Compte-rendu : projet C++

Etudiant : alexandra HULOT
Spécialité : EI-SE
Année : 4

Sommaire

1. Description de l’application développée

“Dans le monde du futur, le réchauffement climatique est devenu non plus un enjeu du lendemain mais du jour même...Bien loin des 2°C des Accords de Paris, la température est devenue la préoccupation numéro 1 du gouvernement. Ce dernier a alors engagé de nombreuses sociétés indépendantes pour trouver une solution rapide et efficace…
Inscrit dans une école d'ingénieurs, vous avez saisi l’occasion pour proposer votre méthode révolutionnaire et montrer que vous n’êtes pas un “bon à rien”. En stage chez une entreprise travaillant pour la cause, vous louez un local dans lequel vous devrez placer des objets refroidissants. Chaque objet a un coût monétaire et énergétique. Il faudra être prudent dans ses choix. La température initiale est de 50 degrés. L’objectif est de descendre en dessous des 25 sans se ruiner.”

Tel est le concept du projet “Your cooling factory”. Du point de vue des contraintes imposées, il est intéressant dans le sens où :

- des sous-ensembles d’objets se développent facilement sur plusieurs niveaux de hiérarchie,
- des objets semblables peuvent facilement se différencier par l’ajout de fonctions virtuelles dans la classe mère : description, dessin,ect.
- 2 objets de même classe peuvent se regrouper. Une première surcharge de l’opérateur+ est alors possible. Ou tout simplement, accéder à un paramètre en particulier (position par exemple) peut se développer avec l'opérateur().

Dans les faits, étant toute seule pour ce projet, tous les objectifs n’ont pas été remplis. Avant d’expliquer en détails ce qui a été fait, commençons par regarder le diagramme UML.

2. Diagramme UML de l'application

(Schéma en annexe et inclus dans le projet)

Le projet est composé d'objets de type Object. Ils ont notamment un prix, un nom, une couleur,etc.
De ces objets, on distingue les objets refroidissants de type CoolingObject, des objets fournisseurs d'énergie de type EnergyPoweredObject.
Un objet refroidissant possède un refroidissement unitaire (par pixel) et se divise également en 2 sous-classes : les objets qui ont besoin d'électricité de type EnergyPoweredObject, et les objets qui n'en ont pas besoin. De ces derniers on détaillera directement les classes spécifiques à chaque objet , comme celle du **Glacoon** par exemple.
De même nous aurons également des classes d'objets telles que **PowerStrip** qui dépendent de EnergySupplyObject , ou encore de **AirConditioner** pour un EnergyPoweredObject.
Bien sûr, nous pouvons en créer d'autres…

Ce qu'il faut noter par rapport à ces classes, c'est leur fonctionnement entre eux :
lorsque le joueur (donné par la classe Player) tente d'acheter un objet de la boutique (ceux en gras ci-dessus) via la méthode buy() - s'il possède assez d'argent, alors il pourra acheter sans aucun problème des Glacoons par exemple. Sinon, s'il tente d'acheter un objet qui nécessite de l'électricité alors il faudra qu'il ait en sa possession un objet de type EnergySupplyObject qui puisse lui fournir l'électricité nécessaire. Si c'est le cas, alors l'appel à la méthode plug() "branche" l'objet à son fournisseur.
De plus, un objet fournisseur d'électricité peut être relié à plusieurs objets (std::map<Energy Powered Objets>) triés entre eux selon une fonction de comparaison de leur demande énergétique. Dans le code final, nous ne nous servons pas tellement de l'utilité d'une std::map mais j'ai trouvé cela intéressant comme tel, puisqu'il permet d'avoir la consommation directe d'un des objets rattachés.

D'ailleurs, si vous regardez en détail le code, beaucoup de fonctions/paramètres ne sont pas utilisés : le nom par exemple, il n'est là que pour donner une description d'un objet. Lorsque j'ai commencé à coder, je n'avais pas encore d'idées très précises de ce que je voulais faire. J'ai alors codé plusieurs éléments qui pourraient s'avérer être utiles.

Encore quelques petites précisions : peut-être auriez-vous déjà vu des getPrice() , getColor() **plusieurs fois** dans des classes dépendant d'une seule classe mère. A ce moment-là, on peut se demander pourquoi ne pas les avoir mises dans la classe mère directement ? Si l'on regarde ces fonctions d'un peu plus près , certaines comportent le mot clé **static.** Pouvoir accéder à certains paramètres sans instancier d'objets de classe est très pratique selon moi (ce pourquoi vous retrouverez ce mot clé un peu partout...). La couleur ou encore le prix sont des paramètres définis par défaut et spécifiques à chaque classe , mais j'ai tenu à ce que ces derniers restent modifiables pour un objet en particulier**, sans en modifier les autres.**
Ainsi les getters que l'on retrouvera dans les classes Glacoon , AirCondioner et PowerStrip ont été créés pour accéder aux paramètres par défaut de la classe, non modifiables.
Tandis que ceux dans la classe Object permettent d'accéder aux attributs propres à l'objet considéré.

Il en va de même pour la méthode toString() et description() : l'une définie les attributs d'une classe, l'autre de l'objet.

Enfin, le programme principal main.cpp est une interface graphique dans laquelle le joueur va interagir avec les objets vus précédemment. Pour la partie graphique j'utilise la librairie SFML, dont l'installation est détaillée ci-après. Je ne pense pas que détailler le fonctionnement d'une telle interface soit intéressante puisque nous ne l'avons pas vu en cours, mais pour résumer :

- le principe d'une interface graphique repose sur la notion d'événements.
- on place des objets dans une fenêtre puis nous scrutons un quelconque évènement (clic gauche, clic sur le bouton 'quit',etc.) auquel correspond une mise à jour (d'un paramètres par exemple) (fermeture de la fenêtre, achat d'un objet,etc.). Puis nous redessinons la fenêtre (pour éviter un clignotement à chaque redessinement, on instancie un renderer).

3. Installation et exécution

Pour ce projet, vous aurez besoin de la librairie SFML comme mentionné précédemment. Pour faciliter son utilisation, la librairie est déjà intégrée au projet. Si vous utilisez éxecutez le code sous windows ou MacOS, il faudra re télécharger la librairie depuis le site officiel : https://www.sfml-dev.org/download/sfml/2.5.1/index-fr.php et sélectionner le

package qui vous correspond. A ce moment-là, il faudra inclure le dossier téléchargé (dézippé) dans le projet et modifier le makefile en ne remplaçant seulement la ligne SFML = SFML-2.5.1 par SFML = <nom_de_votre_dossier> , où <nom_de_votre_dossier> est le nom du dossier que vous venez de télécharger.

Le makefile contient une règle clean et une règle all. Il permet entre autre de créer 2 exécutables : un pour les testcases via la commande make testcase, et un autre pour le main via make main.
Si la compilation prend un petit peu de temps c'est normal…
Lorsque vous exécuterez le main via la commande make ./main, une fenêtre graphique s'ouvrira. Vous pouvez la fermer, elle ne fonctionne pas. Elle n'est là que pour montrer l'idée originelle du jeu.
Enfin, pour exécuter le testcase, tapez ./tests/testcase.

4. Commentaires

Dans ce projet, les parties dont je suis le plus fière, sont notamment :

- la méthode plug() car elle mêle le container std::map et les aspects d'une fonction template. Et pour cause, le code est plus compliqué que la plupart des autres fonctions mais elle ne m'a pas posé tant de soucis. Je pense que cela est dû au fait que j'avais une idée claire et précise de ce que je voulais à ce moment-là.
- l'opérateur operator() (const EnergyPoweredObject& s1,const EnergyPoweredObject& s2)const qui sert de comparateur entre 2 objets du container std::map vu dans le paragraphe précédent. En effet, lors de mes TPs j'avais passé énormément de temps sur la surcharge des opérateurs et les fonctions de comparaisons. Cependant, pour ce projet je n'ai rencontré aucun problèmes avec ses derniers (bien que la fonction reste très simple…)

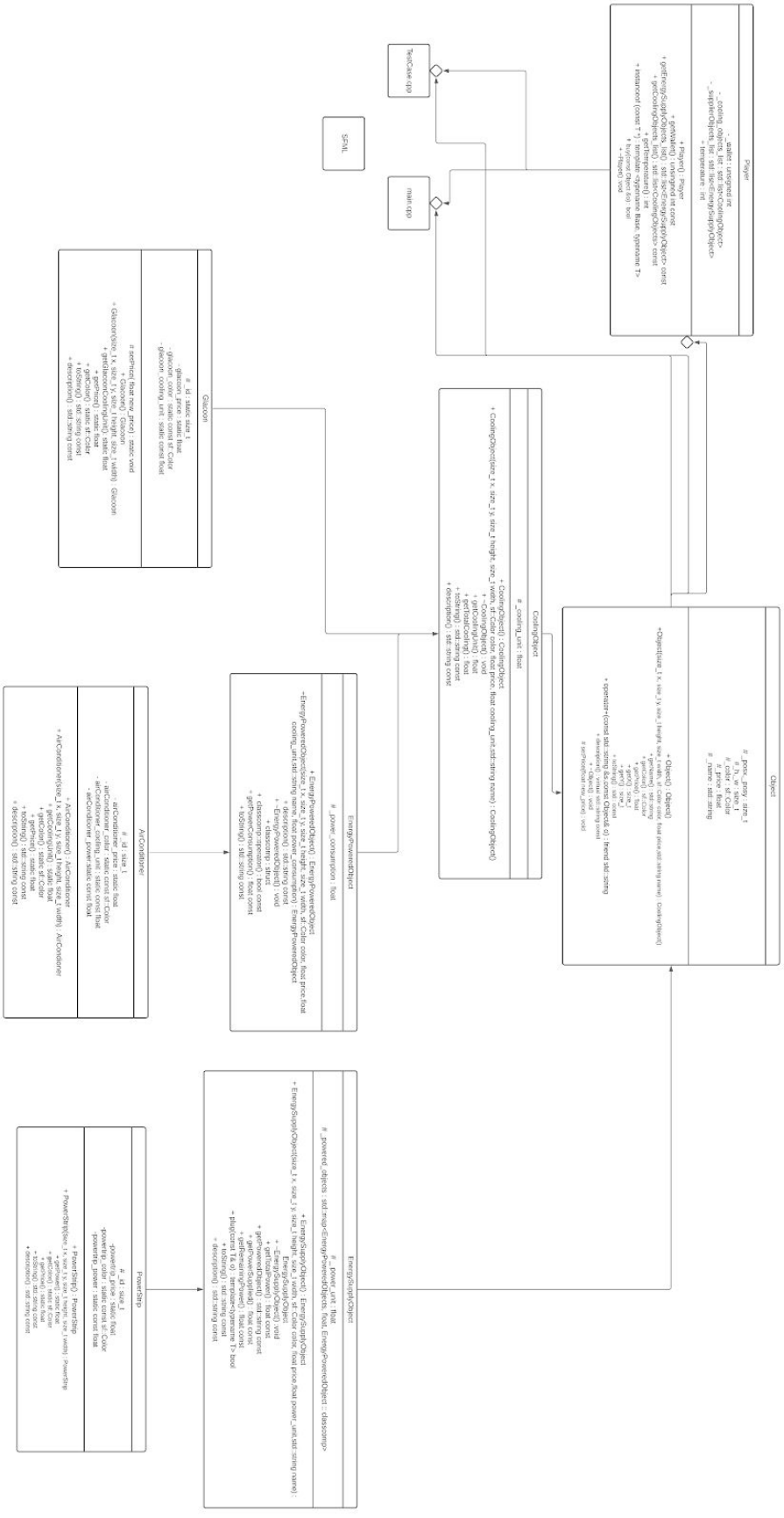
Les parties dont je suis le moins fières sont les suivantes :

- la partie graphique. Elle est pour moi une grande déception. Elle semblait relativement simple dans le sens où le principe n'est pas très complexe et qu'il n'y avait pas vraiment d'images à incorporer (un objet peut se matérialiser en un simple bloc/rectangle). Mais elle ne l'était pas - du moins pas autant….De plus, la partie graphique a été le dernier élément que j'ai codé. Pour bien faire, il aurait fallu la coder en même

temps que le reste, pour éviter qu'elle ne soit trop détachée du reste (et vérifier au passage qu'elle fonctionne). Je ne l'ai pas fait car elle n'était pas dans les contraintes, ce n'était donc pas une priorité.

- la fonction virtuelle draw(), passée en commentaire. Sans elle (reliée à la partie graphique), je n'ai pas d'autres fonctions virtuelles autre que description()...
- l'organisation du code : j'ai la mauvaise habitude de coder dans les fichiers headers, alors qu'il faudrait plutôt faire l'inverse et coder dans les fichiers.cpp. Pour le projet, je n'ai pas eu vraiment le temps de tout réorganiser, d'autant plus que le code principal compile.

5. Annexes


Player
Object
CoolingObject
Glacoon
EnergyPoweredObject
AirConditioner
PowerStrip
EnergySupplyObject
main.cpp
TestCase.cpp
SFML