raisonnement que j’ai eu afin de mettre en place la base de données qui sera utilisé plus tard.

En résumé, j’ai détaillé dans cette partie comment est architecturer mon projet, en vous détaillant mon diagramme de classe ainsi que les interactions entre chaque classe.

Passons à présent à la spécification des classes.

# SpÉcification des classes

Pour cette partie, je vais décrire chaque attribut, chaque méthode de chaque classe du projet.

Pour cela, ce ne serait pas très pratique et pas très organisé de tout décrire dans ce document word, c’est pourquoi j’ai réalisé une documentation (javadoc) complète afin de comprendre ce que chaque classe et chaque méthode de ces classes permettent de faire.

Cliquer « [ici](../javaDoc/index.html) » pour lancer « index.html » de la javadoc et voir la description de chaque classe et méthode.

Si pour quelconque raison vous n’arrivez pas à ouvrir la javadoc avec ce lien, celle-ci se trouve a l’emplacement suivant : « \[gestionCompteBancaire\javaDoc\index.html](../javaDoc/index.html)».

Pour plus de détail technique, je vous invite à consulter le code directement, celui-ci est commenté afin de simplifier sa compréhension.

# documentation de compilation

## Prérequis

Quels sont les prérequis pour ce projet :

* Un ordinateur avec une version récente de Java(Personnellement, avec ce projet, j’en ai profité pour mettre à jour ma version et j’ai utilisé java 19.0.1, mais le code marchera avec une version antérieur), installée. Vous pouvez télécharger Java à partir du site web [d’Oracle](https://www.oracle.com/fr/java/technologies/downloads/).
* Vous avez besoin d’avoir maven sur votre machine, personnellement ma version de maven est : « Apache Maven 3.8.6 ».
* Vous devez installer MySQL sur votre machine afin de pouvoir voir l’état des tables du projets, personnellement j’ai utilisé la version 8.0.31. Vous pouvez aussi utiliser un logiciel de visualisation de SGBD pour plus de visibilité et éviter de répéter les mêmes requêtes à chaque fois.
* Un navigateur web pour accéder à la javadoc.
* Pensez également à bien déclarer vos variables d’environnement afin de pouvoir exécuter les commandes dans l’invite de commande.

## Comment créer la base de données

Pour créer la base de données il vous faudra exécuter le script SQL que j’ai créé, il est disponible à l’adresse suivante :

« [gestionCompteBancaire\script.sql](../script.sql) ».

Pour cela il faut suivre une à une les étapes suivantes :

* Vérifier que vous avez bien mysql sur votre machine
* Ouvrer une invite de commande.
* Diriger vous dans l’invite de commande à l’adresse de « script.sql », ici il est à la racine du projet
* Se connecter à votre compte mysql avec la commande suivante :

mysql -u username -p

Dans mon cas, je me connecte a l’utilisateur root.

Mysql -u root -p

Votre mot de passe va être demandé, vous le rentrez.

Une fois cela fait rentrer la commande suivante :

create database banquev2;

Il s’agit du nom que j’ai donné à ma base de données, ce nom est présent dans le code java, donc si vous ne mettez pas ce même nom votre programme de ne fonctionnera pas.

Une fois que cela est fait vous pouvez vérifier quelle a bien été créer en faisant :

show databases ;

Enfin, faite :

use banquev2;

Une fois cela fait, faite :

Source script.sql

Vérifier à bien être au même endroit que le script sinon cela ne marchera pas, si tout ce passe bien vous devrez avoir ceci :

Une image contenant texte

Description générée automatiquement

11 : creation de la base de données réussi

Une fois cela fait vous pouvez compiler le projet.

## Comment compiler le projet

Comme j’ai tous configuré dans le fichier « pom.xml »,et que j’ai inclus les librairies . Il faut il suffit de se mettre à la racine du répertoire : « /gestionCompteBancaire ».

Une fois ici, lancer cette commande :

> mvn compile

Cette commande permet de compiler le projet.

Vous devriez avoir quelque chose similaire a ceci :

Une image contenant texte

Description générée automatiquement

12 : Capture d'écran du succès de la compilation

Une fois cette compilation faite et réussi. Vous devez exécuter cette commande afin de lancer le projet :   
> mvn exec :java

Cette commande permet de lancer la fonction main du projet, ainsi vous pouvez profiter du projet.

Une autre possibilité est de lancer le projet via un IDE, tel que netbeans ou intelliJ IDEA.

# jeu de tests

Il existe 2 modes pour mon programme :

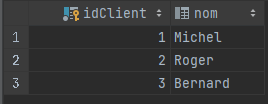
-Le mode simple

-Le mode aléatoire

Avant des les décrire, prenons conscience de l’État de mes données avant de lancer le programme.

Car je le rappelle, je n’ai pas géré la possibilité de créer des clients ou bien des banquiers directement dans mon programme. Nous partons du principe qu’ils existent déjà. Donc pour les créer je passe par des requêtés SQL. Vous pouvez les visualiser dans le script.

## Les donnée de ma base

Quels sont nos clients, de notre base de données ?  
Nous en avons 3.

13 : clients

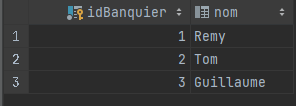
Quels sont les comptes de la base de données, afin de voir quel compte appartient à quels client :

Une image contenant texte

Description générée automatiquement

14 : comptes

Enfin voyons nos banquiers :



15 : banquiers

Voici donc l’ensemble de nos données de tests, cela peut sembler peu, en revanche vous avez une influence sur le nombre de demandes de ces clients. Comme je l’ai dit plusieurs fois, j’ai agi ainsi, car j’ai interprété le sujet ainsi. Nous avons une banque déjà en place, avec des clients et des comptes précis et ces clients font des demandes sur ces comptes.

Une autre raison est que faire des tests sur 100 comptes aurait été beaucoup moins pratique pour moi.

Passons aux explications du premier mode, le mode simple, vous pouvez choisir de lancer ce mode à l’exécution du programme.

Ce mode est un petit mode pour essentiellement utile afin de voir que tous les acteurs communiquent bien entre eux et de voir que les bons messages sont renvoyé aux clients lorsque celui-ci fais ses demandes.

Ici, j’ai choisi de faire 7 demandes :

* 1 demande sur chaque compte des clients
* 1 demande sur un compte qui n’existe pas
* 1 demande sur un compte qui n’appartient pas au client faisant la demande

Observons les résultats :

Une image contenant texte

Description générée automatiquement

Dans un premier temps, on remarque que les données arrive pas forcement dans l’ordre, c’est totalement normal, comme je l’ai expliquer précédemment, « akka » fonctionne de manière asynchrone et les banquiers traitent les demandes parallèlement. Ainsi cela justifie cela.

On remarque qu’on bien 5 bonne réponse et 2 mauvaise (ligne 52 et 64) car pour la réponse ligne 52, le client 2 fais une demande sur le compte n°9 qui n’existe pas en base de données.

Et pour la ligne 64, c’est la réponse à une demande concernant le compte 4 qui n’appartient pas au client 2 mais au client 3.

Pour le mode aléatoire, je ne vais pas joindre de capture d’écran car il s’agit du même principe à la différence qu’ici, j’ai voulu simuler un semblant de réalisme en ajoutant une probabilité qu’un client ne fasse pas de demande et qu’il fasse uniquement des demandes sur un de ces comptes.   
D’où la faite d’avoir traité précédemment le cas des erreurs possible afin de vous montrer que j’ai bien gérer la gestion des messages en cas d’erreurs.

# conclusion

Pour conclure, grâce dans ce projet de gestionnaire de comptes bancaire, j'ai beaucoup mieux maîtrisé Akka car en TP je n'avais pas eu le temps de tout faire. J'ai pu bien comprendre ce qu'est la programmation acteur, et j'ai également remis à jour mes connaissances sur la programmation objet et sur les différents paradigmes DAO et Factory. J'ai appris à lier du code java à une base de données. J'ai également compris l'importance de mettre en place ce genre de système parallèle, notamment dans le cas d'une gestion bancaire, pour éviter tout ce qui touche au conflit de concurrence.

Toutefois, je dois ajouter que, dans mon cas, comme j'ai choisi d'utiliser une seule banque, ce n'est pas très sécurisé. Si la banque fermait ou faisait faillite, il n'y aurait pas de solution pour les clients pour garder leur argent. Cependant, cet argent serait stocké sur des serveurs car j'ai tout de même mis en place un système de persistance des données.

En utilisant Akka, j'ai également pu bénéficier de la flexibilité offerte par le modèle acteur pour évoluer mon programme au fil du temps. J'ai pu ajouter de nouvelles fonctionnalités en créant de nouveaux acteurs, et j'ai pu adapter mon programme à de nouvelles exigences en modifiant les comportements des acteurs existants. Cela m'a permis de rendre mon programme plus évolutif et plus adaptatif.

En résumé, j'ai utilisé Akka dans ce projet de gestionnaire de comptes bancaire pour gérer la facilité de maintenance du programme en offrant une gestion simplifiée de la concurrence, de la communication entre les acteurs et de l'évolution du programme au fil du temps. Cela m'a permis de rendre mon code plus modulaire, plus fiable et plus évolutif.