IUT de Paris - Rives de Seine
Université Paris Cité

Rapport de projet

Architecture logicielle : Projet médiathèque

Yohan RUDNY – 209 Parcours B

Daniel SZATYN – 208 Parcours A

BUT 2 FA – Semestre 4

Promotion 2022 – 2023

Introduction, structuration globale du projet et outils utilisés

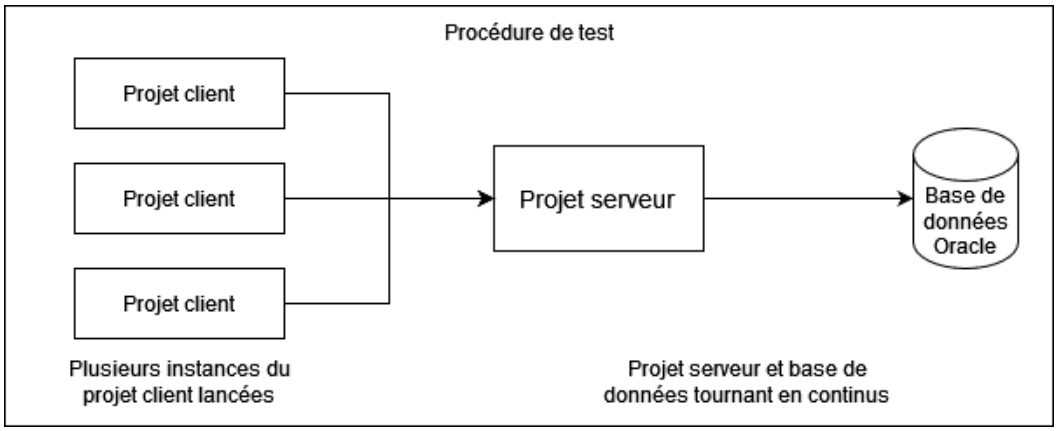
Dans le cadre de ce projet de réalisation d'une médiathèque comprenant des échanges client-serveur ainsi que des interactions avec une base de données : nous avons sélectionné la structuration et les outils de travail (IDE, SGBD…) les plus adéquats avec le contexte présenté. Bien que cela n'impacte que peu le rendu final, il est premièrement important de présenter et justifier nos choix vis-à-vis de ces outils.

- Environnement de développement **IntelliJ IDEA**
  L'IDE IntelliJ nous a permis de rendre notre code plus optimisé et compacte, ce qui améliore considérablement la lisibilité et la maintenance évolutive de celui-ci. La plus-value qu'apporte l'usage de cet IDE dans notre contexte est la fonctionnalité de gestion en temps réel de notre base de données directement depuis l'interface d'IntelliJ. En s'y connectant en local, toutes nos tables ainsi que nos données sont visibles et gérables depuis notre IDE sans passer par l'interface du SGBD, ce qui a permis de nous éviter un bon nombre de confusions sur les requêtes SQL à écrire et sur la cohérence des tables et de leurs contraintes.

- SGBD **Oracle XE 11G**
  Nous avons naturellement sélectionné Oracle en tant que SGBD pour ce projet. Celui-ci étant pris comme exemple dans les cours et TP d'architecture logicielle auxquels nous avons assistés, nous nous sommes rapidement familiarisés avec son installation et son fonctionnement et n'avons pas perdu de temps à essayer de faire fonctionner un autre SGBD que celui-ci.

- **GitHub**
  Pour collaborer efficacement en binôme, nous avons utilisé GitHub et créé deux répertoires (client et serveur) dans lesquels nous déposions nos avancées en y laissant un commentaire à chaque modification.

À l'aide ces outils, nous avons mis en place la structuration de projet qui nous a paru la plus simple d'utilisation au vu des tests que nous avons effectué tout au long du développement. Deux projets Java distincts ont été créés : **un projet client et un projet serveur**. L'idée a été de se mettre en conditions réelles d'utilisation du service de médiathèque. Lors de nos tests, le projet serveur et la base de données tournent en continu puis le projet client est lancé (souvent en plusieurs instances pour tester la concurrence) à un port choisi au préalable.



1. Programmation côté serveur

1.1. Structuration et stabilité du code

Le projet côté serveur est composé de 7 packages pour un total de 18 classes Java. Chaque package regroupe des classes portant sur un aspect précis de l'application et ont pour objectif de toutes respecter certains concepts de programmation ainsi que les principes SOLID.

- **SRP** : chacune des classes possède une responsabilité unique. Par exemple, la classe « **DataHandler** » a pour unique mission de gérer les échanges entre l'application et la base de données Oracle. De même que la classe « **TimerHandler** » ne sert qu'à amorcer ou annuler les tâches planifiées du serveur (annulation d'une réservation après 2h, bannissement d'un abonné après 2 semaines de retard…).

- **OCP/LSP/ISP/DI** : les évolutions de l'application consistent, dans notre contexte, à rajouter des types de documents différents et d'autres services. Les méthodes contenues dans l'interface « **Document** » sont codées dans une classe « **AbstractDocument** » qui implémente celle-ci et qui est héritée par la classe « **DVD** ». Les méthodes de l'interface ne sont pas à modifier pour l'ajout d'un nouveau type de document. Elles peuvent uniquement être spécialisées par les classes filles (DVD, Livre…). Explications plus détaillées dans la partie « Maintenabilité évolutive » de ce rapport.

- **Encapsulation** : les données de toutes les classes ont été encapsulées correctement en fonction des besoins. Exemple : « **checkAbonne** » et « **checkDocument** » sont des méthodes « **protected** » car uniquement utilisées par les sous-classes de « Service ».

1.2. Découplage

En dehors des bibliothèques logicielles, le découplage des classes au sein des packages a pour but de rendre les dépendances cohérentes et compréhensibles par n'importe quel développeur ayant le contexte du projet en vue de potentielles maintenances futures.

- « **mediatheque** » regroupe explicitement les objets exploités par le serveur. C'est ici que l'on ajoutera naturellement les nouveaux types de documents ou modifierons les méthodes de base de ceux-ci (emprunt, réservation, retour) en vue d'un changement de fonctionnement global.

- « **data** » est le package gérant les données stockées par le serveur. On y trouve la classe « **DataHandler** » responsable des échanges avec la base de données ou « **TimerHandler** » qui gère l'instanciation des tâches planifiées. Toutes les classes contenues dans ce package sont « **static** » et la plupart des méthodes sont faites pour être appelées à tout moment dans notre code.

- « **tasks** » contient toutes les tâches planifiées que le serveur est susceptible d'invoquer. Il est facile d'y ajouter, au besoin, une nouvelle tâche et de gérer ses invocations depuis la classe « **TimerHandler** ».

- « **services** » est composé des trois classes de services demandées pour ce projet. Tous sont des classes filles de la classe abstraite « Service » contenue dans la bibliothèque logicielle « **bserveur** ». Si un nouveau service doit être implémenté dans la médiathèque, il sera ajouté dans ce package et devra hériter de la même classe mère que les autres pour bénéficier des méthodes de celle-ci.

- « **exception** » contient la classe « **RestrictionException** » qui peut être levée par la classe « **DVD** ». Un package lui est dédiée car d'autres exceptions peuvent, au besoin, être ajoutées en suivant le même schéma.

La classe principale « **Application** » n'est contenue dans aucun package. Elle se contente simplement d'instancier trois threads, correspondant respectivement aux trois services demandés, et d'invoquer une nouvelle instance de « **DataHandler** » pour importer les données de la base de données vers les listes.

1.3. Bibliothèques logicielles mises en œuvre et utilisées

Côté serveur, **quatre bibliothèques logicielles** sont nécessaires au bon fonctionnement du code dont deux qui ont été mises réalisées durant le TP 4 d'architecture logicielle.

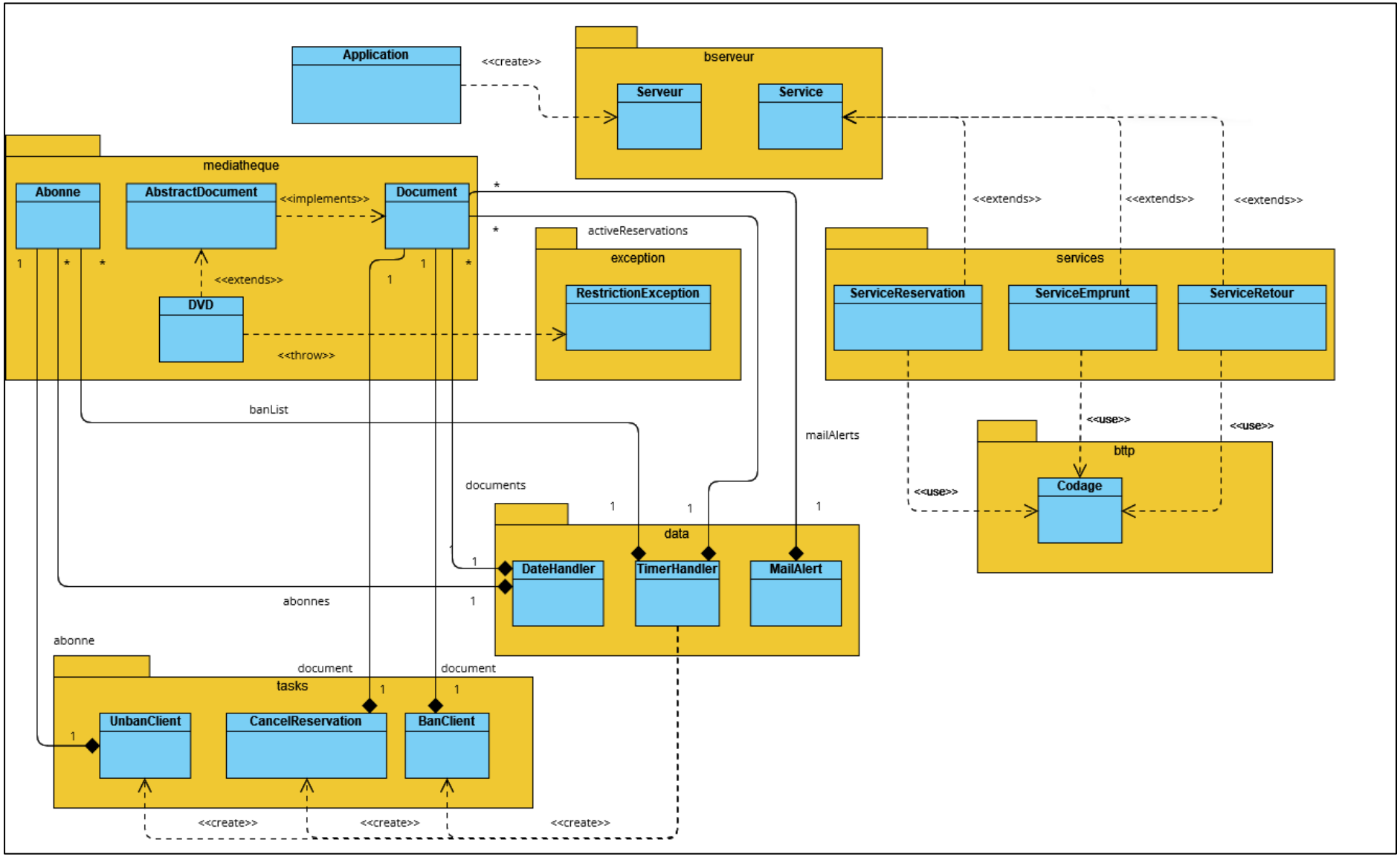
- **bserveur**
  Composée d'un couple « **Serveur – Service** » dans un package « **serveur** », cette bibliothèque permet une généralisation des services. Un service prend en paramètre un socket client et sera utilisé pour instancier un nouveau serveur qui prendra à son tour en paramètre un service ainsi qu'un port sur lequel le client pourra se connecter. Le main de notre projet serveur créer un nouveau thread de « **Serveur** » par service mis en place (ici trois services). Cette bibliothèque permet d'énormément faciliter la maintenance évolutive de l'application car pour ajouter un service : il suffit d'ajouter une classe héritant de la classe « **Service** » et de créer un nouveau thread dans le main au port voulu.

- **bttp2**
  Cette bibliothèque est présente côté serveur et client et contient une seule classe « **Codage** ». Celle-ci est composée de deux méthodes statiques « **coder** » et « **decoder** » qui permettent de remplacer les « \n » par des « ## » puis inversion afin de permettre à la méthode de lecture « **bufferedReader.readLine()** » de prendre en compte les sauts de ligne.

- **javax.mail et activation**
  Ces deux bibliothèques permettent, complémentairement, de fournir les classes et les méthodes nécessaires à l'envoi de mail en langage Java. Elles ont été utilisées dans le cadre du BretteSoft© Sitting Bull que nous détaillerons plus tard dans ce rapport.

Nous précisons que dans le rendu des sources : les bibliothèques « **bserveur** » et « **bttp2** » ont été glissées en tant que packages dans le répertoire « **librairies** » afin que les codes sources soient consultables. En conditions réelles d'utilisation de l'application, l'idéal est de les importer au projet en tant que bibliothèques au format « **.jar** » comme vu en TP.

1.4. Graphe des dépendances inter-package



2. Programmation côté client (échanges client/serveur)

Au niveau du client, **les échanges sont assurés par un client BTTP 2.0**. Une classe « **ClientBttp** » a été écrite et est utilisée pour factoriser la lecture et l'envoie de chaînes de caractères avec le serveur dans le contexte d'un échange de type « question – réponse ». Développée lors du TP 4, le client BTTP est **indépendant du service et des questions posées**. Cette classe est un thread (qui hérite donc de Runnable) qui comprends, en plus du run, deux méthodes :

- « **read()** » affiche dans la console du client le message reçu par le serveur via un attribut « **sIn** » de type BufferedReader. La chaîne de caractères reçue passe dans la méthode « **decoder** » de la bibliothèque « **bttp2** » afin de lui rendre son affichage originel, celle-ci étant passée dans la méthode « coder » lors de l'envoi par le serveur.

- « **send()** » va lire l'entrée clavier du client à l'aide d'un second BufferedReader et l'envoyer au serveur via un attribut « **sOut** » de type PrintWriter. Entre ces deux opérations, la méthode va vérifier si la chaîne entrée par le client est égale à « stop ». Il s'agit de la seconde condition d'arrêt de l'application rappelée à chaque lancement du client. Si la chaîne « stop » est entrée dans le BufferedReader, le socket client se ferme et l'exécution prend fin.

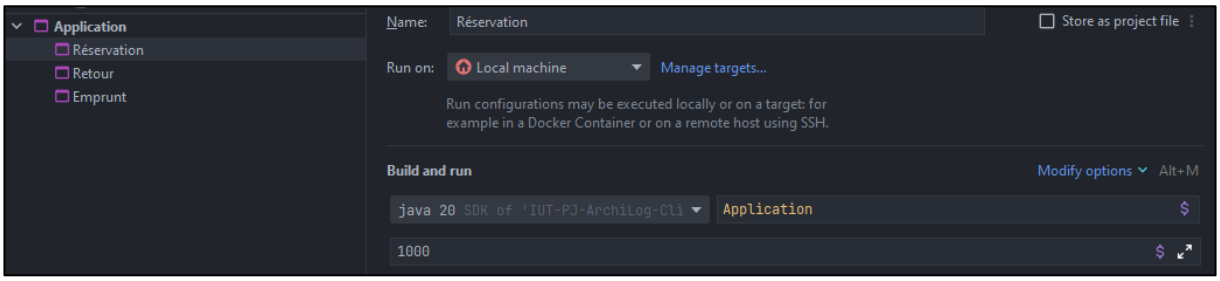
La méthode run va exécuter successivement dans une boucle while les méthodes « read() » puis « send() ». La première condition d'arrêt du client (en plus du « stop ») est **la capture d'une IOException**, signifiant que **le socket client a été fermé côté serveur**. Lorsque le serveur ferme le socket client, cette exception est levée et affiche dans la console client le message « Échanges terminées. ».

Le main « Application » du client a été réfléchi de façon à **pouvoir centraliser la connexion aux services dans une seule et même classe**.

Pour se faire, nous avons utilisé **les arguments passés en paramètre lors du lancement de l'application**. Un seul argument est doit être passé lors du lancement pour pouvoir faire fonctionner l'application : un numéro de port valide.

Une liste statique et constante contenant les numéros de ports valides est définie dans la classe principale. Au démarrage de l'application, après plusieurs vérifications au niveau de l'argument « args[0] » : un nouveau thread « **ClientBttp** » est créé en passant l'argument de spécifié en paramètre en tant que port de connexion.

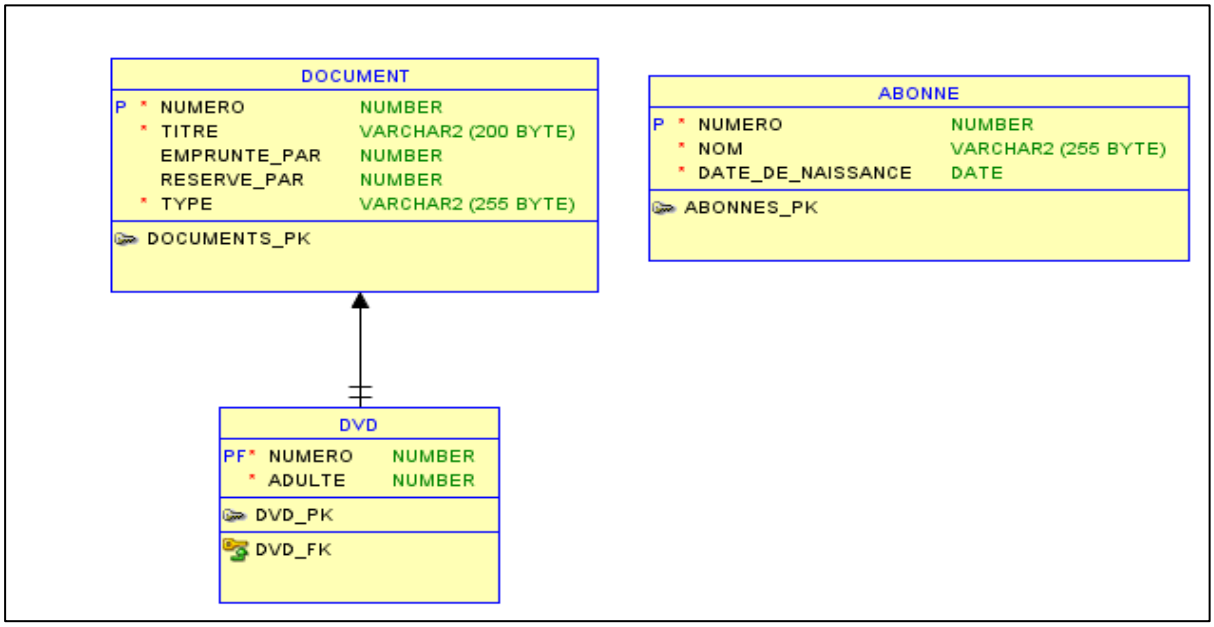
Dans la pratique : nous pouvons imaginer que le logiciel de réservation installé chez le client est préconfiguré pour choisir le port 1000 au lancement et pareil avec les bornes emprunt et retour de la médiathèque avec les ports 1001 et 1002. Nous avons pu créer ces trois configurations dans notre IDE pour effectuer nos tests.



3. Interactions entre la base de données et l'application serveur

3.1. La base de données relationnelle

La base de données est composée de trois tables : « Document », « DVD » et « Abonne ». Les colonnes sont celles énoncées dans le sujet du projet. Il est possible de savoir quel abonné a emprunté ou réservé quel document et les diverses informations des document et abonnés sont également stockées. Le schéma relationnel de notre base est le suivant :



Il a été décidé de créer une table mère « Document » qui sera héritée par une table fille « DVD ». L'idée ici est, pour ajouter de nouveaux types de document, d'avoir simplement à créer une nouvelle table contenant une colonne « Numéro » en clé primaire-étrangère référant un document existant et des champs spécifiques au type à ajouter (par exemple « Livre » avec un champ « nbPages »). Une colonne « Type » est prévue dans la table des documents indiquant explicitement le nom du type de celui. Pour rendre tout cela plus simple, on pourrait imaginer un trigger qui alimenterait la table DVD à chaque ajout de type « DVD » dans la classe Document.

3.2. Chargement des données

Le chargement des données (liste d'abonnés et de documents) se fait au début et respectivement par l'invocation des fonctions « **fetchAbonnes** » et « **fetchAllDocuments** » dans le constructeur de la classe statique « **DataHandler** ». Une nouvelle instance de cette même classe est créée par un appel « **getConstructor().newInstance()** » après la création des trois threads dans le main du projet serveur pour importer toutes les données.

La méthode d'importation des abonnés est très simple. Une requête sélectionnant tous les abonnés est effectuée à la BD à l'aide d'une « PreparedStatement » puis les résultats sont stockés dans une variable « ResultSet ». Dans une boucle while, tant qu'il y a des résultats (méthode « next »), on instancie un nouvel abonné de la classe « Abonne » avec les données obtenues puis on l'ajoute dans la liste statique « abonnes » de la classe de gestion de la BD.

Pour importer les documents, la procédure est similaire mais avec une particularité : la prise en compte du type de document à instancier. La requête SQL inclue une jointure avec la table « DVD » afin d'obtenir la colonne spécifique « adulte ». Un champ de caractères « Type » dans la table des documents permet de connaître son type (DVD, livre, CD…) et de l'obtenir côté Java. Un switch case est mis en place avec des cas en fonction du type de document récupéré. Le cas « DVD » ajoute à la liste des documents une nouvelle instance de la classe DVD qui prend donc en compte le champ « adulte » qui n'est pas présent pour les autres types.

```
switch (type.toLowerCase()) {
    case "dvd" -> documents.add(new DVD(numero, adulte, emprunteur,
reserveur));
    default -> throw new RuntimeException("Type de document non pris en
charge par l'application.");
}
```

3.3. Mise à jour de la base

Afin de sauvegarder les données et de tenir la BD à jour de toutes modifications faites sur les documents dans le cadre d'une mise en place d'applications deux-tier : **nous avons décidé d'effectuer des « Update » à la fin de l'échange entre le client et le serveur.**

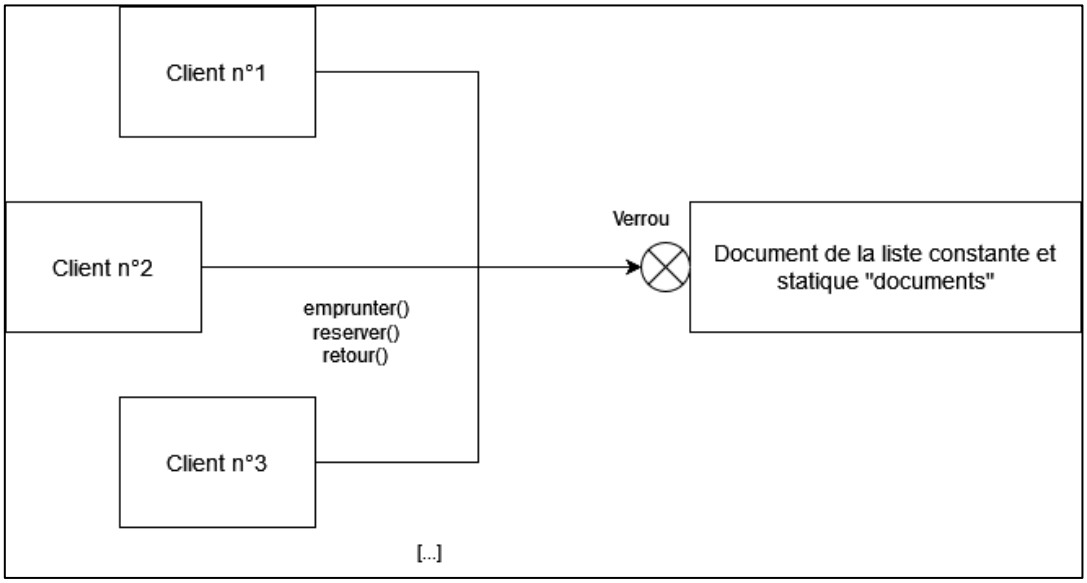
Une méthode statique « **updateDataBase** » est présente dans la classe « DataHandler » et prend en paramètre un document. Celle-ci va simplement, pour un document, récupérer ses attributs « **emprunteur** » et « **reserveur** » et faire une update de la BD avec les numéros récupérés (l'auto-commit étant activé). Toujours à l'aide d'une PreparedStatement : trois arguments sont passés à la requête (numéros emprunteur, réserveur et document) à l'aide de « setObject » (pour prendre en compte le résultat « null » côté BD).

Cette méthode est appelée à la fin des méthodes d'emprunt, de réservation et de retour d'un document dans la classe « AbstractDocument » après que les attributs des objets « Document » aient été mis à jour de manière synchrone.

4. Gestion de la concurrence

Côté serveur, nous avons pu identifier **trois ressources partagées** : les documents, les collections de Timer de la classe « TimerHandler » et la mise à jour du booléen « banned » de la classe « Abonne ».

Les méthodes **emprunter, reserver et retour** de la classe « **AbstractDocument** » présentent des sections critiques modifiant directement les valeurs des attributs emprunteur et reserveur de la ressource partagée, ce qui peut potentiellement causer des problèmes de thread-safety dus aux accès concurrents. La solution utilisée est de **placer des verrous** au niveau de ces trois méthodes pour **séquencer l'accès** aux attributs de AbstractDocument. Les sections critiques ont alors été encapsulées dans des blocs « **synchronised (this)** » permettant au moniteur de Hoare du thread en cours de posséder le verrou jusqu'à la fin de l'exécution de ces blocs et qu'aucun autre thread ne vienne interférer dans la modification des valeurs des attributs du document.



Au niveau de la classe « TimerHandler », trois attributs statiques représentés par des HashMap ont pour rôle de stocker les différents Timer avec le document ou l'abonné correspondant dans le but d'exécuter des tâches planifiées. HashMap **n'étant pas un type d'objet thread-safe** et celles-ci étant mises à jour en fonction des emprunts, réservations et retour des clients : il y a un accès concurrent. Il a donc été décidé de remplacer ces attributs par des « **ConcurrentHashMap** » qui est simplement **un équivalent thread-safe** des HashMap.

Pour finir avec la concurrence, dans la classe « Abonne » se trouve **un setteur « setBanned »** pour l'attribut booléen « banned » d'un abonné (utilisé pour le Brettesoft Geronimo). Celui-ci a donc été défini comme synchronized pour éviter une nouvelle fois les problèmes d'accès concurrents.

5. Maintenance évolutive

5.1. Ajouter un nouveau type de document

Afin de respecter le principe SOLID « OCP » visant à limiter les modifications à l'intérieur des classes au profit d'en créer de nouvelles : nous avons intégré une classe abstraite implémentant l'interface « Document ». C'est au sein de celle-ci que sont codées les méthodes de l'interface et donc utilisables par n'importe quel document. La classe « DVD » hérite de cette classe abstraite et y spécialise la levée d'une « RestrictionException » propre aux DVD vérifiant l'âge de l'abonné et le statut « adulte » du DVD.

```
@Override
public void emprunt(Abonne ab) throws RestrictionException {
    assert (reservePar() != null && reservePar() != ab);
    if (adulte && ab.getAge() < 16) {
        throw new RestrictionException("Vous n'avez pas l'âge requis pour
emprunter ce document.");
    }
    super.emprunt(ab);
}
```

Pour ajouter un nouveau document à prendre en charge : **il suffit de créer dans le package « mediatheque » une nouvelle classe (exemple : Livre) héritant de « AbstractDocument » et, si besoin est, d'ajouter les attributs spécifiques et de spécialiser les méthodes.** Cela permet de ne jamais modifier la classe abstraite et les méthodes **qui resteront toujours telle quel.**

Pour que l'application importe depuis la BD un nouveau type de document, une très légère maintenance est à faire dans « DataHandler ». Après avoir rajouté une table à la BD, dans la méthode « fetchAllDocuments » : il faut **ajouter une jointure à la requête SQL puis ajouter un cas au nom du type de document dans le switch case** créant une nouvelle instance de la classe nouvellement créée.

```
PreparedStatement psDocs = connection.prepareStatement("SELECT * FROM
DOCUMENT JOIN DVD ON DOCUMENT.NUMERO = DVD.NUMERO");
```

Ainsi, il ne suffit (au niveau du code Java) que de deux étapes de maintenance pour que la prise en compte d'un nouveau type de document soit effective et fonctionnelle.

5.2. Transition vers une application WEB

Une transformation de notre application client/serveur en application WEB est envisageable à condition de **trouver un Framework WEB** adapté à la structure du projet comme Spring. Nos services doivent également prendre en compte des **requêtes HTTP**, substituant ainsi les échanges « ping-pong » de notre client BTTP.

Nous pensons que **Java EE** serait une solution possible car (d'après ce sur quoi nous nous sommes renseignés) la plateforme permet la fourniture de composants conçus pour le WEB tels que les Servlet ou les JSP (Java Server Pages). Les méthodes déjà codées ainsi que leur fonctionnement resteront tel quel. C'est le fonctionnement des services et des échanges client/serveur qui changeront.

5.3. Ajout d'un nouveau service

Si nous souhaitons ajouter un service au port 6000 à notre serveur, cela se fait simplement **en créant une nouvelle classe dans le package « services » qui doit hériter de « Service »** depuis la bibliothèque bserveur. Ayant normalisé cet aspect de notre application, il ne suffira plus qu'à créer un nouveau thread dans notre main qui ouvrira le service au port 6000 comme voulu, aspect restant tel quel pour chacun des services ajoutés.

```
new Thread(new Serveur(ServiceReservation.class, 1000)).start();
new Thread(new Serveur(ServiceEmprunt.class, 1001)).start();
new Thread(new Serveur(ServiceRetour.class, 1002)).start();
new Thread(new Serveur(ServiceNouveau.class, 6000)).start();
```

Une nouvelle configuration se connectant au port 6000 devra également être créée côté client.

6. Mise en place des BretteSoft

6.1. « Sitting Bull » : envoi de mails d'alerte

L'objectif explicite de ce BretteSoft est, pour chaque réservation échouée (document réservé ou emprunté) : **proposer à l'abonné d'envoyer un mail automatique** à une adresse unique lorsque le document sera disponible de nouveau (retourner ou réservation annulée).

Pour centraliser cette fonctionnalité, nouveau créé une classe « **MailAlert** » comportant un attribut et deux fonctions statiques :

- **mailsAlerts** : liste de documents qui ne sont pas disponibles et pour lesquels une alerte par mail a été demandée par un abonné.

- **addToAlertList** : prend en paramètre un document et l'ajoute à « **mailsAlerts** ».

- **sendMailAlert** : méthode utilisant les bibliothèques « javax.mail » et « activation » dans laquelle une connexion au service de Gmail avec le protocole SMTP (utilisé pour l'envoi de mail par Google) est établie. Cette méthode commence par définir la connexion en SMTP puis instancie un nouvel objet « Session » dans lequel nous renseignons l'adresse Gmail de la médiathèque ainsi que son mot de passe d'application (fonctionnalité de Google pour sécuriser un mot de passe). Le message est ensuite préparé en y indiquant l'objet, le texte puis le receveur et est envoyé au destinataire choisi.

Au niveau des services emprunt et réservation, une méthode « **proceedResponse** » a été créée et est appelée si le document n'est pas disponible. Il propose directement à l'abonné de mettre une alerte, celui-ci devant répondre par Oui (O) ou Non (N). S'il accepte, le document est ajouter à la liste présentée précédemment. L'envoi de mail se fera au niveau de la méthode « retour » qui, pour chaque document de la liste « mailsAlerts » invoquera la méthode « sendMailAlert ».

6.2. « Geronimo » : système de bannissement

Nous voulons ici **mettre en place un système bannissant les abonnés** qui ont plus de deux semaines de retard après un emprunt ou retournant un document endommagé.

Pour commencer, nous avons d'abord dû **ajouter une nouvelle méthode à l'interface document (autorisé pour certains BretteSoft d'après le sujet).** Celle-ci retourne la date limite de retour d'un document que l'on aura fixé dans la méthode « emprunt » dans un attribut de type LocalDateTime nommé « **dateRetour** ». Il a été décidé de laisser deux semaines à l'abonné avant de devoir rendre le document et que, deux semaines après cette date butoir, celui-ci soit banni pendant un mois. Un booléen « **banned** » a également été ajouté à la classe « Abonne » avec un setteur synchronisé.

Nous avons dû mettre en place deux tâches planifiée « **BanClient** » et « **UnbanClient** » permettant respectivement de bannir et débannir l'abonné en modifiant son attribut « banned ». Elles ont été planifiées dans « TimerHandler » et deux ConcurrentHashMap ont été créées :

- « **activeBorrows** » qui contient tous les documents nouvellement emprunté avec un Timer qui, deux semaines après la date de retour, enclenche la tâche « BanClient » qui va bannir l'abonné puis ranger celui-ci dans une seconde liste « banList ».

- « **banList** » contient tous les abonnés bannis avec un Timer d'un mois qui, après cette période, enclenchera la tâche « UnbanClient ».

Lorsqu'un abonné banni essayera d'accéder à nouveau aux services d'emprunt ou de réservation, la méthode « **checkAbonne** » lui indiquera sa date et son heure de débannissement.

Pour prendre en compte la dégradation d'un document, nous avons mis en place un petit utilitaire (de la même forme que pour l'alerte par mail) qui s'affiche à chaque rendu de document demandant si des dégradations ont été constatées (nous pouvons imaginer que c'est un hôte de la médiathèque qui procède au rendu une fois le document récupéré). Si la réponse est positive, le client est banni par la méthode « **addToBanList** » est sera banni de la même manière que pour les retards.

7. Conclusion

La réalisation de ce projet à été, pour nous deux, un excellent moyen de mettre en œuvre les concepts de multithreading, de gestion de la concurrence mais également d'exploitation d'une BD en Java. Le principe d'ajouter des certifications qui sont une plus-value à notre travail d'origine pour approfondir celui-ci nous a particulièrement plu. Néanmoins, nous aurions voulu améliorer la maintenabilité en n'ayant pas besoin de modifier la classe DataHandler pour ajouter un document mais nous n'avons pas trouvé de solution viable. Le BretteSoft restant nous a également beaucoup intéressé mais nous ne sommes pas arriver à le mettre en place malgré nos multiples essais.