# DIOUF Samba Diarra ALASSAF Mohamed

# Rapport projet c++

Le but de notre projet est de modéliser et gérer diverses entités liées au processus de prêt financier, comme les emprunteurs, les prêteurs, les contrats de prêt (deals), les facilités de prêt, et les paiements.  
Pour ce faire nous avons créé dans nos systèmes différentes classes

**Date** : Cette classe permet de gérer les dates de notre système. Cette classe nous permet de gérer et de renseigner les dates de contrat sous le format dd/mm/yyyy.  
  
**Exchange** : Cette classe nous permet de gérer les différentes devises de notre système. Elle nous permet de faire les conversions de montant dans le cas où par exemple la devise de la part est différente de celui de la facility.

**Borrower** : Représente un emprunteur dans le système. Chaque emprunteur a un nom, une adresse, un numéro de téléphone et un secteur d'activité.

**Lender** : Représente un prêteur. Chaque prêteur a un nom, une adresse, un numéro de téléphone et un classement.

**Deal** : Représente un contrat de prêt. Chaque deal contient des informations sur l'emprunteur, le prêteur, le montant du prêt, la devise, la date de signature du contrat, la date de fin du contrat et le statut du contrat.

**Facility** : Représente une facilité de prêt au sein d'un contrat de prêt. Chaque facilité contient des informations sur la date de début, la date de fin, le montant et la devise.

**Part** : Représente un paiement dans une facilité. Chaque partie contient des informations sur la date du paiement, le montant du paiement et la devise.

**Portfolio :** Cette classe permet de stocker les intérêts par part de Lender.

**DataManager** : Sert de gestionnaire central pour toutes les autres classes, permettant la création, la récupération et la gestion des objets.

## **Héritage et polymorphisme**

Dans le cadre de ce projet, nous avons choisi d'utiliser le concept **d'héritage**. Nous avons créé une classe mère **Company**, et deux classes filles, **Borrower** et **Lender**, qui sont toutes deux des types de compagnies. Comme une entreprise a généralement des caractéristiques communes telles qu'un nom, une adresse, un numéro de téléphone, etc., nous avons décidé de généraliser ces attributs dans une classe **Company**. Les attributs spécifiques aux classes filles ont été placés dans leurs classes respectives, par exemple, l'attribut **interestRate** est spécifique à la classe **Lender**.

Nous avons également tenté de mettre en œuvre le concept de **polymorphisme**, en créant une méthode **displayInformation** dans la classe **Company** qui est redéfinie dans les classes filles **Borrower** et **Lender**. Cette décision a été motivée par le fait que chaque classe n'a pas nécessairement besoin d'afficher les mêmes informations.

De plus, nous avons utilisé le concept de **surcharge** de méthode au sein de la classe **DataManager**. Nous avons mis en place deux méthodes **getDeals**(): l'une sans paramètre, qui permet de récupérer tous les deals, et l'autre avec un paramètre **DealStatus**, qui permet de récupérer les deals ayant un statut spécifique.

Cependant, l'usage de l'héritage et du polymorphisme reste limité dans notre application. La nature du problème ne justifiait pas une large application de ces concepts. Chaque classe avait ses propres attributs et comportements distincts, il n'était donc pas nécessaire d'utiliser l'héritage pour éviter la duplication de code. De même, le polymorphisme n'a pas été largement utilisé car il n'y avait pas beaucoup de situations où différents types d'objets auraient dû être traités de manière interchangeable.  
  
**Encapsulation**

L'encapsulation a été un choix de conception essentiel dans notre projet. Nous l'avons largement utilisée pour garantir la sécurité des données des objets. Les données ne peuvent être modifiées que par les méthodes définies, ce qui assure l'intégrité des données tout au long de la vie de l'application.

Fonctionnalités apportéesInitialement, nous avons codé toutes les classes que nous avons précédemment présentées ainsi que les méthodes pour gérer leurs attributs. Parmi ces méthodes, celle qui s'est distinguée par sa complexité est celle qui calcule les intérêts lorsqu'un emprunteur rembourse une part.

Dans notre système, la classe Facility dispose d'un attribut sous forme de tableau pour stocker la liste des parts liées à cette facilité. L'ajout d'un remboursement implique donc la création d'une part et son ajout à la facilité. Lorsqu'une part est ajoutée à une facilité, plusieurs opérations sont effectuées.

Au début, le système vérifie si le montant du paiement de la part est inférieur ou égal au montant restant de la facilité. Si le montant est supérieur, le système affiche un message d'erreur et arrête le calcul des intérêts.

Si le paiement est acceptable, le système récupère le taux d'intérêt de l'agent lié à la facilité. Puis il calcule le montant des intérêts en multipliant le montant restant de la facilité par le taux d'intérêt journalier et le nombre de jours écoulés entre le dernier paiement et le paiement actuel.

Comme le montant du paiement peut être dans une devise différente de celle de la facilité, une conversion de devise est effectuée avant la mise à jour du montant restant de la facilité.

Ensuite, pour chaque prêteur participant à la facilité, le système calcule la part d'intérêts qui lui revient en fonction de son pourcentage de participation. Ce montant d'intérêt est ajouté à son total d'intérêts dans le portefeuille.

Enfin, la fonction checkStatus est appelée sur le deal pour vérifier si toutes les facilités ont été remboursées. Si c'est le cas, le statut du deal passe à "Terminated".

**Fonctionnalité dans le main**  
  
Nous avons également incorporé un menu interactif au sein de la classe principale pour permettre à l'utilisateur de créer, de rechercher et de consulter la liste de tous les éléments, tels que les deals, les emprunteurs (borrowers), les prêteurs (lenders), et ainsi de suite.

Prenons l'exemple de la création d'un deal. À chaque étape du processus, nous demandons à l'utilisateur de fournir les informations pertinentes au deal. Une fois le deal créé, si l'utilisateur le souhaite, il peut ajouter des facilités (facility) et des parts de la même manière.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Ceci est un autre exemple qui permet de voir la liste des lenders de notre application.

Une image contenant texte, capture d’écran

Description générée automatiquement

Cette interaction étend l'expérience utilisateur, tout en laissant la flexibilité de naviguer à travers différentes parties du programme selon leurs besoins.

Dans l'ensemble, ce menu interactif offre une approche pratique pour manipuler divers aspects de notre application financière, améliorant ainsi son accessibilité et son utilisation.

**Fonctionnalités manquantes**

En raison de contraintes de temps, certaines fonctionnalités n'ont pas pu être implémentées. Par exemple, le système ne prend pas encore en charge la suppression ou la modification d'entités.  
Il serait donc intéressant de coder ces fonctionnalités dans le futur.

**Difficulté**   
  
La première difficulté majeure que nous avons rencontrée au cours de ce projet était la compréhension du sujet. Plus précisément, la conceptualisation de la classe Portfolio et la gestion des intérêts se sont révélées délicates. Pour surmonter cet obstacle, nous avons adopté une approche systématique, en consacrant du temps à une étude approfondie du sujet. Cela impliquait la recherche d'informations complémentaires et la recherche d'exemples concrets de ces concepts dans le secteur financier. Cette démarche nous a permis de mieux saisir l'essence et l'application pratique de ces notions, facilitant ainsi leur implémentation dans notre système.

Outre la compréhension du sujet, d'autres difficultés techniques ont émergé tout au long du développement de ce projet. Une de ces difficultés était la gestion des pointeurs en C++. Bien que les pointeurs soient un outil puissant, leur utilisation incorrecte peut entraîner des erreurs de segmentation et des fuites de mémoire. Nous avons rencontré des problèmes dans l'allocation et la libération de la mémoire, mais nous avons réussi à les surmonter en utilisant judicieusement les pointeurs intelligents de C++, qui gèrent automatiquement la durée de vie de l'objet pointé.

Une autre difficulté technique était la gestion des inclusions de fichiers d'en-tête, en particulier lorsqu'il y avait des dépendances cycliques entre les classes. Par exemple, nous avons rencontré des problèmes lorsque la classe Deal avait besoin d'inclure le fichier d'en-tête de Facility et vice versa, créant une dépendance cyclique qui conduisait à des erreurs de compilation. Pour résoudre ce problème, nous avons utilisé des déclarations anticipées, qui permettent à une classe de faire référence à une autre classe sans avoir besoin de son fichier d'en-tête complet. Cette approche nous a permis de résoudre les dépendances cycliques et d'éviter les erreurs de compilation.

Au cours de ce projet, nous avons donc dû faire face à plusieurs défis techniques. Cependant, chacun d'eux a servi de catalyseur à notre apprentissage et à notre compréhension de la programmation en C++.

**Répartition du travail**

Le travail a été réparti également entre les deux membres du binôme, avec une collaboration constante pour s'assurer que les deux comprenaient toutes les parties du système.

## **Conclusion**

L'objectif principal du projet était de développer un système de gestion de prêts financiers en utilisant le langage de programmation C++. L'accent a été mis sur l'architecture orientée objet pour refléter le domaine du problème. Le projet a été un succès, bien qu'il y ait eu des difficultés. Dans l'ensemble, ce projet a été une excellente occasion d'appliquer et d'approfondir nos connaissances en C++.