Automne 2015

Programmation avancée en C++

GIF-1003

Thierry EUDE, Ph.D

Travail à faire exclusivement
en équipe de 2 personnes
à rendre avant
jeudi 10 décembre à 12h (midi)
(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 4
Intégration, travail en équipe

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

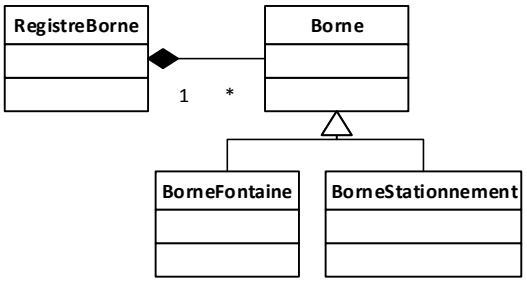
Notre objectif final est de construire un outil permettant au service de l'aménagement du territoire de la ville de Québec de localiser rapidement les bornes de paiement (horodateurs) ou de stationnement et les bornes fontaine. La série des travaux pratiques devrait tendre vers cet objectif. Chaque travail pratique constituera donc une des étapes de la construction de cet outil.

But du travail

- Utiliser les exceptions
- Respecter des normes de programmation et de documentation.
- Utiliser la théorie du contrat et mettre en place les tests unitaires.
- Utiliser des itérateurs pour les accès aux conteneurs de la STL
- Approfondir la gestion de l'utilisation de la mémoire
- Intégrer des classes préalablement testées pour le développement d'une application
- Développer une application avec interface graphique
- Utiliser un système de gestion de versions (Git)

Utilisation de la hiérarchie des bornes

Au sommet de la hiérarchie, la classe `Borne` sera utilisée. Cette classe est spécialisée par deux types différents. Le premier est `BorneFontaine`, le second est `BorneStationnement`:



On se servira de cette hiérarchie pour implémenter le polymorphisme.

Ce que vous avez développé pour le TP3 va donc être réutilisé et complété.

Classe Borne

La classe Borne programmée au troisième TP, représente tous les types de Bornes présents dans la ville de Québec. Elle reste inchangée.

Classe BorneFontaine

La classe `BorneFontaine` programmée au troisième TP, représente les bornes fontaine de la ville de Québec. Elle reste inchangée.

Classe BorneStationnement

La classe `BorneStationnement` représente les bornes de stationnement et de paiement (horodateurs) de la ville de Québec. Elle reste inchangée.

Classe RegistreBorne

La classe RegistreBorne permet la gestion des bornes fontaine et de stationnement.
Cette classe développée au tp3; doit être modifiée (tous les « parcours » de `m_vBornes` doivent être faits à l'aide d'itérateur) et complétée.

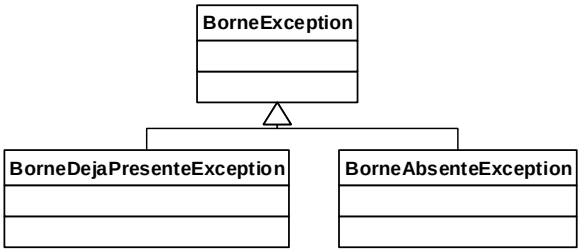
Vous devez réécrire le destructeur ainsi que la méthode privée `bool BorneEstDejaPresente` en utilisant un itérateur plutôt que d'utiliser l'indexage pour les tableaux (notation [i]).

La méthode publique suivante doit être modifiée:
```
void ajouteBorne(const Borne& p_Borne);
```
Cette méthode permet d'ajouter une borne au vecteur de bornes **seulement si la borne n'est pas déjà présente dans cette liste**. Autrement, l'ajout n'est pas effectué.
Dans le cas où la Borne est déjà existante, une exception du type `BorneDejaPresenteException` sera lancée. On construit l'exception avec un objet string décrivant la situation. Par exemple :
```
throw BorneDejaPresenteException(nouvelleBorne.reqBorneFormate());
```
pourrait faire l'affaire.

La méthode
```
void supprimeBorne(int p_idBorne);
```
doit être ajoutée à la classe. Cette méthode supprime une borne de la liste dont l'identifiant (idBorne) est reçu en paramètre. S'il n'y a pas de Borne qui possède cet identifiant dans la liste des bornes, une `BorneAbsenteException` est lancée (voir plus bas). Sinon, la méthode doit trouver la position de la borne dans le vecteur, avec un itérateur, et appeler la méthode `erase` :
```
m_vBornes.erase (iter);
```
où `iter` pointe sur la position de la borne à supprimer. N'oubliez pas de libérer la mémoire avant de supprimer le pointeur dans le vecteur.

Hiérarchie d'exception

Pour ce travail, il est demandé d'élaborer les classes d'une nouvelle hiérarchie d'exception. Toutes ces classes doivent être déclarées et définies dans le même module nommé `BorneException`, comme pour la théorie du contrat.



Classe BorneException

Cette classe permet de gérer les exceptions liées aux bornes. Cette classe hérite de `std::runtime_error`. Elle contient seulement une méthode qui est le constructeur suivant :
```
BorneException (const std::string& p_raison);
```
Ce constructeur ne fait qu'appeler le constructeur de la classe parent en lui passant la raison comme paramètre.

Classe BorneDejaPresenteException

Cette classe permet de gérer l'exception de l'ajout d'un doublon de Borne dans la liste des bornes. Cette classe hérite de `BorneException` parce que cette exception est une erreur qui pourra toujours se produire, ce n'est pas une erreur de programmation.
Elle contient seulement une méthode qui est le constructeur suivant :

```
BorneDejaPresenteException (const std::string& p_raison);
```

Ce constructeur ne fait qu'appeler le constructeur de la classe parent en lui passant la raison comme paramètre.
La méthode `what()` de la classe `std::exception` en haut de la hiérarchie permet d'obtenir l'objet string que l'on a passé.

Classe BorneAbsenteException

Cette classe permet de gérer l'exception de la tentative d'effacement d'une Borne absente dans la liste des bornes. Cette classe hérite de `BorneException` parce que cette exception est une erreur qui pourra toujours se produire, ce n'est pas une erreur de programmation.
Elle contient seulement une méthode qui est le constructeur suivant :

```
BorneAbsenteException (const std::string& p_raison);
```

Ce constructeur ne fait qu'appeler le constructeur de la classe parent en lui passant la raison comme paramètre.
La méthode `what()` de la classe `std::exception` en haut de la hiérarchie permet d'obtenir l'objet string que l'on a passé.

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 (module pour implanter la théorie du contrat : disponible sur la page Contenu et activités/Travaux Pratiques/Documents).

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 normes : `BorneTesteur.cpp, BorneFontaineTesteur.cpp, ...`
Cependant, comme les classes `Borne`, `BorneFontaine` et `BorneStationnement` n'ont pas changé par rapport au TP3, les testeurs pour ces classes ne seront pas évalués, seuls les testeurs des classes modifiées le seront.
Les testeurs des classes d'exception ne sont pas attendus.

Documentation

Toutes les classes ainsi que toutes les méthodes devront être correctement commentées pour pouvoir générer une documentation complète à l'aide de l'extracteur DOXYGEN (il ne doit pas y avoir d'erreur de compilation de signalée). Des précisions sont fournies sur le site Web (dans activité de la semaine 6, voir également dans les normes de programmation) pour vous permettre de l'utiliser (syntaxe et balises à respecter etc.).
Remarque importante :
La documentation du code de la partie intégration n'est pas attendue.

Utilisation

Après avoir implanté, modifié et testé les classes comme demandé, vous devez écrire un programme proposant une interface graphique (GUI) qui utilise la classe RegistreBorne avec les différents types de Borne.
Ce programme devra être « construit » en utilisant le framework Qt.
La fenêtre principale proposera un menu Operations.
Les fonctionnalités suivantes devront être disponibles:

- Ajout d'une borne fontaine
- Ajout d'une borne de stationnement
- Suppression d'une borne
- Quitter (pour terminer l'application)

Pour l'ajout, le choix d'un type fera apparaitre une fenêtre de dialogue permettant de saisir les informations sur la borne, et ce selon le type choisi précédemment (boites de dialogue spécifiques au type de borne).
Une fois les informations saisies, l'action sur un bouton ok provoquera un retour à la fenêtre principale. Au centre de celle-ci on retrouvera l'affichage des informations formatées sur le registre mise à jour.
Dans le cas d'une borne déjà présente (ajout) ou absente (suppression) un message sera affiché.

Le choix de la présentation (ergonomie, esthétique, …) est laissé à votre discrétion.

Remarque

Il aurait été intéressant de pouvoir charger le registre à partir de fichier de données comme évoqué dans le tp1. En effet, le travail demandé consistait en partie à « préparer » ce chargement en vérifiant la validité des données lues et prendre en charge l'extraction de données pertinentes pour l'application, ceci à partir des fichiers de données disponibles. L'intégration de ce travail dans cette dernière version du projet de session aurait donc pu être très intéressante. Il a malheureusement fallut que je fasse un choix des priorités devant le peu de temps qu'il vous reste. Cela pourrait faire l'objet d'un travail personnel pour les curieux… utiliser Qfile et extraire des données d'un fichier pour charger le registre avant de pouvoir ajouter de nouvelles bornes, en supprimer… bref permettre la gestion de ce registre.

Travail d'équipe, utilisation du système de gestion de version

Vous devez vous organiser avec votre partenaire pour la répartition des tâches. Attention, ne vous « spécialisez » pas. Faites en sorte que chacun puisse mettre en pratique toutes les notions abordées. Utilisez un dépôt Git (sur le serveur de la FSG) pour pouvoir partager et suivre l'évolution du code.
Pour votre dépôt Git, si vous ne l'avez pas fait à l'occasion du laboratoire 4, vous devez faire une demande de dépôt en utilisant Pixel, menu Applications/Logiciels :

Des tutoriels présentant le fonctionnement de Git, son utilisation à l'aide différents outils, des exercices etc. sont à votre disposition sur le site Web du cours dans notes de cours/ Git - Principes et utilisations.
Revoir à l'occasion, l'enregistrement du laboratoire 4, semaine 4.

Modalités de remise, bien livrable

Le quatrième travail pratique pour le cours GIF-1003 Programmation avancée en C++ doit être réalisé **exclusivement en équipe de 2 personnes**. Vous devez remettre votre environnement de développement, soit le code source dans un espace de travail (workspace) Eclipse mis dans une archive **ZIP** en utilisant le dépôt de l'ENA :.
Attention, vérifiez qu'une fois déplacé et décompressé, votre workspace est toujours fonctionnel (il doit donc être « portable »). Pensez au correcteur ! Sachez qu'il utilisera la machine virtuelle fournie pour le cours. Ce travail est intitulé TP #4. **Aucune remise par courriel n'est acceptée**.
Vous pouvez remettre autant de versions que vous le désirez (éventuellement numérotez vos versions). Seul le dernier dépôt est conservé. **Il est de votre responsabilité de vous assurer de ce que vous avez déposé sur le serveur**.

⚠ 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.

Utilisez 7-zip directement dans la machine virtuelle pour faire votre archive. Rappel : Votre archive doit être une archive **zip et contenir le répertoire de votre workspace au complet.**.

Date de remise

Ce travail doit être rendu avant le **jeudi 10 décembre 2015 12h (midi)**. 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 (uniquement pour les classes développées)
3) Structures, organisation du code, personnalisation de l'environnement de développement (très important!)
4) Exactitude du code
5) Théorie du contrat et tests unitaires
6) Le programme principal (fonctionnalité, convivialité de l'interface)
7) Utilisation du système de gestion de versions (Git)

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!