Projet Informatique – Sections Electricité et Microtechnique

Printemps 2020 :Archipelago

Rendu1 (dimanche 22 mars 23h59)

Objectif de ce document : *en plus de préciser ce qui doit être fait, ce document identifie des ACTIONS à considérer pour réaliser le rendu de manière rigoureuse. Ces ACTIONS sont équivalentes à celles indiquées pour le projet d'automne ; elles ne sont pas notées, elles servent à vous organiser. Vous pouvez adopter une approche différente du moment que vous respectez l'architecture minimale du projet (donnée Fig 11a). La différence principale avec le projet du semestre d'automne est qu'avec la compilation séparée il y a moins de contraintes concernant l'ordre d'exécution des actions, c'est pourquoi elles n'ont pas de numéro. Il y a aussi plus de liberté dans les choix de structuration des données ; vous avez une certaine marge de manœuvre si les options proposées ne vous conviennent pas. Sans surprise on retrouvera l'approche introduite en TP de Topic1 sur les méthodes de développement de projet.*

1. Buts du rendu1

Le but du rendu1 est de tester des fichiers pour vérifier s'il y a des collisions initiales et des incohérences dans la structure du graphe (sections 7 et 2.1 de la donnée).

Nous vous fournissons un module **error** comme détaillé maintenant.

1.1 utilisation du module error :
Le module **error** est fourni dans un fichier archive sur moodle ; il ne doit pas être modifié.
Son interface **error.h** détaille l'ensemble des fonctions à appeler.

Ces fonctions sont prévues pour afficher un message vers **cout** (et pas ailleurs) comme ceci :

```
if(une détection d'erreur est vraie)
      cout << error::appel_de_la_fonction(paramètres éventuels);
```

Une seule fonction n'est pas liée à une détection d'erreur ; elle doit être appelée quand la lecture de fichier et toutes les validations ont été effectuées avec succès :

```
cout << error::success();
```

1.2 Syntaxe du lancement de votre exécutable pour le rendu1 et quand il s'arrête :

```
./projet nom_fichier.txt
```

Le programme cherche à initialiser l'état de la ville mais il **s'arrête** dès la ***première*** erreur trouvée dans le fichier.

Il **s'arrête** aussi après la lecture du fichier s'il n'y a aucune erreur ; dans ce cas, il y a affichage d'un message indiquant le succès de la lecture.

Le rendu1 ne doit PAS utiliser GTKmm.

1.3 Comment utiliser le programme de démo pour le rendu1 :
Le programme de démo se comporte comme ce qui est demandé pour le rendu1 en l'appelant ainsi sur la VM:

```
./demo console nom_fichier.txt
```

2. Architecture du rendu1 (Donnée fig 11a) :

2.1 Module projet

Le module **projet** contient la fonction **main()** en charge d'analyser si la ligne de commande est bien conforme à la syntaxe (donnée section 7). Si c'est le cas elle délègue l'action de **lecture** et d'initialisation de la ville au module **ville** (Fig 11a). Il n'y a pas de structure de donnée de ville dans le module projet car c'est une responsabilité du Modèle.

2.1.1 ACTION : test du module projet / de la ligne de commande

La fonction main() peut très rapidement être testée pour savoir si le bon nombre d'argument est fourni. Si c'est le cas, transmettre la chaine de caractère du nom de fichier à un *stub* de la fonction du module **ville** en charge de lire un fichier. A ce stade le module **ville** peut être réduit à une interface très incomplète ; seule compte d'avoir la fonction utile pour la lecture pour faire ces tests.

2.2 Le sous-système du Modèle

2.2.1 Module tools

Commençons par le plus bas niveau : le module utilitaire **tools** qui peut travailler avec des types concrets réalisés avec des structures. Un point et un vecteur peuvent être représentés par une même structure **S2d** comme illustré dans ce slide qui a été ajouté au cours du Topic2. Pour représenter un cercle, il suffit d'un point et d'un rayon, tandis que pour un segment de droite, un point et un vecteur font l'affaire. Ensuite la liste des fonctions de test est vite faite : un test d'intersection de 2 cercles, et un test d'intersection cercle-segment. On veillera à ajouter un paramètre **dist_min** en vue de ré-utiliser ces fonctions dans les 2 scénarios prévus (donnée section 2.1.3).
Remarque : vous pouvez utiliser des classes pour ce module ; dans ce cas les attributs doivent être *private*.

2.2.1.1 ACTION : test du module tools

Comme pour le projet d'automne mettre en œuvre la méthode de l'échafaudage (*scaffolding*) en écrivant cette fois un module indépendant de test (comme prog.cc sur le slide). Ce module est en charge de vérifier les fonctions de test mises à disposition par le module tools. Veiller à couvrir une variété suffisamment large de positions relatives de cercles et de segments

2.2.2 Module nœud

On peut traiter le module nœud d'un point de vue géométrique dès qu'on dispose du module tools. Le centre d'un nœud peut être représenté avec un point ; on peut aussi parti d'un cercle dont le rayon sera lié à l'attribut nbp. Une décision importante est le choix de représentation des liens vers les nœuds voisins. Ce qui est clair c'est que ce nombre est variable et que nous connaissons un outil pour gérer cela : **`vector`**. Ensuite à vous de voir ce qui vous semble le plus pratique/efficace/robuste de mémoriser dans ce **`vector`** de liens vers les voisins. Pour répondre à cette question il est nécessaire de se poser les questions suivantes :

- Comment mémoriser l'ensemble des nœuds ?
  - Puisqu'il s'agit là aussi d'un nombre variable, un **vector** est pertinent. Ceux qui maîtrisent d'autres outils avancés peuvent les utiliser. Nous refusons toute solution à base de tableau de taille fixe.

- Où mémoriser l'ensemble des nœuds ?
  - Plusieurs stratégies sont acceptées :
    - **Un seul ensemble *global* au module nœud (et invisible en dehors de ce module)**: Si on suppose qu'un seul ensemble de nœuds existe, celui-ci pourrait être géré par le module nœud de manière implicite en étant global à ce module (mais invisible à l'extérieur de ce module). Cela se fait avec l'espace de nom non-nommé/static (cf slide). L'avantage est que les fonctions qui manipulent l'ensemble de nœuds n'ont pas besoin d'avoir un paramètre indiquant sur quel ensemble de nœuds elles travaillent. En effet elles y accèdent directement car cet ensemble est global (au module).
    - **Ensemble mémorisé au niveau supérieur comme un attribut de la classe ville** : dans ce cas il faudra que certaines fonctions du module nœud acceptent un paramètre « ensemble de nœuds » pour pouvoir travailler.

2.2.2.1 ACTION : tests du module nœud => ajout d'un nœud/lien à un ensemble de noeud

Le module nœud est au cœur du projet ; il sera complété progressivement au cours des rendus. Pour le rendu1 il y a deux niveaux de vérification :

- Elémentaire : Création d'un nœud à partir de paramètres et ajout du nœud à un ensemble de nœuds
  - On ne peut faire qu'une partie des vérifications car les liens ne sont pas connus.
  - Un tel nœud est créé avec zéro lien par défaut.

- Avancé : Ajout d'un lien à une paire de nœuds appartenant à un ensemble de nœuds

Pour savoir quels test effectuer, voici la liste des fonctions du module **error** telles que disponible dans error.h :

```
// Two nodes have the same uid that is supposed to be unique.
std::string identical_uid(unsigned int uid);

// One of the nodes indicated for a link does not exist.
std::string link_vacuum(unsigned int uid);

// A housing node number of links exceeds the allowed maximum number
std::string max_link(unsigned int uid);

// The same link is defined several times
std::string multiple_same_link(unsigned int uid1, unsigned int uid2);

// A node overlap a link
std::string node_link_superposition(unsigned int uid);

// Two nodes overlap
std::string node_node_superposition(unsigned int uid1, unsigned int uid2);

// A node is using the reserved uid
std::string reserved_uid();

// Everything went well => file reading and all validation checks
std::string success();

// A link refers twice to the same node
std::string self_link_node(unsigned int uid);

// A node that has too little capacity
std::string too_little_capacity(unsigned int capacity);

// A node that has too much capacity
std::string too_much_capacity(unsigned int capacity);
```

Ecrire un module indépendant de test destiné à vérifier d'abord le niveau élémentaire avant de passer au niveau avancé. Ces tests doivent être faits avec des méthodes de type *manipulateur*. Pour cette ACTION, il suffit d'écrire du code qui crée *manuellement* un ensemble de nœuds puis de liens entre ces noeuds. Validez une méthode à la fois. Ré-utilisez aussi les valeurs de position/rayon utilisées pour les tests de l'ACTION 2.2.1.1.

2.2.2.2 ACTION : tests du module nœud => lecture d'un nœud ou d'un lien

Le module nœud est responsable de toutes les tâches de gestion d'un nœud (c'est le cœur de l'approche orienté objet). Cela inclut le décodage d'une chaine de caractère correspondant à une ligne lue dans un fichier. Ici vous devriez effectuer le décodage seulement à l'échelle d'une telle chaîne de caractères (le concept d'*automate de lecture* vu en cours Topic3 est de la responsabilité du module ville). Ce décodage est combiné avec l'appel des méthodes de type *manipulateur*, et s'il y a échec de l'appel de la méthode alors il faut appeler la fonction fournie dans le module **error**.

2.2.3 Module ville

Ce module gère une classe ville. Il est correct de supposer que ce module gère une seule ville dont l'instance unique est déclarée dans l'espace de noms non-nommé du module ville. Son interface doit au moins offrir une fonction de lecture qui est destinée à initialiser implicitement cette unique ville. Cela permet de ne pas avoir à déclarer d'instance de ville en dehors du sous-système du Modèle.

2.2.3.1 ACTION : tests du module ville

Au stade du rendu1, seul le constructeur et la fonction de lecture de fichier sont nécessaires. Dans ce module l'opération de lecture met en place *l'automate de lecture* (cours Topic3) qui filtre les lignes inutiles du fichier, vérifie les nombres annoncés d'éléments (qu'il y a autant de lignes de fichier décrivant des nœuds ou des liens que le nombre indiqué avant ces lignes)[1], et délègue l'analyse fine de lecture d'une ligne au module nœud qui a les compétences pour faire les vérifications nécessaires.

On peut préférer faire les vérifications avancées une fois que l'ensemble des noeuds a été initialisé. Dans ce cas il faut prévoir d'appeler la ou les fonctions de vérification supplémentaires après la fin de la lecture elle-même.

A ce stade l'intégration avec le module projet est immédiate puisqu'il suffit de remplacer le stub de la fonction de lecture du fichier par un appel de la fonction complète du module ville.

3. Forme du rendu1

Pour chaque rendu **UN SEUL membre d'un groupe** (noté **SCIPER1** ci-dessous) doit télécharger un fichier **zip** sur moodle (pas d'email). Le non-respect de cette consigne sera pénalisé de plusieurs points.
Le nom de ce fichier **zip** a la forme : `SCIPER1_SCIPER2.zip`
Exemple pour le groupe Wagnières/Zimmer: `301438_301900.zip`

Compléter le fichier fourni **mysciper.txt** en remplaçant 111111 par le numéro SCIPER de la personne qui télécharge le fichier archive et 222222 par le numéro SCIPER du second membre du groupe[2].

Le fichier archive du rendu1 doit contenir (**AUCUN répertoire**) :

- Fichier texte édité **mysciper.txt**
- Votre fichier **Makefile** produisant un executable **projet**
- votre code source (.cc et .h)

*On doit pouvoir produire l'exécutable **projet** à partir du* Makefile *après décompression du contenu du fichier zip. La commande* **make** *ne doit faire AUCUN déplacement de fichier ; tout reste dans l'unique répertoire créé par la décompression du fichier archive.*

Auto-vérification : Après avoir téléversé le fichier zip de votre rendu sur moodle (upload), récupérez-le (download), décompressez-le et assurez-vous que la commande **make** produit bien l'exécutable et que celui-ci fonctionne correctement.

Exécution sur la VM: votre projet sera évalué sur la VM du cours (c'est la même qu'au premier semestre).

Backup : Vous êtes responsable de faire votre copie de sauvegarde du projet. Il y a un backup automatique seulement sur votre compte myNAS. Sur la VM, vous devez activer vous-même le backup (icone « engrenage » en haut à droite, choisir system settings, choisir backup et activer cette fonction en précisant les paramètres). Une alternative est de s'envoyer la dernière version du code source par email.

Outils possibles (mais pas obligatoires) pour la gestion du code au sein d'un groupe :

- vous pouvez envisager d'utiliser **gdrive.epfl.ch** pour définir un répertoire partagé par les 2 membres du groupe. Cependant, il n'y a pas d'éditeur de code en mode partagé.
- Un tutorial (en anglais) sur l'outil **git** de *gestion de versions de code* est disponible sur moodle. De plus, ce tutorial fait le lien avec l'offre EPFL gitlab.epfl.ch qui permet de *synchroniser le code* d'un groupe (vérifier que ce projet sur gitlab.epfl.ch n'est PAS public).

---

[1] Optionnel : nos fichiers de tests seront corrects de ce point de vue (pas de fonction dans **error**)
[2] L'unique groupe de 3 personnes rajoute le 3ieme SCIPER sur une troisième ligne dans mysciper.txt