Hiver 2018

Programmation avancée en C++

GIF-1003

Thierry EUDE, Ph.D

Travail individuel

à rendre avant
jeudi 12 avril 2018 14h
(Voir modalités de remise à la fin de l'énoncé)

Tout travail remis constitue une contribution originale et distincte de travaux remis par d'autres. Le plagiat est strictement défendu. Tout travail plagié obtiendra la note 0. De plus, les étudiants ou étudiantes ayant collaboré au plagiat seront soumis aux sanctions normalement prévues à cet effet par les règlements de l'Université.

Par ailleurs, vu l'importance des communications écrites dans le domaine de la programmation, il sera tenu compte autant de la présentation que de la qualité du français et ce, dans une limite de 10% des points accordés.

Travail Pratique #3

Hiérarchie de classes, contrat, test unitaire, gestion mémoire

UNIVERSITÉ LAVAL
Faculté des sciences et de génie
Département d'informatique
et de génie logiciel

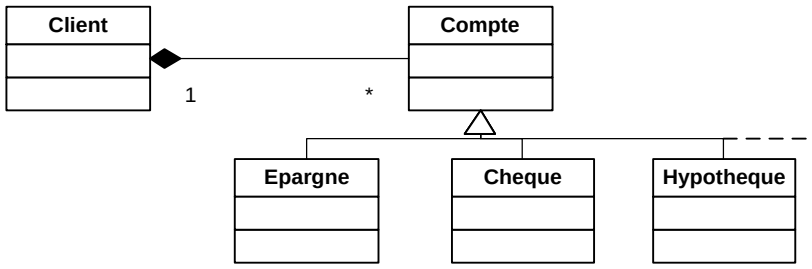
Notre objectif final est de construire un outil de gestion de comptes des clients d'une banque. La série des travaux pratiques devrait tendre vers cet objectif. Chaque travail pratique constituera donc une des étapes de la construction de cet outil.

But du travail

- Apprivoiser le processus de développement de plusieurs classes dans un environnement d'implémentation structuré;
- Faire la maintenance de code existant;
- Apprivoiser les concepts d'héritage et de polymorphisme.
- Utiliser un conteneur de STL.
- Respecter des normes de programmation et de documentation.
- Utiliser un outil de génération de documentation
- Utiliser la théorie du contrat et mettre en place les tests unitaires.

Mise en place de la hiérarchie des comptes

Au sommet de la hiérarchie, la classe Compte sera utilisée. Cette classe est spécialisée par des types de comptes différents. Le premier est Epargne, le second est Cheque, on pourra en imaginer d'autres dans le futur :



On se servira de cette hiérarchie pour implémenter le polymorphisme.

Classe Compte

La classe abstraite Compte représente tous les types de comptes d'un client. Elle contient les attributs :

`int m_noCompte` (doit être positif)
`double m_tauxInteret`
`double m_solde`
`string m_description` (doit être non vide)
`Date m_dateOuverture` (date à la création du compte)

La classe inclut un constructeur qui aura la signature suivante :

```
Compte      (int p_noCompte,
             double  p_tauxInteret
             double p_solde,
             const std::string& p_description)
```

On construit un objet Compte à partir de valeurs passées en paramètre. Si chacun de ces paramètres n'est pas considéré valide, une erreur de contrat sera générée. Pour déterminer les conditions préalables, voir la description des attributs.

La classe inclut aussi les méthodes de lecture et d'assignation suivantes:

```
reqNoCompte
reqTauxInteret
reqSolde
reqDescription
reqDateOuverture
asgTauxInteret
asgSolde
asgDescription
```

```
string reqCompteFormate()
```
Cette méthode retourne dans un objet string les informations sur le compte dans le format suivant :

```
numero                  : 12345
Description             : compte courant
Date d'ouverture        : Mercredi le 07 mars 2018
Taux d'interet          : 0.2
Solde                   : 3000 $
```
Utilisez la classe ostringstream du standard pour formater les informations sur le compte. La déclaration de cette méthode doit permettre un traitement polymorphe dans toute la hiérarchie de classe.

Vous devez également ajouter les méthodes suivantes :

```
virtual double calculerInteret()
```
Cette méthode permet de calculer les intérêts d'un compte selon sa spécialisation. Toutefois, au niveau de la classe Compte, il n'est pas nécessaire de la définir, sa déclaration doit imposer son implémentation dans toute classe dérivée; cette méthode est purement virtuelle.

La classe inclut également les méthodes suivantes:
```
virtual ~Compte (){} ;
```
Destructeur virtuel (voir manuel page 315 ou Penser en C++ - Bruce Eckel)

```
virtual Compte* clone() const
```
Cette déclaration doit imposer l'implémentation de cette méthode dans toute classe dérivée. Il n'y a rien à définir dans la classe Compte pour cette méthode (voir classes dérivées).

Classe Epargne

La classe Epargne représente les comptes dont les intérêts sont calculés en fonction du taux d'intérêt et du solde. Le taux d'intérêt annuel minimum est de 0.1% et au maximum 3.5%.

La classe inclut les méthodes suivantes :

```
Epargne (int p_noCompte, double p_tauxInteret, double p_solde,
             const std::string& p_description="Epargne")
```
Constructeur de la classe. La description est "Epargne" par défaut, mais peut être autre chose.

```
virtual double calculerInteret()
```
calcule l'intérêt de la manière « simplifiée » suivante : Taux d'intérêt multiplié par le solde (on pourra envisager de faire mieux plus tard…).

```
string reqCompteFormate() const
```
Cette méthode retourne dans un objet string les informations sur le compte sous la forme :

```
Compte Epargne
numero                  : 12345
Description             : Etudes
Date d'ouverture        : Mercredi le 07 mars 2018
Taux d'interet          : 0.3
Solde                   : 3000 $
Interet                 : 9$
```

```
virtual Compte* clone() const;
```
Cette méthode permet de faire une copie allouée sur le monceau de l'objet courant. Pour faire une copie, il s'agit simplement de faire :
```
return new Epargne(*this);  // Appel du constructeur copie
```

Classe Cheque

La classe Cheque représente les comptes dont les intérêts sont calculés en fonction du nombre de transactions et du taux d'intérêt. Elle contient les attributs :
```
int m_nombreTransactions
double m_tauxInteretMinimum
```

Un compte chèque ne peut pas avoir plus de 40 transactions.
Le taux d'intérêt minimum par défaut est de 0.1%.
La relation d'ordre entre le taux du compte et le taux d'intérêt minimum doit bien sûr être respectée.

La classe inclut les méthodes suivantes :

```
Cheque( int p_noCompte, double p_tauxInteret, double p_solde,
            int p_nombreTransactions, double p_tauxInteretMinimum,
            const std::string& p_description)
```
Constructeur de la classe. Par défaut la description devrait être "Cheque". On construit un objet Cheque à partir de valeurs passées en paramètre. Si chacun de ces paramètres n'est pas considéré comme valide, une erreur de contrat sera générée.

```
void   asgNombreTransactions (int p_nombreTransactions)
```
pour modifier le nombre de transactions

```
reqTauxInteretMinimum
reqNombreTransactions
```
Les accesseurs de la classe

```
virtual double calculerInteret()
```
L'intérêt se calcule sur un solde négatif (si le solde est positif ou nul, il n'y a pas d'intérêt à payer) de la manière suivante :
- entre 0 et jusqu'à 10 transactions, l'intérêt est payé au taux d'intérêt minimum.
- de 11 jusqu'à 25 transactions, l'intérêt est payé au taux d'intérêt du compte fois 40%.
- de 26 jusqu'à 35 transactions, l'intérêt est payé au taux d'intérêt du compte fois 80%.
- au-delà de 35 transactions, l'intérêt est payé au taux d'intérêt du compte;

`string reqCompteFormate()`
Cette méthode retourne dans un objet string les informations sur le compte sous la forme :

```
Compte Cheque
numero                 : 12345
Description            : Mon compte courant
Date d'ouverture       : Mercredi le 07 mars 2018
Taux d'interet         : 1.2
Solde                  : -3000 $
nombre de transactions : 4
Taux d'interet minimum : 0.3
Interet                : 9 $
```

`virtual Compte* clone() const;`
Cette méthode permet de faire une copie allouée sur le monceau de l'objet courant. Pour faire une copie, il s'agit simplement de faire :
`return new Cheque(*this);  // Appel du constructeur copie`

Classe Client

La classe Client programmée au deuxième tp, doit être modifiée. Elle permet de faire la gestion des comptes. **Vous devez y inclure la théorie du contrat**.
Rappel des attributs :
`int m_noFolio`
Le numéro de folio du client. Doit être dans l'intervalle [1000, 10000[
`std::string m_nom;`
`std::string m_prenom;`
Le nom et le prénom du client. Doivent être dans un format valide tel que déterminé par la fonction `util ::validerFormatNom`.
`std::string m_telephone;` Le numéro de telephone du client. Le numéro de téléphone doit être dans un format valide tel que déterminé par la fonction `util::validerTelephone` développée dans la deuxième partie du projet (tp1).
`util::Date m_dateNaissance;` // **attention, cet attribut a changé, il remplace m_dateOuverture**
La date de naissance du client. Ne couvre que l'intervalle [1970, 2037].
Remarque :une nouvelle classe Date incluant le contrat vous est fournie.

On retrouve dans la classe Client le constructeur modifié, les méthodes d'assignation et de lecture, et des opérateurs surchargés :

`Client (…`
Constructeur de la classe. On construit un objet Client à partir de valeurs passées en paramètre. Si chacun de ces paramètres n'est pas considéré comme valide, une erreur de contrat sera générée. Pour déterminer les conditions préalables, voir la description des attributs. Attention, vous devez ajouter un paramètre de type Date au constructeur du TP2. La Date de naissance doit être passée en paramètre.
Les accesseurs et le mutateur sont inchangés:
`reqNoFolio`
`reqNom`
`reqPrenom`
`reqTelephone`
`asgTelephone`

`reqDateDeNaissance` remplace reqDateOuverture

`reqClientFormate`
doit être modifiée
Elle retourne dans un objet std::string les informations correspondant à un client formatées sous le format suivant :

```
Client no de folio :5000
Louis Jean
Date de naissance : Samedi le 12 mai 1979
418 656-2131
```

Opérateur de comparaison d'égalité. La comparaison se fait sur la base du numéro de folio, du nom, du prénom de la date de naissance et du numéro de téléphone du client.
`operator==`

L'opérateur de comparaison d'infériorité reste inchangé. La comparaison se fait sur la base du numéro de client.

```
bool operator<(const Client& p_client)
```

Par ailleurs, la classe Client doit contenir tous les comptes du client dans un conteneur de type « vector »:

```
std::vector<Compte*> m_vComptes;
```

En fait, ce sont des pointeurs à Compte. Pour faire du polymorphisme en C++, il est nécessaire d'avoir le pointeur sur un objet.

Elle contient aussi la méthode privée :

```
bool compteEstDejaPresent(int p_noCompte) const;
```

Cette méthode permet de vérifier si le client a déjà ce compte. Si oui, elle retourne true et false sinon.

Elle contient aussi la méthode publique suivante :

```
void ajouterCompte (const Compte& p_nouveauCompte);
```

Cette méthode permet d'ajouter un compte au vecteur de comptes seulement si le compte n'est pas déjà présent dans cette liste. Autrement, il ne « se passe rien ». La classe Client conserve des pointeurs sur des comptes et elle est responsable de la gestion de la mémoire. Pour réaliser cet objectif, le compte passé par référence constante est cloné et ajouté dans le vecteur. Les comptes ont une méthode virtuelle clone() pour faire cela.
Exemple d'ajout d'un compte dans le vecteur :

```
m_vComptes.push_back(p_nouveauCompte.clone());
```

La méthode publique suivante doit être définie :

```
std::string reqReleves() const;
```

Cette méthode parcourt tous les comptes du client et retourne dans une chaine de caractères (string) les informations sur ceux-ci sous le format suivant :

```
Client no de folio :5000
Louis Jean
Date de naissance : Samedi le 12 mai 1979
418 656-2131
Compte Cheque
numero                 : 10001
Description            : courant
Date d'ouverture       : Mercredi le 07 mars 2018
Taux d'interet         : 1.2
Solde                  : 500 $
nombre de transactions  : 4
Taux d'interet minimum  : 0.3
```

```
Interet                 : 0 $
Compte Epargne
numero                  : 10007
Description             : voyage
Date d'ouverture        : Mercredi le 07 mars 2018
Taux d'interet          : 1.25
Solde                   : 500 $
Interet                 : 6.25$
```

Finalement, la classe Client contient un destructeur qui est responsable de désallouer tous les comptes du client dans le vecteur.

```
~Client();
```

La classe étant d'une forme de Coplien, il faut prévoir, un constructeur copie et un opérateur d'assignation. On se satisfera de seulement empêcher leur utilisation par un code utilisateur.

Théorie du contrat

Les classes doivent implanter la théorie du contrat en mettant les PRECONDITIONs, POSTCONDITIONs et INVARIANTs aux endroits appropriés. Bien sûr, la méthode `void verifieInvariant() const` doit être implantée pour vérifier les invariants. Voir l'exemple avec la présentation de la théorie du contrat. Vous devez aussi déterminer les endroits où tester l'invariant de la classe.

- Ajouter dans votre projet les fichiers `ContratException.h` et `ContratException.cpp` pour implanter la théorie du contrat.

Test unitaire

Pour chaque classe, vous devez construire un test unitaire selon la méthode prescrite dans le cours en vous appuyant sur la théorie du contrat. Les fichiers de test doivent s'appeler, pour respecter les conventions :

```
CompteTesteur.cpp
CompteEpargneTesteur.cpp;
CompteChequeTesteur.cpp;
ClientTesteur.cpp;
```

Documentation

Toutes **les classes développées** devront être correctement commentées pour pouvoir générer une documentation complète à l'aide de l'extracteur DOXYGEN. Des précisions sont fournies sur le site Web pour vous permettre de l'utiliser (syntaxe et balises à respecter, etc.). La documentation du programme minimaliste n'est pas attendue, ni celle des testeurs bien qu'il soit préférable que ces derniers soient documentés.

Utilisation

Après avoir implanté et testé les classes demandées, vous devez écrire un programme minimaliste qui utilise Client avec les différents types de comptes.

Le programme commence par créer un client (les données nécessaires (numéro, nom, prénom, etc.) sont demandées à l'usager). Vous pouvez réutiliser ici ce que vous avez fait dans le TP2.
Il est ensuite demandé à l'usager de saisir des informations pour créer puis ajouter successivement deux nouveaux comptes (un compte Cheque et un compte Epargne).
Rappelons que le client ainsi que les comptes ne peuvent être construits qu'avec des valeurs valides. C'est la responsabilité du programme principal qui les utilise de s'assurer que ces valeurs sont valides. Les critères de validité ont été énoncés dans la description des attributs des classes.
Le relevé de compte du client est ensuite affiché.

Modalités de remise, bien livrable

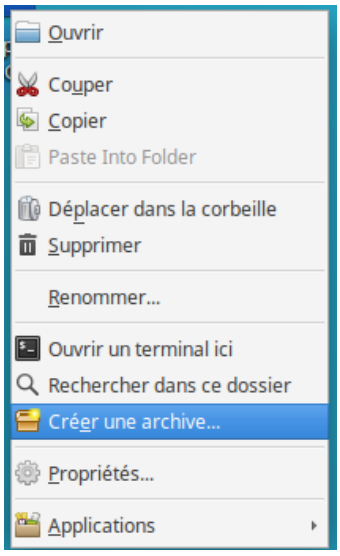
Le troisième travail pratique pour le cours GIF-1003 Programmation avancée en C++ est un travail individuel. Vous devez, en utilisant le dépôt de l'ENA, remettre votre environnement de développement complet, dans un espace de travail (workspace) Eclipse mis dans une archive **.7z**.

Très important

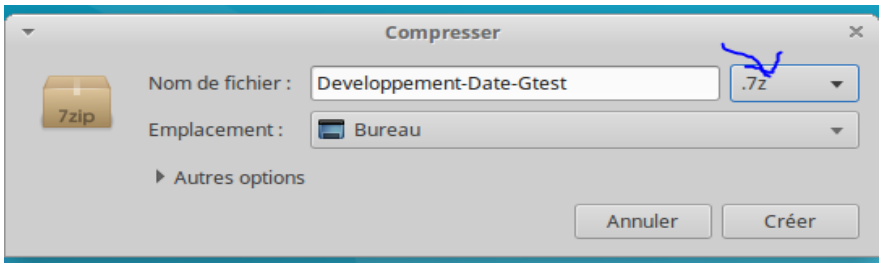
Pour faire votre archive, et pour éviter les problèmes dus à une archive trop volumineuse à la remise, auparavant dans le répertoire de votre workspace :

- .metadata\.plugins\org.eclipse.cdt.core , supprimer tous les fichiers d'extension .pdom.
- .metadata\.plugins\org.eclipse.core.resources supprimer le répertoire .history
- La documentation générée avec Doxygen. Le correcteur devrait pouvoir la régénérer par un simple clic dans Eclipse.

Rappel: un tutoriel est disponible sur la page des travaux pratiques pour vous guider, mais attention, bien choisir le format **7z** et non zip comme dans le tutoriel.
Utilisez l'outil natif de la machine virtuelle Linux du cours:
Clic droit sur le répertoire



choisir créer une archive, sélectionner **.7z** (premier de la liste)



Ce travail est intitulé TP 3. Aucune remise par courriel n'est acceptée.
Vous pouvez remettre autant de versions que vous le désirez.
Pensez à supprimer vos anciennes versions sur l'ENA pour ne laisser que celle qui sera corrigée. **Il est de votre responsabilité de vous assurer de ce que vous avez déposé sur le serveur.**

Date de remise

Ce travail doit être rendu avant le jeudi 12 avril 2018 14h. Pour tout retard non motivé (voir plan de cours; motifs acceptables pour s'absenter à un examen), la note 0 sera attribuée.

Critères d'évaluation

1) Respect des normes de programmation,
2) Documentation avec DOXYGEN
3) Structures, organisation du code (très important!)

Ce critère sera utilisé pour évaluer la qualité 5, utilisation des outils d'ingénierie du BCAPG, tel que précisé dans les objectifs spécifiques du plan de cours. En particulier :

- Organisation de l'espace de travail
- Configuration de l'espace de travail (propriétés)
- Utilisation des outils recommandés (contrat, test unitaire)
- Adaptation de l'espace de travail (personnalisation)

4) Exactitude du code
5) Théorie du contrat et tests unitaires
6) Utilisation des classes, c.-à-d. le programme principal

Particularités du barème

- *Si des pénalités sont appliquées, elles le sont sur l'ensemble des points.*
- *Si un travail comporte ne serait-ce qu'une erreur de compilation, il sera fortement pénalisé, et peut même se voir attribuer la note zéro systématiquement.*
- *Il est très important que votre travail respecte strictement les consignes indiquées dans l'énoncé, en particulier les prototypes des méthodes, les noms des fichiers et la structure de développement sous Eclipse sous peine de fortes pénalités*

Bon travail!