1 Introduction

1.1 Deux camps irréconciliables?

Dans la plupart des équipes de travail, on remarque souvent que le groupe se divise en deux camps. Le premier camp aime plonger dans les abysses de l'abstraction, discutant pendant des heures de design sans jamais produire une seule ligne de code. Le second camp a de la difficulté à saisir les abstractions. Cependant si vous prenez le temps de montrer un petit fragment de code à ce groupe, la situation devient plus claire en plus de contribuer à l'implémentation.

1.2 Testez tôt l'interface de chaque classe

L'implémentation minimale de l'interface d'une classe devrait s'effectuer tôt au moyen de fonctions partielles (ou « stubs »). Il s'agit en fait de coder les fonctions en insérant non pas les détails complets de la solution mais simplement des messages imprimées et en retournant, lorsque cela est nécessaire (fonction non void) une valeur de retour possible à la fois. On peut ainsi être tester de manière générale le système.

1.3 Développez un prototype d'interface avec l'utilisateur

Il est plus facile de concevoir l'ensemble des fonctionnalités désirées et de laisser les classes inutiles de côté lorsque vous avez une idée des fonctionnalités que l'utilisateur désire. Il n'est pas nécessaire ici de procéder à une implémentation complète de cette interface. Un simple dessin sur une feuille suffit à rendre l'ensemble du système plus clair en plus de nous aider à réduire la quantité de caractéristiques propres à chaque classe. Avant tout, questionnez l'utilisateur.

2 Schématisation : production de l'étape 2 du projet

Le but de l'étape 2 est de tester le fonctionnement global du système afin de nous aider à découvrir d'autres classes. Il sera alors peut-être nécessaire de réviser notre design de base.

Ne traitez AUCUN cas d'erreurs pour le moment ce qui ne veut pas dire que vous devriez les oublier. Simplement, n'embourbez pas le code avec ces détails. Choisissez une méthode simple pour les traiter (un message, une valeur de retour particulière) et soyez constant.

Si vous ne pouvez concevoir une classe en l'incorporant au système, développez cette classe de manière isolée dans une petit fichier. Lorsque vous aurez fait quelques tests, vous pourrez alors envisager de l'introduire dans le système.

Ne concentrez AUCUN effort sur l'interface. Lors de l'étape 3, nous créerons les classes nécessaires à cette finalisation particulière afin de rendre notre système indépendant de la plate-forme.

2.1 Définition C++ du système

Nous pourrions utiliser ici un outil tel que UML pour procéder à cette étape. Notre système est cependant assez petit pour que nous puissions procéder directement.[1]
Cela comporte cependant un danger : celui de voir une équipe développer le système entier sans consulter qui que ce soit. Résistez à la tentation car votre implémentation risque de rencontrer un mur; certains détails ne seront connues que lorsque l'étape 2 sera terminée!
Le programme remit doit être fonctionnel. Voici les exigences pour décrire chaque classe :

- Le nom significatif de chaque caractéristique et son type en mentionnant si cette caractéristique est publique ou privée.
- Le nom significatif de chaque fonction, le type de la valeur retournée (ou void) ainsi que le nom et le type de chacun de ces paramètres. Spécifiez également si cette fonction est privée ou publique.
- Spécifiez les « const » là où cela est nécessaire.
- La description de chaque constructeur et du destructeur s'il y a lieu.
- Uniquement les prototypes des fonctions, constructeurs et destructeurs devraient se retrouver dans la classe.
- La définition des fonctions devraient comporter très peu de détails. Concentrez-vous sur l'essentiel. Le but de l'étape 2 est de tester le fonctionnement global du système afin de nous aider à découvrir d'autres classes.
- La définition du constructeur vous permettra, même si celle-ci est partielle, de cerner l'état initial nécessaire de chaque objet instancié.

2.2 Réutilisation

Il vous est permis de réutiliser les classes que nous avons vues en classe (vecteur, pile, file). Copiez simplement la classe ou mieux utilisez-les en librairie. Veillez à utiliser les versions les plus récentes de ces classes.

2.3 Simulation

Le fonctionnement du programme devrait montrer clairement ce qui se passe. L'interprétation des messages affichée doit permettre à ceux et celles qui n'ont pas vues vos classes de comprendre ce qui se passe.
Faites interargir vos objets pendant un certain temps et conservez les sorties afin que chaque équipe puisse évaluer votre travail.
Il n'est pas nécessaire de TOUT coder! Si vous ne savez pas comment coder quelque chose, le refroidissement d'une benne par exemple, laissez ce problème de côté et contentez-vous de faire exécuter le programme jusqu'à ce que celui-ci manque de bennes! La fonction « rand » peut également être utilisée en attendant qu'une solution se présente…

2.4 Commentaires

VOUS DEVEZ ABSOLUMENT COMMENTER votre code!!! Il sera lu par d'autres.

[1] L'apprentissage de UML ou d'un logiciel tel que Rational Rose (www.rational.com) exigerait à lui seul quelques semaines de cours.

2.5 Remise

Vous remettez le (ou les) fichier(s) CPP (et .H s'il y a lieu) contenant votre schématisation fonctionnelle.
Un fichier texte comportant un exemple de sortie et la manière de l'interpréter afin qu'un lecteur plus ou moins averti (c'est-à-dire, un étudiant) puisse comprendre ce qui se passe.

2.6 Évaluation

Cette partie compte pour 15% de la note finale du travail. Afin de mieux comprendre son importance, voici la pondération qui sera utilisée pour évaluer le travail dans son ensemble : étape 1, 5%, étape 2, 15%, étape 3, 12%, étape 4, 8%.

3 Quelques heuristiques pratiques

Une heuristique n'est pas une règle. Il s'agit simplement d'un conseil qui, la plupart du temps, s'avère extrêmement utile pour guider notre démarche.

3.1 Toutes les données devraient être cachées dans la classe.

L'argument souvent invoqué pour rendre des données publiques s'exprime souvent de la manière suivante : « j'ai besoin de rendre XYZ publique car j'en ai besoin pour effectuer la tâche ABC ». Demandez-vous alors comment la classe dont vous violez les données pourrait effectuer cette tâche pour vous.

3.2 Minimisez le nombre de méthodes publiques dans une classe.

Le problème avec les interfaces trop lourds c'est que vous ne pouvez jamais trouver ce que vous désirez. Vous pouvez ainsi trouver sur Internet une merveilleuse classe implémentant une structure de listes chaînées dynamiques. Cette classe comporte plus de 4000 méthodes dans son interface, un véritable cauchemar.

3.3 Ne placez pas les fonctions qui servent uniquement à regrouper du code commun à plusieurs autres fonctions dans la partie publique.

Les clients de votre classe n'ont pas à voir des méthodes qu'ils ne devraient pas voir de toute façon.

3.4 Conservez les données et leurs modifieurs à un seul endroit.

La multiplication des fonctions du type « get » et plus particulièrement celles du type « set » devrait vous rendre suspicieux. Par exemple, pour une classe implémentant un four de cuisine, on peut songer à une méthode permettant de questionner le four : « es-tu-assez-chaud? » au lieu de procéder avec des « get_température_actuelle » et « get_température_désirée ».

3.5 Cassez une classe en deux si celle-ci contient des informations non reliées entre elles.

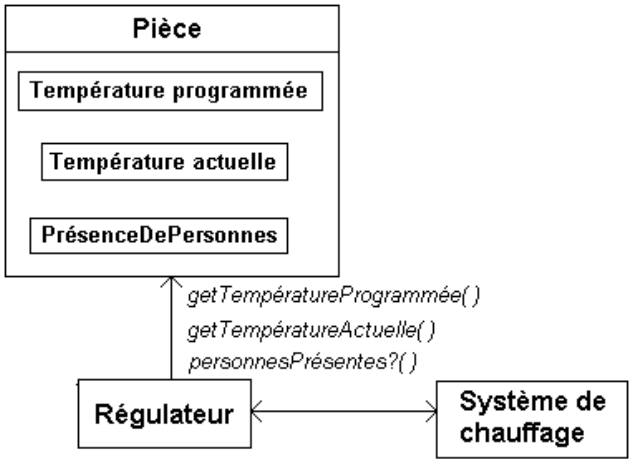
Si un groupe de fonctions n'utilisent qu'un sous-ensemble des données, pensez à regrouper ces données et ces fonctions dans une classe à part.
Une autre solution consiste également, dans certains cas, à procéder par héritage. Nous verrons bientôt comment procéder.

3.6 Méfiez-vous des classes « divines » qui régulent tout.

Lorsque votre classe porte un nom tel que « manager », « boss », « Système »…la classe est soupçonnée de se prendre pour un dieu. Ce genre de classes présente souvent un comportement de type « procédural ». Premier symptôme : la classe procède constamment à des « set » et des « get » sur les autres objets. Deuxième symptôme : elle affiche un comportement anti-social en ne communiquant aucune information. Voici un exemple pour appréhender le concept.

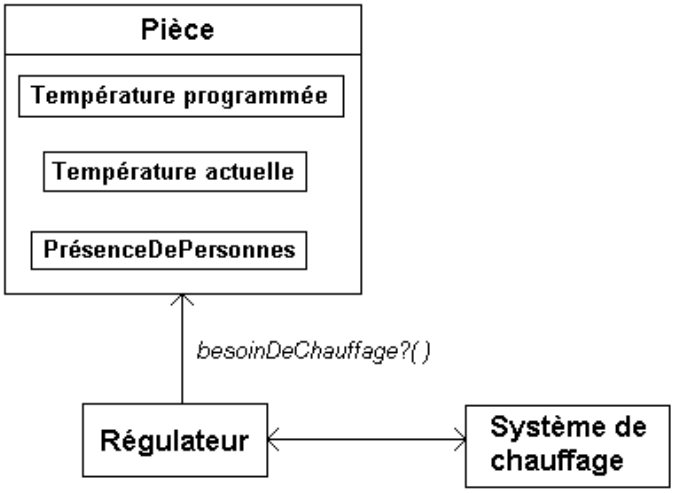
Nous devons implémenter un système permettant de réguler le chauffage dans chaque pièce d'une maison « intelligente ». Chaque pièce est programmable et contient des senseurs permettant de savoir s'il y a quelqu'un dans la pièce. La classe « pièce » contient ainsi les informations suivantes : la température programmée (double), la température actuelle (double) et un drapeau (int) permettant de savoir s'il y a quelqu'un dans la pièce ou non. Chaque pièce est connectée à un régulateur de chaleur qui sert à commander le système de chauffage.

Voici maintenant un mauvais design :



Remarquez comment le régulateur prend la décision d'envoyer ou non de la chaleur dans une pièce : utilisation massive de fonctions d'accès du type « get ».

Voici maintenant un bien meilleur design :



La pièce possède les informations; elle peut donc décider elle-même. Un autre modèle où la pièce demanderait du chauffage au régulateur serait également possible.

3.7 Éliminez les classes inutiles de votre design.

Ces classes comportent souvent un ensemble de données et une panoblie de « get » et de « set ». Lorsqu'aucune action significative n'est réalisée par une classe, celle-ci est inutile.

3.8 Éliminez les classes externes au système ou au niveau d'abstraction que vous désirez solutionner.

Il n'est pas facile de détecter ces classes inutiles. Voici deux cas de conception qui vous permettront d'appréhender le problème.

Premier cas : une compagnie conserve une quantité phénoménale de données sur des moteurs de voiture dont plusieurs types de dynamomètres. Les programmeurs ont la tâche de créer une système permettant de générer des rapports. Les utilisateurs pourraient ainsi décrire la forme du rapport, puis mentionner les résultats de test qu'ils désirent y voir apparaître. Lors du design initial, les programmeurs se mirent à créer la belle classe « dynamomètre ». Il s'agit d'un très bel objet. Cependant, dans le contexte du problème à résoudre, le fonctionnement d'un dynamomètre n'est pas du tout pertinent.

Deuxième cas : une compagnie se spécialise dans la vente, par Internet, d'une panoplie de petits accessoires de cuisine : mélangeurs, grille-pain etc. Cette compagnie désire créer un système de gestion d'inventaire. Dans ce cas précis, les programmeurs doivent évidemment résister à la tentation de créer la classe « mélangeur », ou la classe « grille-pain » sous prétexte que ce type d'objets effectuent plusieurs opérations dans la vie courante (ex. le mélangeur coupe, hache, réduit en purée). Ces comportements ne sont pas toujours pertinents au problème à résoudre.

3.9 Des classes « agents » sont parfois nécessaires.

Voici deux histoires.

- Dans une ferme orientée-objet, il y avait une vache orientée-objet avec un certain lait orienté-objet. Le dilemme était le suivant : est-ce que la vache orientée-objet devait envoyer un message au lait orientée-objet lui demandant de « se dévacher lui-même », ou est-ce que le lait orientée-objet devait envoyer un message à la vache orientée-objet lui demandant de « se traire elle-même »?
- Dans une librairie orientée-objet, il y avait un livre orienté-objet et une certaine étagère orientée-objet. Le dilemme était le suivant : est que le livre orienté-objet devait envoyer un message à l'étagère orientée-objet lui demandant de « replacer le livre sur elle-même » ou est-ce que l'étagère orientée-objet devait envoyer un message au livre orienté-objet lui demandant de « se placer lui-même »?

Les classes « agent » recoivent des messages d'un objet et le transmettre à un autre. Parfois, on se rend compte que ces classes sont inutiles mais il faut tout de même considérer leur présence possible. Si la classe agent est imaginée au départ, on pourra

l'éliminer éventuellement si elle ne développe pas d'autres fonctionnalités propres que celles de transmettre des messages.
Dans le cas de la vache et du lait, une classe agent importante intervient : le fermier orienté-objet. Dans le cas du livre et de l'étagère, la classe libraire intervient. Les deux sont chargées de recevoir et transmettre des messages et de plus, possèdent certainement un ensemble de comportements qui leur sont propres.

3.10 Minimisez le nombre de classes avec lesquelles une autre classe collabore.

Lorsque cette heuristique est violée, on voit alors apparaître une quantité incontrôlable de petites classes insignifiantes s'utilisant les unes les autres à qui mieux mieux.
Lorsque le système se développe ainsi, demandez-vous si vous pouvez remplacer un groupe de classes par une seule qui contiendrait le groupe de manière à réduire les collaborations.

3.11 Si une classe en contient une autre alors la classe contenant devrait utiliser les classes qu'elle contient.

La classe proprement « conteneur » (ex. pile, file, vecteur, etc.) constitue une exception possible de cette heuristique.

3.12 Une classe ne devrait pas contenir plus de données que le développeur est capable de conserver en mémoire!

La valeur 7±2 est souvent mentionnée.Ce nombre magique a été proposé en 1956 par G.A. Miller dans son article « The magical number 7, plus or minus two : some limits on our capacity for processing information » (Psychological Review N° 63). Ce nombre est désormais mentionné par une quantité phénoménale de personnes bien intentionnées qui ne savent même plus d'où provient cette affirmation (en bref, aucun n'a lu l'article en question!). Au lieu de dire des imbécillités, contentons-nous de tester notre mémoire de la manière suivante : s'il vous faut quelques secondes pour décrire l'ensemble des caractéristiques d'une classe et que parfois, vous hésitez, c'est qu'il y en a trop ou que leur nom est non significatif.

3.13 Une classe doit connaître ce qu'elle contient. Cependant, elle n'a pas à connaître qui la contient.

Une classe ne devrait donc jamais être dépendante de son utilisation à l'intérieur d'une autre. Par exemple, la classe « chaise » n'a pas à savoir qu'elle est utilisée par un joueur de dés ou un professeur d'université. Ainsi, l'entrepôt n'a pas à savoir qu'elle est utilisée par la salle d'électrolyse ou la salle de scellement.

4 Liste de contrôle pratique pour le concepteur d'une classe

Cette liste vous permettra de construire vos classes de manière professionnelle. Pour chaque classe créée, passez au travers de cette liste. Cette vérification vous permettra de créer immédiatement des classes professionnelles et sans surprise pour la suite. Certaines classes complexes nécessitent des réponses affirmatives à l'ensemble des questions tandis que des classes plus simples pourraient nécessiter une majorité de réponses négatives.

4.1 Est-ce que votre classe possède un constructeur?

Une classe qui ne nécessite pas de constructeur est suspecte ou insignifiante. Assurez-vous que l'état initial de toutes vos classes est cohérent dans le cadre du système.

4.2 Les données de la classe sont-elles privées?

Les données publiques sont toujours suspectes. Si vous acceptez que des données soient modifiées par le programmeur-client, assurez-vous que ces modifications soient effectuées au travers des fonctions d'accès.

4.3 Est-ce que votre classe possède un constructeur sans argument?

Si vous désirez que le programmeur-client puisse créer des tableaux contenant des objets de votre classe, assurez-vous de placer ce constructeur dans la partie publique et définissez-le. Autrement, placez simplement le prototype dans la partie privée et ne le définissez que si vous, en tant que programmeur-concepteur en avez besoin.

4.4 Est-ce que chaque constructeur initialise toutes les données de la classe?

Un nouvel objet instancié devrait toujours posséder un état cohérent dans le système. Si une donnée n'est pas initialisée, attendez-vous à tout.

4.5 Est-ce que votre classe exige un destructeur?

Ce n'est pas toutes les classes qui exigent un destructeur. Cependant, si votre classe alloue dynamiquement de la mémoire au moyen de l'opérateur « new », il est évident que cette classe nécessite un destructeur pour libérer cette mémoire.

4.6 Est-ce que votre classe exige un destructeur virtuel?

Certaines classes nécessitent un destructeur virtuel simplement pour mentionner que leur destructeur est virtuel! Bien sûr, une classe qui n'est pas une classe de base n'a pas besoin de destructeur virtuel : les fonctions virtuelles ne sont utiles que si nous sommes en présence d'héritage, sinon oubliez cet item. Nous verrons bientôt en quoi cela consiste.

4.7 Est-ce que votre classe exige un constructeur de copie?

La réponse est souvent, mais pas toujours, « non ». Il faut simplement se demander s'il y a une différence entre le fait de copier un objet appartenant à cette classe et le fait de copier individuellement les données de cette classe. S'il n'y a pas de différence (ex. la classe « PaquetDes »), le constructeur de copie fourni gratuitement en C++ est suffisant, sinon (ex. la classe « vecteur »), il vous faut absolument coder un constructeur de copie. En somme, si votre classe effectue des allocations dynamiques de mémoire, vous devriez coder une constructeur de copie.

4.8 Est-ce que votre classe exige un opérateur d'assignation?

Si votre classe nécessite un constructeur de copie, il y a toutes les chances qu'elle exige également un opérateur d'assignation. Si vous désirez que le programmeur-client ne puisse pas effectuer des assignations, placez le prototype de l'opérateur d'assignation dans la partie privée de la classe.

4.9 Est-ce que votre opérateur d'assignation est capable de traiter correctement l'assignation d'un objet sur lui-même?

Si votre classe nécessite un opérateur d'assignation, testez-le afin de voir s'il traite correctement le cas limite suivant :

```
UneCertaineClasse yoyo;
yoyo = yoyo;
```

4.10 Est-ce que votre classe nécessite des opérateurs relationnels?

Est-il logique de comparer deux objets de votre classe avec l'aide des opérateurs ==, !=, <=, etc. Si la réponse est « oui », surchargez les opérateurs nécessaires.

4.11 Avez-vous pensé à utiliser l'opérateur « delete[] » lorsque vous libérez un tableau dynamique?

C'est une erreur fréquente que d'oublier les « [] » lorsque la mémoire d'un tableau dynamique est libérée.

4.12 Lorsqu'une fonction possède des arguments passés par référence, devraient-ils être spécifiés « const »?

Le seul cas où une fonction devrait avoir des arguments passés par référence sans « const » est lorsque cette fonction doit modifier ses argument.

4.13 Avez-vous pensé à utiliser « const » devant les arguments du constructeur de copie et de l'opérateur d'assignation?

Dans certains livres de C++, on suggère les prototypes suivants pour les constructeurs de copie et l'opérateur d'assignation :

```
UneCertaineClasse::UneCertaineClasse(UneCertaineClasse &);
UneCertaineClasse::operator=(UneCertaineClasse &);
```

Cela est incorrect. Après tout, le fait de copier un objet ne devrait jamais modifié l'original. De fait, cette manière de procéder nous empêche de copier des expressions simples.

4.14 Avez-vous pensé à déclarer « const » les fonctions qui doivent l'être?

Lorsqu'une fonction appartenant à la classe garantie qu'elle ne modifie pas l'objet, on la déclare « const » de manière à pouvoir l'utiliser sur des objets constants. Les fonctions d'accès qui retourne simplement la valeur d'une donnée devraient toujours être spécifiées « const ». Ainsi, dans la classe « PaquetDes », on trouverait :

```
int vert(long k) const;
int rouge(long k) const;
```

Cette liste est présentée pour vous aider à rendre vos classes uniformes et lisibles par tous en plus de vous assurez que la réutilisation de vos classes sera possible.