FACULTÉ DES SCIENCES ET TECHNIQUES
DÉPARTEMENT INFORMATIQUE

RAPPORT DE PROJET Analyse et Programmation Orientées objets - C++

Smail OUABED — Majd MRIKA

2023 – 2024

Introduction Générale

Ce rapport a pour objectif de détailler la création d'un logiciel dédié à la gestion de caves à vin. Nous avons suivi une approche méthodique qui débute par une analyse approfondie du sujet. Ensuite, nous avons élaboré des diagrammes UML pour représenter nos concepts, avec une justification détaillée. Enfin, ces concepts ont été mis en œuvre dans un programme en langage C++.

L'ensemble du rapport se divise en deux phases distinctes :

- **Phase d'Analyse** : Cette première partie expose nos choix de conception à travers des diagrammes UML, les détaillant et justifiant chacun d'entre eux. L'objectif est de fournir une compréhension claire des fondements conceptuels du logiciel à développer.
- **Phase d'Implémentation en C++** : La seconde partie du rapport aborde la matérialisation concrète des concepts définis dans la phase d'analyse. Elle décrit comment les éléments du logiciel ont été traduits en code, mettant ainsi en œuvre les principes énoncés dans les diagrammes UML.

1. Diagramme de classe :

Le diagramme de classe est une représentation visuelle cruciale dans la conception de notre projet. Il détaille les relations entre trois classes principales : **Cave**, **Vin**, et **Fournisseur.** Chacune de ces classes a des attributs définis aussi évident, ce qui abstrait reste à le définir de la meilleure façon pour répondre à l'attente du cahier des charges du projet.

Le projet en question nous demande d'implémenter cela :

En dehors des relations qui sont claires et qui ne demande pas de réflexions, ils restent quelques exigences qui demandent un choix crucial pour leur implémentation ce choix va permettre au logiciel, d'être implémenter et de fonctionner mais surtout de permettre au futur d'avantages mises à jours et ajouts. Et finalement une maintenance facile. Ici dessous présente des exigences tirées du projet.

- **Gérer le nombre de vins en stock pour les caves ou il est proposé,**
- **Un même vin peut être fournit par plusieurs fournisseurs avec des prix de base différents.**
- **Pour chacun de ces fournisseurs nous avons également des réductions par nombre de bouteilles commandées.**

2. Choix d'implémentation :

2.1 Implémentation de stock :

Comme le stock est une relation qui concerne une cave et un vin, on a décidé d'implémenter le stock tout simplement comme un attribut pour la classe cave puisque y'a une relation directe entre la classe **Cave** et **Vin.** Ce stock est un std::map, l'utilisation de **map** est convenue car il répond au contraints d'un stock réel :

- **Pas de duplication c.à.d. on n'aura pas deux mêmes entités dans un stock et cela est facilement gérable car les std::map repose sur la sauvegarde de clé unique du coup si jamais on essaie d'insérer un même vin avec une quantité différente pour avoir ce même vin deux fois avec deux quantités différentes alors cette dernière valeur va tout simplement écraser l'ancienne.**
- **Lorsque on a un stock on ne s'intéresse pas qu'au vin stocké mais aussi à la quantité stockée de ces vins, et les map sont parfaits pour cela car il stock des valeurs paires <clé, valeur> ce qui nous permet la gestion de ces vins avec leurs quantités.**

2.2 Implémentation de la relation Fournitures entre cave et Fournisseur et vin :

-Pour cela on décider de faire une classe association nommée **Fournir** pour désigner la fourniture d'un vin de la part d'un fournisseur pour une cave et cela avec un prix de base, cela dû au fait a plusieurs choses :
D'abord on ne pouvait pas mettre un attribut prix pour la classe vin car il se peut que son prix de base soit différent d'un fournisseur a un autre.
-Si on mettait le prix de base comme attribut a la classe fonction ça sera possible mais par contre pour couvrir les autres aspects du projet fallait d'autres critères pour l'attribut clé et pour cela on a décidé de le faire en tant que classe association pour pouvoir répondre à la pluspart des exigences du cahier de charges.

2.3 Implémentation des réductions :

Contrairement à l'ajout d'un attribut "Réduction" à la classe fournisseur, nous avons choisi d'implémenter une classe séparée pour "Réduction". Cette décision permet une gestion plus souple et extensible des réductions offertes par les fournisseurs en fonction du nombre de bouteilles commandées, avec deux valeurs qui représentes un intervalle, une implémentation avec des conteneurs aurait était possible mais pas convenable **(difficulté de manipulation)**

Après une étude approfondie du projet, le diagramme UML final prend en compte ces quatre classes et leurs relations.

Après une étude profonde du projet on est arrivé à ce diagramme UML :

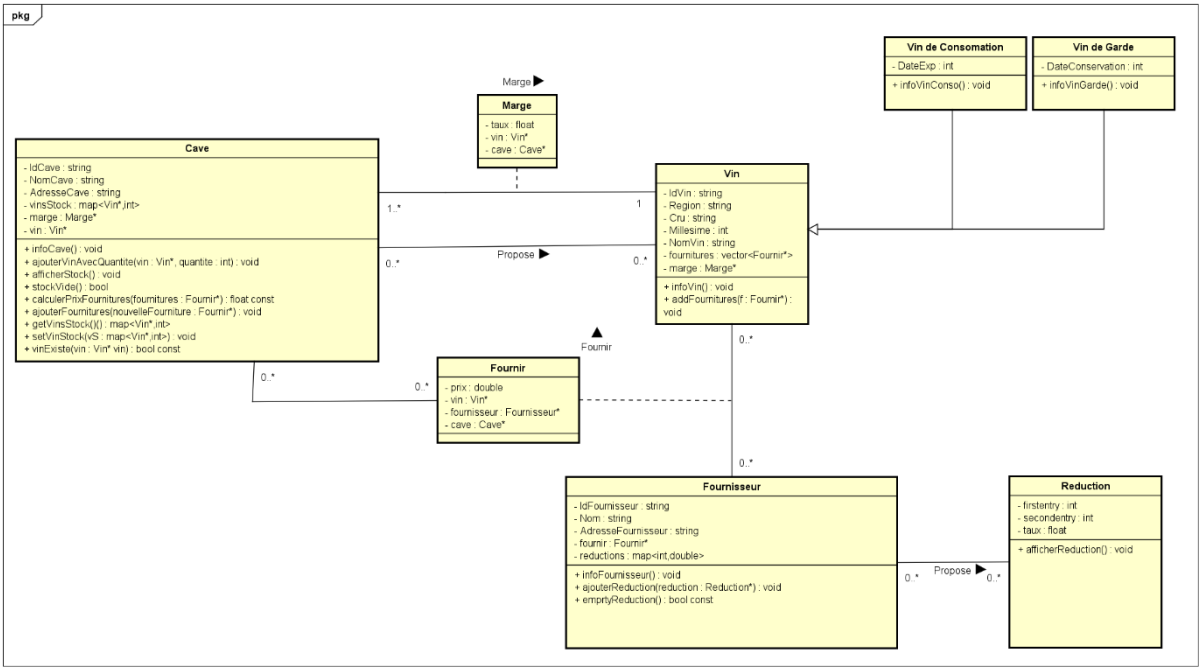

Figure 1 : Diagram de classe

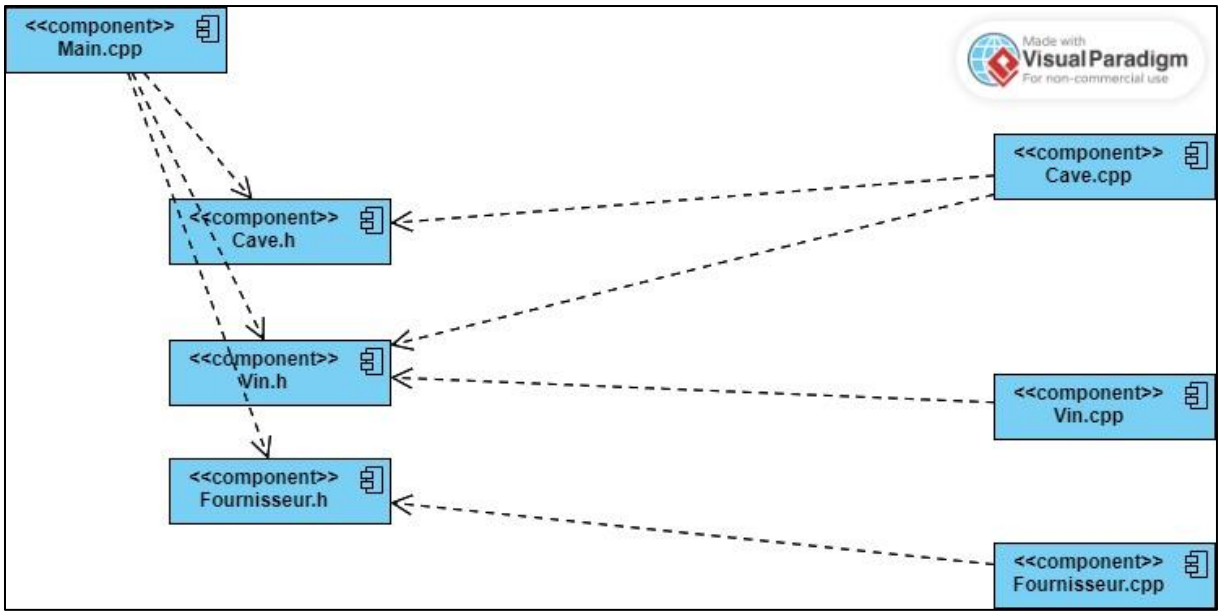

Figure 2 Diagramme de composant

3. Explication du choix des classes :

3.1 Classe "Cave" :

La classe "Cave" a été conçue pour représenter les caves à vin dans notre système. Elle est caractérisée par un identifiant unique, un nom et une adresse. Pour gérer le stock de vins dans une cave, nous avons utilisé un attribut de type std::map dans la classe, associant chaque vin à sa quantité en stock. La relation avec la classe "Marge" a été ajoutée pour refléter la marge appliquée par la cave sur les prix des vins. Ces choix sont étroitement alignés avec la logique de modélisation présentée dans le code de la classe "Cave".

3.2 Classe "Vin" :

La classe "Vin" a été élaborée pour encapsuler les différentes caractéristiques d'un vin dans notre système de gestion de caves. Chaque instance de cette classe est associée à un identifiant unique, une région de production, un cru, un nom, et un millésime. Ces attributs sont définis pour répondre aux exigences du projet et sont également liés aux fonctionnalités implémentées dans la classe.
Les vins étant sujets à des variations de prix en fonction des fournisseurs, nous avons introduit une relation avec la classe "Fournir" pour modéliser cette association. De plus, afin de gérer les réductions qui dépendent du fournisseur, nous avons décidé de créer une classe dédiée "Réduction" plutôt que de les considérer comme des attributs de la classe "Fournisseur". Ces choix sont en corrélation directe avec la conception du code de la classe "Vin".

3.3 Classe "Fournisseur" :

La classe "Fournisseur" représente les entités qui approvisionnent les caves en vins. Chaque fournisseur est défini par un identifiant unique, un nom, et une adresse. Pour représenter les réductions spécifiques offertes par un fournisseur en fonction du nombre de bouteilles commandées, nous avons créé une classe "Réduction".

4. Explication du choix conteneurs utilisés :

- map **pour Stocker les Vins dans la Cave** : La classe Cave utilise un map pour stocker les vins avec leurs quantités respectives, permettant une recherche rapide et l'éviction des doublons.
- vector **pour Stocker les Fournitures dans la Cave** : La classe Cave utilise un vector pour stocker les fournitures associées à la cave.
- map **pour stocker les réductions dans la classe Fournisseur.**
- vector **pour Stocker les Fournitures du Vin** : La classe Vin utilise un vector pour stocker les fournitures associées à ce vin.

5. Explication de certaines bibliothèques utilisées :

#include <iostream> : Cette bibliothèque est une bibliothèque standard de C++ qui offre des fonctionnalités pour les opérations d'entrée et de sortie. Dans notre code, on utilise cout et cin pour afficher des messages à l'utilisateur et obtenir des entrées de sa part. Par exemple, lors de la création d'une nouvelle cave, d'un nouveau vin, d'un nouveau fournisseur, etc., vous utilisez cin pour obtenir des entrées.

#include <vector> : L'en-tête <vector> fournit le conteneur std::vector, qui est un tableau dynamique pouvant augmenter ou diminuer de taille. Dans notre code, on les utilise pour stocker des pointeurs vers des objets représentant des caves, des vins, des fournisseurs, des marges, etc. Les vecteurs offrent une taille dynamique, un accès facile aux éléments et une gestion automatique de la mémoire.

#include <algorithm> : Cet en-tête fournit une collection de fonctions pour effectuer diverses opérations sur des plages d'éléments. Dans notre code, on utilsie l'algorithme std::find_if de cet en-tête. La fonction find_if permet de rechercher un élément qui satisfait une condition spécifiée. Dans notre cas, elle est utilisée pour rechercher une cave avec un ID spécifique et d'autres opérations similaires.

#include <list> : Cette bibliothèque offre le conteneur std::list, une liste chaînée double permettant des opérations efficaces d'insertion et de suppression, sans invalider les itérateurs. Utile lorsque des modifications fréquentes de la structure des données sont nécessaires.

#include <map> : Intégrant le conteneur std::map, cette bibliothèque propose une structure de données associative triée par clé. Dans votre code, elle est utilisée pour gérer le **stock de vins** dans la classe **Cave**, associant des pointeurs vers des objets **Vin** à des quantités. Cela facilite la recherche, l'ajout et la suppression ordonnée des vins dans le stock de la cave.

6. Explication de certaines fonctions utilisées :

- **AfficherMenu() :**Cette fonction affiche simplement le menu du programme. Elle imprime différentes options disponibles, comme la création d'une cave, d'un vin, d'un fournisseur, etc. Elle est utilisée pour présenter les choix disponibles à l'utilisateur.

```
void afficherMenu() {
    cout << "===== Menu =====" << endl;
    cout << "1. Creer une cave" << endl;
    cout << "2. Creer un vin" << endl;
    cout << "3. Creer un fournisseur" << endl;
    cout << "4. Creer une Fournir" << endl;
    cout << "5. Creer une marge" << endl;
    cout << "6. Creer les reductions d'un fournisseur" << endl;
    cout << "7. Afficher le stock d'une cave" << endl;
    cout << "8. Getion de stock (ajout de vins dans une cave) "<<endl;
    cout << "9. Trouver la cave ou un vin est propose au meilleur prix" << endl;
    cout << "10. Editer une cave "<<endl;
    cout << "12. Quitter" << endl;
    cout << "==L'edit des autres elements va venir lors d'une nouvelle mise a jour !!=="<<endl;
}
```

- **Les fonctions de création :** Les méthodes de création du programme regroupent des fonctionnalités clés pour permettre à l'utilisateur d'ajouter de nouvelles entités à la gestion de caves à vin. La fonction **creerCave**() assure la création d'une nouvelle cave en vérifiant l'unicité de son identifiant, tandis que **creerVin**() permet la création d'un nouveau vin avec des caractéristiques spécifiques, en fonction du type choisi. La méthode **creerFournisseur**() offre la possibilité de créer un nouveau fournisseur avec un identifiant un nom et une adresse. La fonction **creerFournir**() établit une relation de fourniture entre un vin, un fournisseur et une cave, nécessitant des choix de la part de l'utilisateur. Enfin **creerMarge**() permet d'établir une marge entre un vin et une cave, contribuant ainsi à la personnalisation du système. Ces méthodes offrent à l'utilisateur une interface conviviale pour enrichir la base de données du programme en fonction de ses besoins spécifiques renforçant ainsi la flexibilité et l'adaptabilité du système de gestion de caves à vin. **creeReduction()** : Gère la création de réductions pour un fournisseur spécifique. L'utilisateur sélectionne un fournisseur existant, voit les réductions actuelles, et peut ajouter de nouvelles réductions en spécifiant deux entrées et un taux de réduction.

- **afficherStock()** : Affiche le stock d'une cave sélectionnée par l'utilisateur. Si la cave est vide, elle informe l'utilisateur de cette condition.

```cpp
void afficherStock() {
    if (!caves.empty()) {
        cout << "Choisissez une cave : " << endl;
        for (size_t i = 0; i < caves.size(); ++i) {
            cout << i + 1 << ". " << caves[i]->getNomCave() << endl;
        }
        int choixCave;
        cout << "Votre choix : ";
        cin >> choixCave;
        if (choixCave > 0 && choixCave <= static_cast<int>(caves.size())) {
            if (caves[choixCave - 1]->stockVide()) {
                cout << "Le stock de la cave est vide pensez a ajoutez du vin au stock d'abord." << endl;
            } else {
                caves[choixCave - 1]->afficherStock();
            }
        } else {
            cout << "Choix de cave invalide." << endl;
        }
    } else {
        cout << "Aucune cave n'a ete creee." << endl;
    }
}
```

Cave::afficherStock() : Affiche le stock de la cave en parcourant les vins présents dans le stock. Pour chaque vin, elle affiche la quantité en stock, le fournisseur associé, le prix final après réductions et la marge.

```cpp
void Cave::afficherStock() {
    cout << "Stock dans la cave " << NomCave << ": " << endl;
    for (const auto& entry : pair<...> const & : vinsStock) {
        Vin* vin = entry.first;
        int quantite = entry.second;
        cout << "Vin: " << vin->getNomVin() << " | Quantite en stock: " << quantite;
        Marge* pMarge = vin->getMarge();
        Fournir *fourniture = this->getFourniture( nomVin: vin->getNomVin(),  millesime: vin->getMillesime());
        if (fourniture != nullptr) {
            Fournisseur *fournisseur = fourniture->getFournisseur();
            // Recherche de la reduction applicable à la quantite actuelle
            float tauxReduction = 0.0;
            for (Reduction *reduction: fournisseur->getReductions()) {
                if (quantite >= reduction->getFirstentry() && quantite <= reduction->getSecondentry()) {
                    tauxReduction = reduction->getTaux();
                    break;
                }
            }
            // Calcul du prix
            float prixFourniture = fourniture->getPrix();
            float prixApresReduction = prixFourniture - (prixFourniture * tauxReduction / 100);
            float prixFinal = prixApresReduction + (prixApresReduction * pMarge->getTaux() / 100);
            cout << "| Fournisseur: " << fournisseur->getNomFournisseur();
            cout << " au Prix de : " << prixFinal << "$ TTC" << endl;
        } else {
            cout << " | Fourniture introuvable" << endl;
        }
    }
}
```

- **ajouterVinaCave()** : Permet à l'utilisateur d'ajouter un vin existant à une cave spécifiée, en indiquant la quantité.

```
void ajouterVinaCave() {
    // Verifier si des caves et des vins existent
    if (caves.empty() || vins.empty()) {
        cout << "Veuillez creer au moins une cave et un vin avant d'ajouter un vin a une cave." << endl;
        return;
    }
    // Afficher la liste des caves
    cout << "Choisissez une cave : " << endl;
    for (size_t i = 0; i < caves.size(); ++i) {
        cout << i + 1 << ". " << caves[i]->getNomCave() << endl;
    }
    int choixCave;
    cout << "Votre choix : ";
    cin >> choixCave;
    // Verifier la validite du choix de la cave
    if (choixCave > 0 && choixCave <= static_cast<int>(caves.size())) {
        // Afficher la liste des vins
        cout << "Choisissez un vin : " << endl;
        for (size_t i = 0; i < vins.size(); ++i) {
            cout << i + 1 << ". " << vins[i]->getNomVin() << endl;
        }
        int choixVin;
        cout << "Votre choix : ";
        cin >> choixVin;
        // Verifier la validite du choix du vin
        if (choixVin > 0 && choixVin <= static_cast<int>(vins.size())) {
            // Entrer la quantite
            int quantite;
            cout << "Entrez la quantite : ";
            cin >> quantite;
```

- **trouverCaveMeilleurPrix()** : Recherche et identifie la cave proposant un vin au meilleur prix, en fonction du nom du vin et de son millésime.

```
void trouverCaveMeilleurPrix() {
    //affichage de la lite des vins
    cout<<"Liste des vins disponibles :"<<endl;
    for(int i=0; i<vins.size(); i++){
        cout<<i+1<<". "<<vins[i]->getNomVin()<<"("<<vins[i]->getMillesime()<<")2"<<endl;
    }
    // Demander à l'utilisateur le nom du vin et son millesime
    string nomVin;
    int millesime;
    cout << "Entrez le nom du vin : ";
    cin >> nomVin;
    cout << "Entrez le millesime du vin : ";
    cin >> millesime;
```

```cpp
void trouverCaveMeilleurPrix() {
    //affichage de la lite des vins
    cout<<"Liste des vins disponibles :"<<endl;
    for(int i=0; i<vins.size(); i++){
        cout<<i+1<<". "<<vins[i]->getNomVin()<<"("<<vins[i]->getMillesime()<<")2"<<endl;
    }
    // Demander à l'utilisateur le nom du vin et son millesime
    string nomVin;
    int millesime;
    cout << "Entrez le nom du vin : ";
    cin >> nomVin;
    cout << "Entrez le millesime du vin : ";
    cin >> millesime;

    // Recherche du vin dans le stock de toutes les caves
    Fournir* fournitureTrouvee = nullptr;
    Cave* caveMeilleurPrix = nullptr;
    float prixMin = MAX_Val; // Initialiser le prix minimum à la valeur maximale possible
    for (Cave* cave : caves) {
        // Vérifier si la cave a des stocks
        if (!cave->stockVide()) {
            Fournir* fournitureDansCave = cave->getFourniture(nomVin, millesime);
            if (fournitureDansCave != nullptr) {
                float prixFourniture = cave->calculerPrixFourniture( fourniture: fournitureDansCave);
                // Mettre à jour la fournitureTrouvee si le prix actuel est inferieur au prix minimum enregistre
                if (prixFourniture < prixMin) {
                    prixMin = prixFourniture;
                    fournitureTrouvee = fournitureDansCave;
                    caveMeilleurPrix = cave;
                }
```

Resulat :

```
Stock dans la cave CAVE1:
Vin: VIN1 | Quantite en stock: 20| Fournisseur: FOUR1 au Prix de : 19.8$ TTC
===== Menu =====
1. Creer une cave
2. Creer un vin
3. Creer un fournisseur
4. Creer une Fournir
5. Creer une marge
6. Creer les reductions d'un fournisseur
7. Afficher le stock d'une cave
8. Getion de stock (ajout de vins dans une cave)
9. Trouver la cave ou un vin est propose au meilleur prix
10. Editer une cave
12. Quitter
==L'edit des autres elements va venir lors d'une nouvelle mise a jour !!==
Entrez votre choix : 7
Choisissez une cave :
1. CAVE1
2. CAVE2
Votre choix : 2
Stock dans la cave CAVE2:
Vin: VIN1 | Quantite en stock: 20| Fournisseur: FOUR1 au Prix de : 14.85$ TTC
===== Menu =====
1. Creer une cave
2. Creer un vin
3. Creer un fournisseur
4. Creer une Fournir
5. Creer une marge
6. Creer les reductions d'un fournisseur
7. Afficher le stock d'une cave
8. Getion de stock (ajout de vins dans une cave)
9. Trouver la cave ou un vin est propose au meilleur prix
10. Editer une cave
12. Quitter
==L'edit des autres elements va venir lors d'une nouvelle mise a jour !!==
Entrez votre choix : 9
Liste des vins disponibles :
1. VIN1(2022)2
Entrez le nom du vin : VIN1
Entrez le millesime du vin : 2022
La cave proposant le vin au meilleur prix est : CAVE2 au prix de : 14.85$
```

- ***calculerPrixFourniture()*:** Calcule le prix d'une fourniture en prenant en compte les réductions du fournisseur et la marge de la cave.

```
float Cave::calculerPrixFourniture(Fournir* fourniture) const {
    Vin* vin = fourniture->getVin();
    int quantite = vinsStock.at( k: vin);
    Fournisseur* fournisseur = fourniture->getFournisseur();
    // Recherche de la reduction applicable à la quantite actuelle
    float tauxReduction = 0.0;
    for (Reduction* reduction : fournisseur->getReductions()) {
        if(! fournisseur->emptyReductions()){
            if (quantite >= reduction->getFirstentry() && quantite <= reduction->getSecondentry()) {
                tauxReduction = reduction->getTaux();
                break;
            }
        }else{
            cout<<"Il ne y a pas de reduction "<<endl;
        }
    }
    // Calcul du prix
    float prixFourniture = fourniture->getPrix();
    float prixApresReduction = prixFourniture - (prixFourniture * tauxReduction / 100);
    float prixFinal = prixApresReduction + (prixApresReduction * marge->getTaux() / 100);

    return prixFinal;
}
```

- **editCave() :** Permet à l'utilisateur de modifier les informations d'une cave existante en saisissant un nouvel identifiant, un nouveau nom et une nouvelle adresse.

```
void editCave() {
    if (caves.empty()) {
        cout << "Aucune cave n'a ete creee." << endl;
        return;
    }
    cout << "Liste des caves disponibles :" << endl;
    for (Cave* cave : caves) {
        cout << "ID: " << cave->getIdCave() << ", Nom: " << cave->getNomCave()<<", Adresse: "<<cave->getAdresseCave()<< endl;
    }
    string IdCave;
    cout << "Entrez l'ID de la cave que vous souhaitez modifier : ";
    cin >> IdCave;
    // Recherche de la cave avec l'ID donne dans la liste des caves
    auto it :iterator<...> = find_if( first: caves.begin(),  last: caves.end(),  pred: [IdCave](Cave *cave) -> bool {...});
    if (it != caves.end()) {
        // La cave a ete trouvee
        cout << "Cave trouvee. Entrez les nouvelles informations :" << endl;
        string nouveauNom, nouvelleAdresse;
        cout << "Nouveau nom de la cave : ";
        cin >> nouveauNom;
        cout << "Nouvelle adresse de la cave : ";
        cin >> nouvelleAdresse;
        // Mettez à jour les informations de la cave
        (*it)->setNomCave( nc: nouveauNom);
        (*it)->setAdresseCave( ac: nouvelleAdresse);
        cout << "La cave a ete modifiee avec succes." << endl;
    } else {
        cout << "Aucune cave trouvee avec l'ID fourni." << endl;
    }
```