Dauphine | PSL
UNIVERSITÉ PARIS

RAPPORT DU PROJET C++
MIAGE IF 2023-2024

Julien HADERER
Ha Anh TRAN
Amal LAMGHARI EL KOSSORI

Dauphine | PSL
UNIVERSITÉ PARIS

Sommaire

Justification des choix

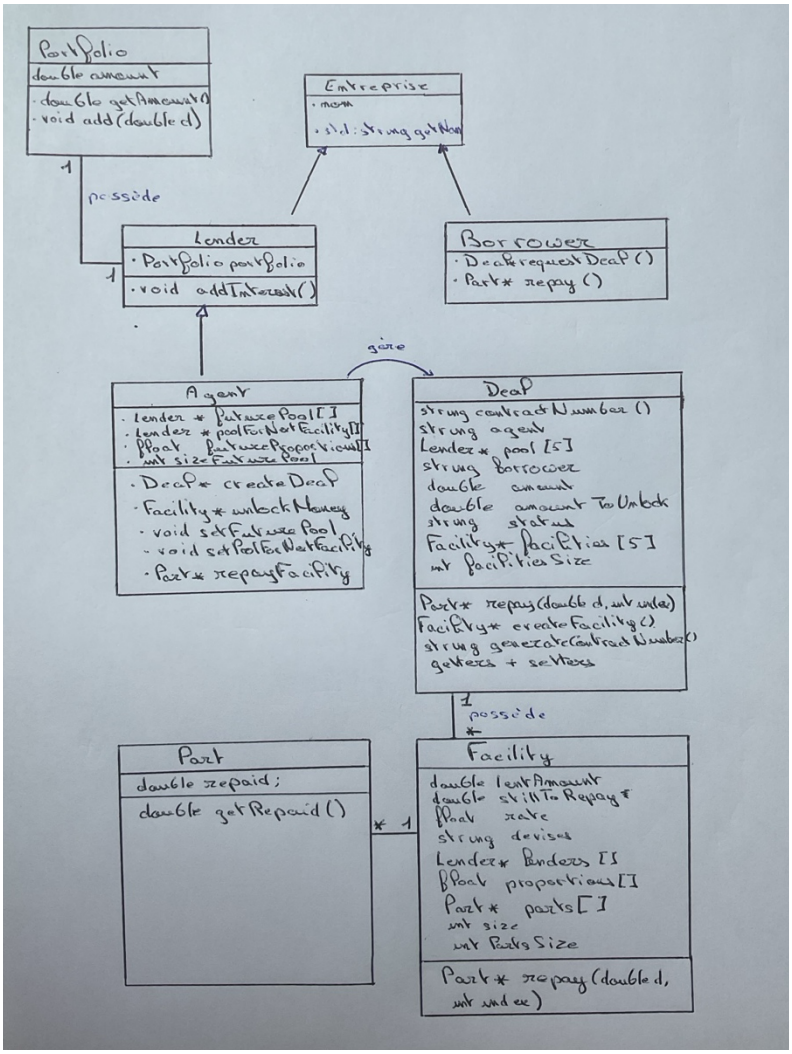
Au cours de ce projet, nous avons essayé autant que faire se peut d'utiliser les concepts présentés durant le cours. Nous avons notamment mis en pratique 2 pilliers de la programmtion orienté objet.

1. **Héritage et polymorphisme**

   - Nous avons représenté chaque entité du système sous forme de classes avec des attributs spécifiques, en fonction des caractéristiques de chaque entité.
   - Nous avons observé que les prêteurs (Lenders), l'agent et l'emprunteur (Borrower) sont tous des entreprises. Nous avons donc décidé d'utiliser l'héritage en créant une classe de base Entreprise, dont chaque entité (prêteurs, agent, emprunteur) hérite. D'une part, cela permet de regrouper les attributs communs dans la classe Entreprise, et d'autre part, chaque entité ayant des caractéristiques spécifiques, nous avons implémenté des méthodes dédiées dans leurs classes dérivées respectives.

2. **Encapsulation :**
   - L'encapsulation a été utilisée pour protéger les données sensibles et restreindre l'accès direct aux attributs des classes. Par exemple, les attributs de Deal, Lender, Borrower, Facility et Portfolio sont privés, et l'accès se fait par des méthodes publiques. Cela garantit l'intégrité des données et facilite la maintenance du code.

Architecture



Le diagramme de classe ci-dessus représente l'architecture de notre application. Nous n'y avons pas fait figurer tous les détails dans un souci de lisibilité (les setters et getters ont été ignorés ainsi que les constructeurs).

Comme expliqué dans la partie « Justification des choix », les lenders et borrowers héritent de la classe Entreprise (de laquelle elle tire leur nom). Cela pour éviter la redondance du nom, de ses getters et de ses setters dans les autrees classes.

De même, la classe Agent hérite de la classe Lender. Cela car, un agent prête de l'argent mais doit être en capacité de réaliser d'autres opérations (création de deal, gestion de remboursement, …).

L'autre choix d'architecture important concerne les deals, facilities et parts. En effet, nous avons fait en sorte qu'un deal possède une liste de de facilities. Et une facility contient une liste de part. En faisant cela, on peut accéder via un deal à tous ses détails, sans forcément créer une variable pour chaque tranche ou part en rapport. Cela simplifie grandement la manipulation.

Fonctionnalités

Au cours de ce projet, nous nous sommes attachés à suivre le déroulé présenté dans le sujet. En effet, l'enchaînement d'action qui y était présentée, était logique et nous a permis d'avancer pas à pas :

1. Tout d'abord, nous avons tâché de créer des lenders (et agent) ainsi que des borrowers.
2. Puis, nous avons fait en sorte qu'un borrower puisse créer un deal (en appelant l'agent qui est « responsable » de la création et de la gestion de celui-ci).

```
int sizeD1 = 3;
Lender* lendersD1[3] = { [0]: l1, [1]: l2, [2]: l3};
a->setFuturePool( lenders: lendersD1, size: sizeD1);
Deal* d1 = b->requestDeal(a, amount: 100, start: 0, end: 3);
```

3. Ensuite, nous avons implémenté la fonctionnalité permettant à l'agent de débloquer de l'argent avec des lenders. Dans les faits, l'agent doit commencer par déterminer les banques participant à la facility ainsi que leur proportion de contribution. Après cela, l'agent peut débloquer un certain montant (tant que celui-ci est inférieur ou égal au montant restant à débloquer). Cette action créant une facility.

```
Lender* lendersD1F3[2] = { [0]: l2, [1]: l3};
float proportionsD1F3[2] = { [0]: 0.6, [1]: 0.3};
a->setPoolForNextFacility( lenders: lendersD1F2, proportions: proportionsD1F3, size: 2);
a->UnlockMoney( d: d1, amount: 80, devises: "USD - EUR", startDate: 1, endDate: 3, rate: 0.1);
```

4. Pour terminer, nous nous sommes penchés sur le remboursement d'une facility. Dans ce cas, le borrower rembourse un montant lié à une facility à l'agent qui va alors partager cet argent entre chaque prêteur (au prorata de leur contribution). Là encore, il n'est pas possible de rembourser un montant supérieur à la somme restant à rembourser.

Difficultés rencontrées

1. Fonctionnelles

- Tout d'abord, il a fallu du temps pour bien comprendre comment fonctionne un financement structuré. Cela inclut la façon dont les prêts sont accordés, comment les montants sont débloqués en tranches (Facilities), et comment les paiements d'intérêts et les remboursements sont distribués aux prêteurs (Lenders).
- Un des principaux défis était de définir clairement les formules de calcul des intérêts et des montants remboursés. Les intérêts doivent être calculés en fonction du montant prêté, du taux d'intérêt et de la période. Pour simplifier, nous avons avons fixé le montant total des intérêts à rembourser pour une facility à la création de celle-ci. Précisément, le calcul est le suivant : montant débloqué * (1+taux d'intérêt) ^ durée de la facility.

```
this->stillToRepay = pow( x: 1+rate,  y: (end-start)) * lentAmount;
```

- Les remboursements sont ensuite répartis proportionnellement entre les prêteurs en fonction de leur contribution initiale à la tranche. Pour faire cela, lorsque l'on choisit les instituts bancaires participant au déblocage, on doit également préciser leur contribution proportionnelement au montant (dans la méthode setPoolForNextFacility via deux listes distinctes – une liste de lender et une liste de float pour la proportion).
- La définition des caractéristiques d'un Facility (tranche) a également été un point clé. Chaque tranche devait inclure des informations telles que le montant, le taux d'intérêt, la période, et les prêteurs participants. Il était crucial de s'assurer que toutes ces caractéristiques soient correctement définies et intégrées dans le modèle de calcul.
- Ensuite, nous avons dû nous pencher sur la question du remboursement. Le portfolio de chaque lender prend en compte uniquement les intérêts perçus. Ainsi, nous avons d'abord cherché à décomposer toute somme remboursée par le borrower. Nous voulions déterminer quelle part de cette somme devait être alloué au remboursement du prêt et quelle part servait à payer les intérêts. Pour ce faire, et suite à notre calcul d'intérêts nous avons simplement déterminé la proportion du montant global à rembourser qui concerne le remboursement et celle qui concerne les intérêts. Tout montant remboursé par le borrower a été décomposé par selon cette proportion. Puis, le montant alloué aux intérêts est redistribué à chaque lender au prorata de sa contribution au déblocage.

2. Techniques

2.1 Problème de Référence Croisée

- Lors de la conception du système, nous avons rapidement constaté que si une classe A fait référence à une classe B, il n'est pas possible de faire référence à la classe A dans la classe B. Cette situation de référence croisée n'est pas réciproque et peut conduire à des problèmes de compilation et de dépendances circulaires.

- Dans notre projet, cela s'est manifesté lorsque nous avons tenté de faire référence aux objets Facility dans la classe Deal et inversement. Il était nécessaire de décider si nous devions inclure un tableau de Facility dans la classe Deal ou déclarer un attribut Deal dans chaque Facility.

- Dans un premier temps, nous avons opté pour la seconde option. Celle avait pour avantage de simplifier l'implémentation des deux classes. On avait simplement à indiquer la référence du deal. D'un autre côté, la gestion des deals et des facility une fois créé n'était pas évidente. On devait garder en tête toutes les facilities et les deals créés. Et les nommer de façon compréhensible relevait de l'exploit. C'est pourquoi, après avoir fait fonctionner le code, nous avons mené un refactoring. Celui-ci visait à créer une liste de facilities dans un deal, et une liste de part dans une facilities. La difficulté est alors de maintenir ces listes. En revanche, dans le

main, nous n'avons plus qu'à stocker les deals. Le reste étant accessibles via un deal. Cela a rendu le code plus lisible mais aussi plus « professionnel ».

2.2 Lien entre fichier .h et .cpp

Afin de rendre notre travail le plus clair possible, nous avons pris soin de travailler avec des fichiers .h en plus des .cpp. D'un côté, cela représente une difficulté car lorsque l'on implémente une méthode, il faut faire des allers retours vers le fichier .h pour obtenir le détail des attributs ou des méthodes pas encore codées. D'un autre côté, cela permet une lecture beaucoup plus fluide et propre des possibilités offertes par une classe. Si bien qu'à la fin du projet, nous sommes pleinement satisfait d'avoir pris le temps de passer par des fichiers .h.

Répartition du travail

Pour ce projet, nous avons travaillé de la façon suivante. Tout d'abord, nous nous sommes concertés afin de voir si tout le monde avait la même compréhension de ce qui était attendu. De même, il nous a fallu définir certaines règles comme celle relative au calcul des intérêts. Puis, nous avons chacun de notre côté commencer à réfléchir à une structure et à son implémentation. Finalement, nous sommes partis sur la structure présentée précédemment car c'est celle qui avait le plus avancé.