Automne 2015

Programmation avancée en C++

GIF-1003

Thierry EUDE, Ph.D

Travail individuel
à rendre avant
jeudi 26 novembre 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 3

Hiérarchie de classes, contrat, test unitaire, gestion mémoire

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

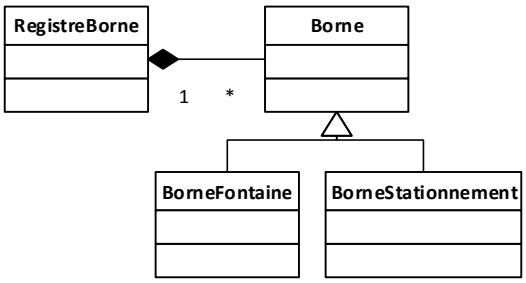
Notre objectif final est de construire un outil permettant au service de l'aménagement du territoire de la ville de Québec de localiser rapidement les bornes de paiement (horodateurs) ou de stationnement et les bornes fontaine. La série des travaux pratiques devrait tendre vers cet objectif. Chaque travail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

But du travail

- 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
- Optimiser son environnement de développement
- Utiliser la théorie du contrat et mettre en place les tests unitaires.

Mise en place 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.

Classe Borne

La classe Borne programmée au deuxième TP, représente tous les types de Bornes présents dans la ville de Québec. Elle contient les attributs :

`int m_idBorne;`
: L'identifiant de la borne.

`std::string m_direction;`
: Le coté du centre de chaussée ou l'intersection dans le cas d'un terre-plein. Il doit correspondre à un point cardinal. La validité du point cardinal doit être établie par la fonction `util::validerPointCardinal` (voir plus loin dans fonction de validation).

`std::string m_nomTopographique;`
: Le nom topographique (générique, liaison, spécifique, direction) du centre de chaussée. Il doit être non vide.

`double m_longitude`
`double m_latitude`
: La longitude et la latitude.

Méthodes d'assignation et de lecture:

```
reqId
reqDirection
reqNomTopographique
reqLongitude()
reqLatitude()
void asgNomTopographique(const std::string& p_nomTopographique)
```

Méthode permettant de changer le nom topographique de la borne.

L'opérateur de comparaison d'égalité. La comparaison se fait sur la base de tous les attributs.

```
bool operator==(const Borne& p_borne)
```

le constructeur aura la signature suivante :

```
Borne( int p_idBorne,
       const std::string& p_direction,
       const std::string& p_nomTopographique,
       double p_longitude,
       double p_latitude);
```

Constructeur de la classe. On construit un objet `Borne` à partir de valeurs passées en paramètre. Chacun de ces paramètres est considéré valide. **Autrement, une erreur de contrat sera générée.** Pour déterminer les conditions préalables voir la description des attributs.

Il faut modifier la méthode

```
reqBorneFormate()
```

de sorte qu'elle soit virtuelle pure. Cette méthode retourne dans un objet `std::string` les informations correspondant à une borne formatée sous le format suivant :

```
Identifiant de la borne    : 300070
Direction           : Nord
Nom topographique   : Boulevard René-Levesque Est
Longitude           : -71.226669
Latitude            : 46.814323
```

Cette déclaration (méthode virtuelle pure, la classe devient abstraite) impose l'implémentation de cette méthode dans toute classe dérivée.

La classe inclut également les méthodes suivantes:

```
virtual ~Borne() {} ;
```

Destructeur virtuel (voir manuel page 456 ou Penser en C++ - Bruce Eckel)

```
virtual Borne* clone() const =0;
```

Cette déclaration impose l'implémentation de cette méthode dans toute classe dérivée. Il n'y a rien à définir dans la classe Borne pour cette méthode (voir classes dérivées).

Classe BorneFontaine

La classe `BorneFontaine` représente les bornes fontaine de la ville de Québec. Elle contient deux attributs représentant la ville et l'arrondissement dans lesquels elle se trouve:

```
std::string m_ville;
std::string m_arrondissement;
```

L'arrondissement ne doit pas être vide si la ville est « Quebec »

Il faut aussi prévoir un constructeur qui aura la signature suivante :

```
BorneFontaine(      int p_idBorne,
                    const std::string& p_direction,
                    const std::string& p_nomTopographique,
                    double p_longitude,
                    double p_latitude,
                    const std::string& p_ville,
                    const std::string& p_arrondissement);
```
Ce constructeur doit construire un objet BorneFontaine valide à partir de valeurs passées en paramètre. Si chacun de ces paramètres n'est pas considéré valide, une erreur de contrat sera générée.

La classe inclut les méthodes d'accès suivantes :
`reqArrondissement()`
`reqVille()`
qui retournent respectivement la ville et l'arrondissement.

Ajouter les méthodes :
`virtual Borne* 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 BorneFontaine(*this);  // Appel du constructeur copie
```

`virtual std::string reqBorneFormate() const;`
Méthode qui augmente la méthode `reqBorneFormate` de la classe de base `Borne` et qui retourne dans une `std::string` les informations correspondant à une Borne fontaine formatées sous le format suivant :
```
Borne fontaine
-----------------------------------------------
Identifiant de la borne    : 103270
Direction                  :
Nom topographique          : Boulevard de l'Ormière
Longitude                  : -71.35887584
Latitude                   : 46.83814462
Ville                      : Québec
Arrondissement             : La Haute-Saint-Charles
```
Utilisez la classe ostringstream de la bibliothèque standard pour formater les informations sur la Borne fontaine.

Classe BorneStationnement

La classe `BorneStationnement` représente les bornes de stationnement et de paiement (horodateurs) de la ville de Québec.
Elle contient les attributs suivants :

`std::string m_type;`
: Le type de borne de stationnement peut être « stationnement » ou « paiement »

`double m_lectureMetrique;`
: La distance mesurée à partir du début du tronçon dans le sens des numéros d'immeuble. Il doit être plus grand que 0.

`int m_segmentRue;`
: L'identifiant du segment de voie publique. Il doit être plus grand que 0.

`std::string m_numBorne;`
: Le numéro de la borne. Il doit être non vide.

`std::string m_coteRue;`
: Le coté par rapport au centre de chaussée où est la borne. Il doit correspondre à un point cardinal. La validité du point cardinal doit être établie par la fonction `util::validerPointCardinal` (voir plus loin dans fonction de validation).

La classe inclut la méthode de lecture suivante:
```
reqType
reqLectureMetrique
reqSegmentRue
reqNumBorne
reqCoteRue
```

```
virtual std::string reqBorneFormate() const;
```
Méthode qui augmente la méthode `reqBorneFormate` de la classe de base `Borne` et qui retourne dans un objet `std::string` les informations correspondant à une `Borne de stationnement` formatées sous le format suivant :

```
Borne de stationnement
-----------------------------------------
Identifiant de la borne   : 300070
Direction           : Nord
Nom topographique   : Boulevard René-Levesque Est
Longitude           : -71.226669
Latitude            : 46.814323
Distance mesuree    : 23.7
Segment de rue      : 20
Numero de la borne  : 2172
Cote de la rue      : Nord
```

Il faut aussi faire un constructeur qui aura la signature suivante :
```
BorneStationnement( int p_idBorne,
                    const std::string& p_direction,
                    const std::string& p_nomTopographique,
                    double p_longitude,
                    double p_latitude,
                    const std::string& p_type,
                    double p_lectureMetrique,
                    int p_segmentRue,
                    const std::string& p_numBorne,
                    const std::string& p_coteRue);;
```
Ce constructeur doit construire un objet BorneStationnement valide à partir de valeurs passées en paramètre. Si chacun de ces paramètres n'est pas considéré valide, une erreur de contrat sera générée.

Ajouter les méthodes :
```
virtual Borne* clone() const;
```
Celle-ci 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 BorneStationnement(*this);  // Appel du constructeur copie
```

Classe RegistreBorne

La classe RegistreBorne permet la gestion des bornes fontaine et de stationnement.
Elle a pour attributs :
```
std::string m_nomRegistreBorne;
```
La version du Registre (ne doit pas être vide).

On y trouve les constructeurs de la classe :
```
RegistreBorne(const std::string& p_nomRegistreBorne);
```
Le constructeur avec paramètre qui prend le nom du Registre. Ce nom ne doit pas être vide sinon une erreur de contrat sera générée.
Il faut également l'accesseur:
```
reqNomRegistreBorne
```

Par ailleurs, la classe RegistreBorne doit contenir toutes les bornes fontaine et de stationnement de la ville de Québec dans un vecteur :

```
std::vector<Borne*> m_vBornes;
```

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

Elle contient aussi la méthode privée suivante :

```
bool BorneEstDejaPresente(const Borne& p_borne) const;
```

Cette méthode permet de vérifier si la borne n'est pas déjà dans le vecteur (on pourra se limiter aux attributs de la classe de base, pensez à l'opérateur). Si oui, elle retourne true et false dans le cas contraire.

La classe RegistreBorne contient aussi la méthode publique suivante:

```
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. La classe RegistreBorne conserve des pointeurs sur des `Bornes` et elle est responsable de la gestion de la mémoire. Pour réaliser cet objectif, l'objet Borne passé par référence constante est cloné et ajouté dans le vecteur. Les `Bornes` ont une méthode virtuelle clone() pour faire cela.
Exemple d'inscription d'une Borne dans le vecteur :

```
m_ vBornes.push_back(p_Borne.clone());
```

La méthode publique suivante doit être également définie :

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

Cette méthode retourne dans un objet `std::string` les informations formatées concernant le registre de bornes. On commence par son nom, puis on parcourt toutes les bornes présentes dans le vecteur pour avoir les informations formatées sur chacune d'elle.
Exemple :

```
Registre : bornes de la ville de Québec 2015
Borne de stationnement
-----------------------------------------------
Identifiant de la borne   : 300070
Direction           : Nord
Nom topographique   : Boulevard René-Levesque Est
Longitude           : -71.226669
Latitude            : 46.814323
Distance mesuree    : 23.7
Segment de rue      : 20
Numero de la borne  : 2172
Cote de la rue      : Nord
-------------------------------------------------
Borne fontaine
-----------------------------------------------
Identifiant de la borne   : 103270
Direction           :
Nom topographique   : Boulevard de l'Ormière
Longitude           : -71.35887584
Latitude            : 46.83814462
Ville               : Québec
Arrondissement      : La Haute-Saint-Charles
-----------------------------------------------
```

Finalement, la classe RegistreBorne contient un destructeur qui est responsable de désallouer toutes les Bornes de la liste dans le vecteur.

```
~RegistreBorne ();
```

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.

Fonction de validation

Il s'agit de compléter le module de validation commencé au tp1.
Dans les fichiers `validationFormat.h` et `validationFormat.cpp`, vous devrez implanter une nouvelle fonction de validation dont les spécifications sont les suivantes.

```
bool validerPointCardinal(const std::string& p_cardinalite);
```

Cette fonction valide le format d'un point cardinal. Les valeurs possibles d'un point cardinal sont : Nord, Sud, Est, Ouest, null, « chaine vide ».
Insérez ce module de validation dans le namespace **util**.

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`, `BorneStationnementTesteur.cpp`, `RegistreBorneTesteur.cpp`.

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.).

Utilisation

Après avoir implantées et testées les classes Borne, BorneFontaine, BorneStationnement, et RegistreBorne, vous devez écrire un programme minimaliste qui utilise RegistreBorne avec les différents types de Bornes. Le programme commence par construire un registre (dont le nom est fixé à priori; il n'est donc pas saisi). Il est ensuite demandé à l'usager de saisir des informations pour créer puis ajouter successivement deux nouvelles bornes (une borne fontaine et une borne de stationnement).
Rappelons que les Bornes ne peuvent être construites 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 contenu du registre est alors affiché au complet.

Modalités de remise, bien livrable

Le troisième travail pratique pour le cours GIF-1003 Programmation avancée en C++ est un **travail individuel**. 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 #3. **Aucune remise par courriel n'est acceptée**.
Vous pouvez remettre autant de versions que vous le désirez. 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.

Ne pas utiliser « windows » pour faire votre archive ; son outil natif présente des lacunes. Utilisez à la place 7-zip (gratuit, inclus dans la machine virtuelle fournie pour le cours), ou winrar (shareware), ou mieux, faites-le dans la machine virtuelle du cours. Rappel : Votre archive doit être une archive **zip**.

Date de remise

Ce travail doit être rendu avant le **jeudi 26 novembre 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 (balises, commentaires, personnalisation du fichier doxyfile)
3) Structures, organisation du code dans l'espace de travail (très important!)
4) Personnalisation de l'environnement de développement (template, éditeurs, …)
5) Exactitude du code
6) Théorie du contrat et tests unitaires
7) Utilisation des classes, i.e. 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!