TP2 Programmation Orientée à Objet

Université du Québec à Chicoutimi

Département d'Informatique

Travail Pratique 2 : Gestion de Garage

Description

Concevoir et développer un ensemble de classes pour gérer les factures de une garage mécanique en C++, en intégrant dès le départ les concepts d'héritage, association, composition, de polymorphisme, de classes abstraites, délegation et de traitement des exceptions.

Objectifs

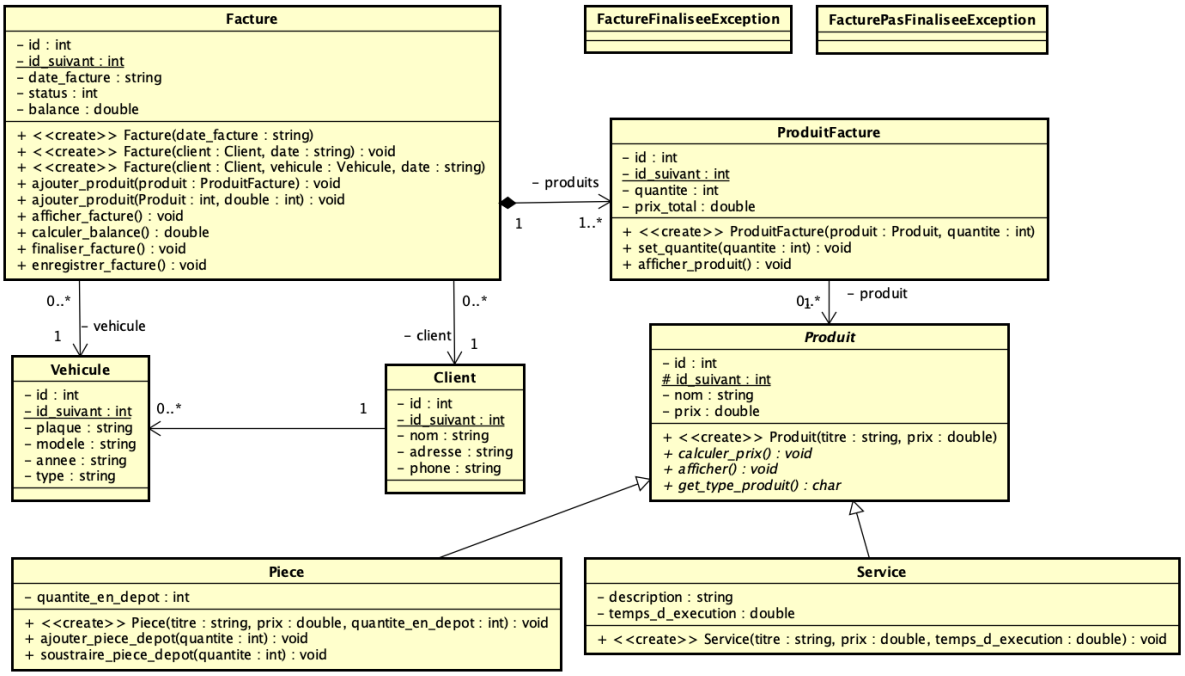
- Appliquer l'héritage et le polymorphisme pour étendre et diversifier les fonctionnalités des classes en C++.
- Concevoir et implémenter des classes abstraites pour structurer des modèles de comportement dans le code.
- Développer une compréhension des associations et compositions pour créer des relations complexes entre objets.
- Intégrer la délégation de comportement.
- Appliquer le traitement des exceptions pour améliorer le code.

Observations

1. **Travail de Groupe :** Le travail peut être réalisé en groupe de 6 personnes maximum. Les discussions entre groupes sont encouragées, mais la mise en œuvre doit être faite séparément.
2. **Format de Livraison :** Le travail doit être soumis en format numérique via Moodle. Envoyez un seul fichier .zip contenant tous les fichiers nécessaires (code source (.h, .hpp, .cpp, exécutables, et documentation). Nommez le fichier .zip avec le nombre du groupe.
3. **Intégrité Académique :** Les travaux copiés recevront une note de zéro. Les programmes qui ne compilent pas ne seront pas évalués et recevront également zéro. Les normes de bonne programmation doivent être suivies.

Instructions :

Implémentez un ensemble de classes pour un système de gestion de garage mécanique. Le garage doit maintenir des informations sur ses clients, pièces, services et factures. À cette fin, la conception suivante a été réalisée (Diagramme de Classes) :



Classe Produit

- abstraite
- **méthode calculer_prix() :** pour les pièces, elle retourne la valeur de leur propre prix,
- **méthodes abstraites :**
    - calculer_prix();
    - afficher();
    - get_type_produit().

Classe Pièce

- **méthode ajouter_piece_depot** : elle augmente la quantité en dépôt de la valeur indiquée par le paramètre quantite.
- **méthode soustraire_piece_depot :**
    - Si possible, elle diminue la quantité en dépôt de la valeur indiquée par le paramètre quantite.
    - Sinon, elle lance une erreur de temps d'exécution avec le texte: "Quantité insuffisante en dépôt".
- **méthode calculer_prix()** : pour les pièces, elle retourne la valeur de leur propre prix
- **sortie de la méthode afficher :**
    - `Pièce : nom de la pièce, $ 10, En Dépôt : 10`

Classe Service

- **méthode calculer_prix()** :
    - Pour les services, elle retourne la valeur du prix du service (qui est donné par heure) multipliée par le temps d'exécution (également établi en heures).
- **sortie de la méthode afficher :**
    - `Service : nom du service, $ 20, Exécution : 3h`

Classe ProduitFacture

- **Dans le constructeur,** la valeur de prix_total doit être calculée en utilisant la quantité multipliée par le retour de la méthode calculer_prix de l'objet produit.
- **si quantite est modifié,** la valeur de prix_total doit être recalculée.
- **sortie de la méthode afficher_produit :**
  - `Qté : 5, Produit/Service : Pièce (P), Unitaire : 10, Total : 50`
- **surcharger l'opérateur << pour retourner :**
  - `5 | Pièce   (P)| 10 | 50 (quand c'est une Pièce)`
  - `5 | Service (S)| 10 | 50 (quand c'est un Service)`

Classe Facture

- **ajouter_produit(ProduitFacture) :**
  - ajoute un objet ProduitFacture à la liste ;
  - **si la facture est finalisée,** ça veut dire, **le statut est égal à 1**, elle lance l'exception **FactureFinaliseeException;**
- **ajouter_produit(Produit, int) :**
  - crée un objet ProduitFacture avec la quantité fournie et l'ajoute ensuite à la liste produits ;
  - **si la facture est finalisée,** ça veut dire, **le statut est égal à 1**, elle lance l'exception **FactureFinaliseeException;**
- **calculer_balance :** calcule la somme de tous les produits de la facture ;
- **finaliser_facture :**
  - **si la facture est finalisée,** ça veut dire, le statut est égal à 1, elle lance l'exception **FactureFinaliseeException** ;
  -
  - **sinon**:
    - modifie la valeur du statut à 1 ;
    - et pour chaque Pièce de la facture, met à jour la quantité en soustrayant la quantité qui est vendue.
- **enregistrer_facture :**
  - **si le statut est égal à 0**, elle lance l'exception **FacturePasFinaliseeException** ;
  - sauvegarde un fichier txt avec les données de la Facture telles que présentées dans la méthode **afficher_facture**;
  - le nom du fichier doit être *facture-n.txt* ; où n est l'ID de la facture.
- **sortie de la méthode afficher_facture :**

```
------------------------------------------------------------
                      FACTURE
------------------------------------------------------------
Client : Eduardo
Adresse : Rue X
Phone : 666 666-66-66
Vehicule : Honda| Plaque: JHJ 233
------------------------------------------------------------
Qte       | Produit / Service       | Unitaire       | Total
6         | Piece                (P)| 10             | 60
5         | Service              (S)| 20             | 300
------------------------------------------------------------
                                          Balance| 360
```

Classe FactureFinaliseeException

- Elle lance un error de temps d'éxecution avec la message: "La facture a déjà été finalisée"

Classe FacturePasFinaliseeException

- Elle lance un error de temps d'éxecution avec la message: "La facture doit être finalisée avant le registre"

Pour toutes les classes

- méthodes getters et setters ;
- l'id est initialisé dans le constructeur à partir de la valeur de la variable id_suivant, qui doit être incrémentée à chaque utilisation.

Fonction main

Une fonction principale doit être fournie contenant :

- Création d'objets clients et de véhicules divers;
- Création de diverses pièces et services;
- Appels aux méthodes de classe de produits ;
- Création de différentes factures, en utilisant les 3 constructeurs possibles ;
- Appels aux méthodes de la classe ProduitFacture ;
- Appels aux méthodes de la classe Facture;
- Deux scénarios d'enregistrement des factures :
    - un avec une facture pas finalisée;
    - un autre avec la facture finalisée.
- Utilisez la version suivante **comme base pour compléter** la vôtre:

```
#include <iostream>
#include <stdexcept>
// TODO: d'autres includes

using namespace std;

// En supposant que les classes Client, Vehicule, Produit, Piece, Service,
ProduitFacture, Facture, FactureFinaliseeException, FacturePasFinaliseeException
soient définies…

int main() {
 // Création de pièces et services
 cout << "**** Pieces et service ****" << endl;
 Piece* piece = new Piece("Piece", 10.0, 10);
 Service* service = new Service("Service", 20.0, 3);
 // TODO: ajoutez plus de pièces et de services

 piece->afficher();
 service->afficher();
 // TODO: ajoutez les appels des méthodes d'autres pièces et services

 cout << "\n**** ProduitFacture ****" << endl;
  // Création de produit-facture
 ProduitFacture* pfacture = new ProduitFacture(piece, 6);
 pfacture->afficher_produit();
 cout << *pfacture << endl;
 //TODO: créez plus des ProduitFacture
```

```
 // Création de clients et de véhicules
cout << "\n**** Clients et véhicules ****" << endl;
Client* client1 = new Client("Eduardo", "Rue X", "666 666-66-66");
Vehicule* vehicule1 = new Vehicule("JHJ 233", "Honda", "2020", "Voiture");
//TODO: créez plus des clients et de véhicules

// Création de facture
cout << "\n**** Factures ****" << endl;
Facture* facture = new Facture("11-11-2023");
// Ajout de client e véhicule à la facture
facture->set_client(client1);
facture->set_vehicule(vehicule1);

//TODO: utilisez les autres versions de constructeur pour la class Facture



cout << "\n**** Factures: ajouter produits ****" << endl;
// Ajout de produits à la facture
facture->ajouter_produit(pfacture);
facture->ajouter_produit(service, 4);

cout << "\n**** Factures: afficher factures ****" << endl;
facture->afficher_facture();

//TODO: ajoutez les produits aux autres factures
//TODO: affichez toutes les factures

cout << "\n**** Factures: scénarios d'exception ****" << endl;
//Scénario pour finaliser une facture
try {
  facture->finaliser_facture();
} catch (const FactureFinaliseeException& e) {
  cout << e.what() << endl;
}

//TODO: Scénario pour finaliser une facture que demande plus des pièces qui sont
disponibles en dépôt

//Scénario pour enregistrer une facture finalisée
try {
  facture->enregistrer_facture();
} catch (const FacturePasFinaliseeException& e) {
  cout << e.what() << endl;
}
```

```
//TODO: Scénario pour essayer d'enregistrer une facture pas finalisée

delete piece;
delete service;
delete pfacture;
delete client1;
delete vehicule1;
delete facture;

return 0;

}
```

- **"TODO"**: signifie "tâches à accomplir" ou "à faire". Dans ce contexte, cela indique les endroits du code à ajouter ou modifier du code pour répondre aux exigences spécifiques du travail pratique. Par exemple, ajouter plus de pièces et de services, créer des scénarios de test supplémentaires ou gérer des cas exceptionnels.
- Une sortie possible pour la fonction ci-dessus (sans TODO implémentés) :

```
**** Pieces et service ****
Piece: Piece, $ 10, En Depot: 10
Service : Service, $ 20, Execution : 3h

**** ProduitFacture ****
Qte : 6, Produit/Service : Piece (P), Unitaire : 10, Total : 60
6         | Piece                (P)| 10             | 60

**** Clients et vehicules ****

**** Factures ****

**** Factures: ajouter produits ****

**** Factures: aficher_factures ****
------------------------------------------------------------
                      FACTURE
------------------------------------------------------------
Client : Eduardo
Adresse : Rue X
Phone : 666 666-66-66
Vehicule : Honda| Plaque: JHJ 233
------------------------------------------------------------
Qte       | Produit / Service       | Unitaire       | Total
6         | Piece                (P)| 10             | 60
5         | Service              (S)| 20             | 300
------------------------------------------------------------
                                            Balance| 360
```

```
**** Factures: scénarios d'exception ****
```

Formatation du code

Concernant la formatation du code, il est essentiel :

- que les noms des classes et des méthodes soient conformes au diagramme de classes fourni;
- les noms des classes doivent suivre la convention CamelCase, tandis que les méthodes et les variables doivent utiliser la notation snake_case;
- l'utilisation appropriée des fichiers d'en-tête (.h) et des fichiers de code source (.cpp) est requise pour une organisation claire du code;
- il est également important de respecter les bonnes pratiques de codage, telles que la clarté des commentaires, l'indentation cohérente et l'utilisation judicieuse des espaces pour garantir la lisibilité du code.