Les enjeux de la conception logicielle

A ce niveau du cours, vous avez toutes les connaissances nécessaires pour créer n'importe quelle application. En effet, pour réaliser un calcul, un programme a besoin de :

- manipuler les données (avec les variables) ;
- réaliser des calculs dessus ;
- pouvoir écrire des algorithmes (avec les tests et les boucles).

Ce sont les fonctionnalités de base, que de nombreux langages de programmation fournissent. Les autres fonctionnalités d'un langage servent en fait à faciliter la conception d'application de plus en plus complexes.

Il est important de bien comprendre les enjeux de la conception d'applications complexe, pour utiliser correctement les outils qu'un langage met a disposition.

Vous avez déjà vu un outil visant à faciliter la conception des applications : les fonctions. Ce chapitre utilisera donc les fonctions comme exemple, mais comprenez bien que cela est applicable aux outils que vous verrez dans l'avenir, en particulier la programmation orientée objet.

Quel est le problème avec la conception d'applications complexes ? La réponse est évidente : elles sont complexes ! A cause de leur taille. A cause des besoins ou des contraintes. A cause de leurs objectifs.

De nos jours, une application sera facilement développée en équipe, sera maintenu pendant plusieurs années, devra fonctionner sur plusieurs systèmes, répondre a des contraintes de coûts de développement et de maintenant, des contraintes de fiabilité. Il faut maîtriser tous ces aspects pour obtenir une application respectant la qualité logicielle :

- la capacité fonctionnelle : le programme fait ce qui est attendu ;
- la fiabilité : le programme fournit le bon résultat ;
- la maintenabilité : il est facile de corriger les erreurs, et d'ajouter de nouvelles fonctionnalités (évolutivité) ;
- la performance : le programme travaille efficacement ;
- la portabilité : le programme est utilisable sur différentes plateformes ;
- la facilité d'utilisation : les programme est facile à utiliser.

Décomposer un problème complexe

Les capacités humaines sont limitées. (Oui, même vous qui pensez être plus intelligents que les autres). Une approche classique pour résoudre un problème complexe est de le décomposer en sous-problèmes un peu moins complexes. Puis diviser ces sous-problèmes en problèmes encors plus simples, et ainsi de suite, jusqu'a obtenir des problèmes que vous savez résoudre.

La décomposition en sous-problèmes peut s'exprimer directement avec des fonctions : chaque sous-probleme donné une fonction. Une décomposition en sous-probleme donné une liste d'appels de fonctions.

```
void problem_1() {
    // résolution du probleme 1
}

void problem_2() {
    // résolution du probleme 2
}

void problem_3() {
    // résolution du probleme 3
}

void complex_problem() {
    problem_1();
    problem_2();
    problem_3();
```

```
}
```

C'est aussi simple que cela. Une problématique bien comprise sera naturellement décomposable en fonctions. Chaque fonction devrait être relativement simple, en général au maximum quelques dizaines de lignes. (Plusieurs guides de codage donne le chiffre - arbitraire - de 25 lignes. Ce n'est pas une contrainte absolue, mais c'est une bonne base d'essayer de respecter une telle limite).

Il faut bien comprendre un point : le découpage en fonctions doit découler naturellement de la décomposition du problème. Découper une code en plusieurs fonctions, sans organisation logique, juste parce que “en C++, on utilise des fonctions” n'a aucun sens.

Lisibilité du code

Il faut bien garder en mémoire qu'un code est plus souvent lu qu'il n'est écrit. Il faut donc qu'il soit le plus lisible possible. Quand une autre personne lit votre code, elle devrait pouvoir comprendre comment votre probleme se décompose, rien qu'en lisant l'organisation de vos fonctions. (Les noms que vous donnez à vos fonctions et variables sont donc primordiaux).

Un facteur important pour la lisibilité est la *localité des informations*. Lorsque vous lisez un code, le cerveau conserve les informations lues dans une mémoire temporaire et ces informations sont utilisées pour comprendre la suite du code. Cette mémoire temporaire a une capacité limitée, ce qui fait que lorsque de nouvelles informations arrivent, les anciennes informations sont oubliées. Si vous avez besoin ensuite de ces informations, il faudra donc aller relire le code déjà lu (perte de temps) ou prendre le risque de ne pas comprendre totalement.

Pour comprendre correctement une ligne de code, il faut que les informations nécessaire à sa compréhension soient en mémoire. Plus les informations sont locales (c'est-à-dire proche de la ligne de code à comprendre), plus un code sera facile a lire.

Une règle (que vous avez déjà vue) est de déclarer une variable au

moment ou vous en avez besoin. (En plus, cela permet souvent d'ajouter `const` dans les déclarations).

Le code suivant :

```
void foo() {
    int i { 1 };
    int j { 2 };
    int k { 3 };

    f1(i);
    f2(j);
    f3(k);

    g1(i);
    g2(j);
    g3(k);

    h1(i);
    h2(j);
    h3(k);

    bar(i, j, k);
}
```

sera moins lisible que le suivant :

```
void foo() {
    int i { 1 };
    f1(i);
    g1(i);
    h1(i);

    int j { 2 };
    f2(j);
    g2(j);
    h2(j);

    int k { 3 };
    f3(k);
    g3(k);
```

```
    h3(k);

    bar(i, j, k);
}
```

(Bien sur, sur un code aussi simple et aussi court, vous ne verrez pas de différence de lisibilité. Cette règle aura un impact lorsque le code devient un peu important).

La décomposition d'un code en fonctions aide également à améliorer la lisibilité du code, en isolant chaque partie du code qui est indépendante des autres. Dans le code précédent, vous voyez qu'il y a trois parties de code indépendantes, puis une partie qui dépendent des autres lignes. Il est donc possible de découper le code en plusieurs fonctions pour montrer cette indépendance dans le code.

```
int foo_1() {
    int i { 1 };
    f1(i);
    g1(i);
    h1(i);
    return i;
}

int foo_2() {
    int j { 2 };
    f2(j);
    g2(j);
    h2(j);
    return j;
}

int foo_3() {
    int k { 3 };
    f3(k);
    g3(k);
    h3(k);
    return k;
}

void foo() {
```

```
    const int i { foo_1() };
    const int j { foo_2() };
    const int k { foo_3() };

    bar(i, j, k);
}
```

(Encore une fois, sur un code aussi simple, la différence de lisibilité n'est pas significative. Cela est important dans des codes plus complexes.)

Dans le premier code, pour comprendre l'appel de la fonction `bar`, il faut lire les déclarations des variables `i`, `j` et `k`, qui sont éloignées de l'appel de `bar`. (Imaginez si la fonction `foo` faisait plusieurs centaines de lignes de code, il faudrait faire défiler le code pour aller chercher la déclaration des fonctions).

Dans le dernier code, la déclaration des variables utilisées dans la fonction `bar` sont plus proches, ce qui a un impact sur la lisibilité. (Dans ce cas, même si la taille du code est plus importante, la déclaration des variables reste proche de leur utilisation).

Les abstractions

Prenez un code d'exemple simple :

```
std::string s { "hello, world" };
std::cout << s.size() << std::endl;
```

Vous devriez comprendre (normalement, si ce cours n'est pas trop mal conçu) que ce code affiche la longueur de la chaîne de caractères “hello, world”. Vous le savez parce que vous savez que `std::string` représente une chaîne de caractères, que celle-ci est initialisée avec la chaîne “hello, world”, que `std::cout` permet d'afficher quelque chose, et que la fonction membre `size()` retourne la taille de la chaîne.

Pourtant, si quelqu'un vous demande ce que fait exactement chacune des étapes de ce code, vous seriez incapable de répondre. Rien que le fonctionnement interne de `std::cout` est relativement complexe. Et pourtant, cela ne vous empêche pas de comprendre ce code, et même

d'écrire vos propres code utilisant ces fonctionnalités.

La raison est que vous avez juste besoin de savoir comment utiliser ces fonctionnalités, pas le détail de ce qu'elles font en interne. Ce qui vous intéresse, c'est abstraction qu'elle représente : `std::string` représente une chaîne, `std::cout` permet d'afficher, etc.

Le découpage du code en fonction permet donc de créer des abstractions qui facilite la lecture du code. En écrivant une fonction qui peut se comprendre sans lire son code, vous renforcer la localité des informations et donc la lisibilité.

La partie publique d'une fonctionnalité est son **interface**. Pour une fonction, c'est donc sa signature (son nom et la liste de ses paramètres). Pour une classe, c'est l'ensemble de ses fonctions membres (par exemple, la classe `std::vector` contient les fonctions membres `size`, `push_back`, `front`, etc.).

Les détails interne d'une fonctionnalité est son implémentation. Pour une fonction, il s'agit du code de cette fonction. Pour une classe, c'est l'ensemble des variables membres et l'implémentation des fonctions membres. Il y a quelques subtilités sur l'implémentation des classes en C++, mais vous verrez cela dans les chapitres sur la programmation orientée objet.

Vous verrez dans les prochains chapitres qu'il est possible (et même recommandé quand c'est possible) de séparer l'interface et l'implémentation des fonctionnalités dans des fichiers différents, ce qui permet bien distinguer les deux. Cela est détaillé dans le prochain chapitre.

La création d'abstractions simples à utiliser (mais remplissant des tâches parfois très complexes) est l'un des buts principaux de la programmation orientée objet et de la méta-programmation. Vous verrez également cela par la suite.

La réutilisabilité

La réutilisabilité du code est quelque chose que vous utiliser depuis le début de ce cours, sans forcément le réaliser. Ce terme signifie simplement que le même code est *peut être* réutiliser plusieurs fois. Par exemple, quand vous utilisez des fonctionnalités de la bibliothèque standard, vous réutilisez le même code (écrit par quelqu'un d'autre que vous) plusieurs fois.

Il faut bien comprendre qu'avoir un code réutilisable est différent de réutiliser un code. Un code réutilisable est un code *conçu* pour pouvoir être réutiliser plusieurs fois (qu'il soit effectivement réutiliser ou non). Si vous concevez un code non réutilisable, vous avez toutes les chances qu'il ne puisse pas être réutilisé. Ecrire un code réutilisable demande un peu plus de travail, mais le gain de temps sur le long terme est plus intéressant. Vous ne pouvez pas savoir à l'avance que vous n'aurez pas besoin de réutiliser un code.

L'impact d'une bonne conception sera plus important sur un code de taille important, qui sera maintenu longtemps, avec une équipe de taille importante. Travailler sur des petits codes (en particulier les exercices réalisés lors de l'apprentissage) ne permettent pas de se rencontre compte de l'importance d'une bonne conception. Durant votre apprentissage d'un langage, il faut accepter de suivre les règles de conception, même si vous ne comprenez pas leur importance.

Le copier-coller

Pour savoir si un code est réutilisable, il faut se poser la question de comment il pourra être réutilisé et quels seront les freins à sa réutilisation.

Une approche “historique” est de simplement copier-coller un bout de code (“snippet”) lorsque vous en avez besoin. Certains éditeurs de code proposent même des fonctionnalités pour réaliser automatiquement cette tâche. C'est particulièrement utile par exemple pour ajouter la

licence logicielle en en-tête des fichiers, la documentation technique du code ou un modèle de code pour une classe.

L'inconvénient de cette approche est qu'ensuite les modifications doivent être fait pour chaque bout de code que vous avez coller. Par exemple, si vous écrivez une boucle `for` avec des itérateurs pour parcourir une collection et que vous copiez ce code a une centaine d'autres endroits dans votre code. Puis, quelques mois plus tard, vous changez d'avis et voulez remplacer cette boucle `for` par une boucle `range-for`. Comment faire ca simplement ? C'est assez difficile : il faut retrouver tous les endroits ou vous avez copier votre code, puis le corriger. Vous risquez donc d'oublier des endroits ou de vous trompez en corrigeant le code.

Le copie-coller de code est donc possible pour le code non critique (commentaires, documentation, etc) mais est à éviter dans les autres cas.

Les macros

Un moyen d'éviter le problème de maintenance d'un code que vous copiez-collez est d'avoir une seule version d'un code et de faire appelle a celui-ci quand vous en avez besoin.

Une méthode historique simple est de définir une macro qui contient votre code, puis d'appeler plusieurs fois cette macro. (Cette approche est en particulier utilisée en C pour écrire du code générique).

```
#define print(x) std::cout << x << std::endl

int main() {
    print(123);
    print("hello");
}
```

Les macros permettent certaines fonctionnalites specifiques, donc il est parfois intéressant de les utiliser. Mais elles presentent egalement de gros problèmes (en particulier que c'est du remplacement de chaînes sans tenir compte du contexte et qu'il n'y a pas de vérification des types

de paramètres). Il est donc préférable de limiter leur utilisation et préférer les fonctions (et les templates pour le code générique).

Les fonctions

Les fonctions sont la base de la réutilisabilité en programmation procédurale. C'est même pour cela que les fonctions ont ete initialement creees.

Vous connaissez déjà les fonctions, mais voici un petit rappel de ce qu'elles apportent en termes de réutilisabilité :

- isoler un traitement de données. Une fonction doit faire une chose et le faire correctement (principe de responsabilité unique).
- documenter un traitement. Le nom de la fonction et des paramètres doivent permettre aux utilisateurs de savoir ce que fait une fonction et comment l'utiliser. Si nécessaire, il est possible de compléter en ajoutant des commentaires et de la documentation.
- imposer une interface. Une fonction prend un certain nombre de paramètres, ayant des types définis. L'utilisateur doit respecter la signature de la fonction, sous peine que le compilateur rejette le code. Une bonne interface est une interface qui facilite l'utilisation correcte de la fonction et décourage les utilisations incorrectes.
- écrivez du code générique. Plus un code sera generique, plus il sera utilisable dans de nombreuses situations. Par exemple, si vous écrivez une boucle `for` utilisant l'opérateur `[]`, votre code ne sera utilisable qu'avec des tableaux, pas toutes les collections. Avec des itérateurs ou une boucle `range-for`, n'importe quelle collection sera utilisable. N'ajoutez pas de contraintes si elles n'ont pas lieu d'être.

Les classes (qui seront détaillées dans la partie Programmation Orientée Objet) est une extension de cette approche. Au lieu de simplement isoler un traitement de données dans des fonctions, les classes permettent d'isoler des données et leurs traitements associées, pour former un tout cohérent (et donc plus simple à réutiliser).

Plus généralement, créer une bibliothèque logicielle permet de faciliter la réutilisation du code. C'est pour cela qu'une partie importante de la conception se focalise sur la création de bibliothèques logicielles, une application n'étant au final qu'un ensemble de bibliothèques qui interagissent entre elles.

Pourquoi un code réutilisable n'est pas réutilisé ?

Malgré le soin que vous pouvez apporter pour que votre code soit le plus réutilisable possible, il peut arriver que votre code ne soit en pratique pas (ou peu) réutilisé. Essayer de comprendre pourquoi les utilisateurs ne réutilisent pas votre code peut aider à améliorer sa réutilisabilité.

Le premier blocage peut être simplement que les utilisateurs ne savent pas que votre code existe. Si vous travaillez sur un projet seul, pendant “que” quelques semaines, et qui contient “que” quelques milliers de ligne de code, il sera possible de ne pas oublier qu'un code existe déjà. Sur un projet en équipe, sur plusieurs années, avec plusieurs millions de ligne de code, il sera facile de ne pas savoir qu'un code existe déjà.

Pour éviter cela, plusieurs pistes sont possibles :

- organisez correctement vos projets. Décomposez en modules facilement identifiables, créez des répertoires pour ranger vos fichiers (l'organisation en modules et répertoires sera détaillé dans un prochain chapitre).
- écrivez de la documentation techniques, expliquant l'organisation du projet, les informations importantes à connaître, comment vous avez conçus le code. (Plus les utilisateurs comprendront comment vous avez pensé votre code, plus ils

arriveront à trouver facilement les informations dont ils ont besoin).
- transmettez vos connaissances et faites en sorte que les autres développeurs partagent aussi leurs connaissances. Une technique classique pour cela est de faire du *peer-reviewing*, qui consiste à se relire mutuellement le code entre développeurs. (Cela sera détaillé dans le chapitre sur la gestion de projet).

Pour résumer, la communication entre développeurs est la base de la reutilisabilite. N'attendez pas simplement que les développeurs communiquent, mettez en place à l'avance les outils et méthodes facilitant la communication.

En particulier, réévaluer constamment les outils et méthodes. Demandez vous si les méthodes sont bien comprises et bien appliquées. Si les outils sont efficaces. Si les besoins ne changent pas. Soyez “Agile”. (La mthode de gestion de projet “Agile” sera également développé dans un prochain chapitre).

La testabilité

La testabilité d'un code correspond a la facilite ou non de tester un code. Comme pour la réutilisabilité, il faut distinguer un code testable et un code testée. La testabilité est une propriété intrinsèque d'un code, qu'il soit effectivement testé ou non.

Tests automatiques et manuels

Les tests ont une importance particulière dans certaines méthodes de gestion de projets, en particulier dans les méthodes Agile, qui sont utilisées dans ce cours. Plus spécifiquement, vous verrez dans les prochains chapitres le développement dirigé par les tests (TDD, pour *Tests Driven Development*), qui consiste à écrire en premier les tests et les utiliser comme point de départ pour écrire votre code.

Il existe plusieurs types de tests, en fonction de ce que vous voulez tester et comment vous tester.

- les tests automatiques sont réalisés par des programmes, qui suivent un ensemble de tâches prédéfinies (appeler une fonction en utilisant des valeurs spécifiques, lire un fichier, telecharger des données en ligne, etc) puis vérifient que le résultat obtenu est conforme à ce que vous avez prévu.
- les tests manuels sont réalisés par des personnes, qui suivent une liste de tâches prédéfinies et vérifie le résultat obtenu.

Les tests automatiques sont tres rapides a executer, alors que les tests manuels sont limités par la rapidité de celui qui fait les tests. (Et plus une personne se précipite, plus il risque de faire des erreurs). Il est donc préférable de privilégier les tests automatiques en priorité, mais ce n'est pas toujours possible. (Par exemple, si vous écrivez un programme qui va rechercher une image de chaton sur internet, il faudra bien qu'un humain vérifie l'image pour etre sur que c'est bien un chaton).

Pour simplifier, plus il sera facile d'écrire des tests automatiques, plus un code sera testable.

Tests unitaires

Un autre critère important de la testabilité est la granularité du code (c'est-a-dire a quelle point vous pouvez tester des parties minimalistes de votre code). Prenez un exemple simple : imaginez que vous devez écrire un code qui réalise deux tâches successives et retourne un résultat.

```
int do_something() {
    int result { 0 };

    // code pour la tâche 1
    ...

    // code pour la tâche 2
    ...
```

```
    return result;
}
```

Vous devez tester si vous code est valide ou non.

Dans cet exemple simple, le résultat attendu sera toujours le même, donc il est possible d'écrire un test qui appelle cette fonction et compare le résultat retourné. Par exemple :

```
int main() {
    const auto result { do_something() };
    assert(result == expected_result);
}
```

Ce simple code est un test : si vous exécutez ce code, si l'assertion est fausse, c'est que la fonction ne retourne pas le résultat attendu, le test a échoué. (En pratique, un vrai test est un peu plus complexe, vous testerez plus de choses en même temps. Et comme vous ferez beaucoup de tests en même temps, un outil spécialisé se chargera de regrouper et présenter les résultats des tests, pour faciliter l'analyse des tests. Mais le principe de base est le meme).

Si le test échoue, cela va nécessiter une correction du code de votre part. La question est laquelle des deux tâches est incorrecte ? La premiere ? La seconde ? Les deux ?

En fait, ce test ne permet pas de répondre à ces questions, parce qu'il ne teste pas séparément les deux tâches. On dit que le test n'est pas “unitaire” (il ne teste pas une chose unique). Plus un test est unitaire, plus il sera facile d'identifier la source d'une erreur et corriger le code.

Il est facile de corriger le code précédent, en respectant le principe de responsabilité unique (SRP), c'est a dire en faisant une fonction pour chaque tâche.

```
void task_1(int& value) {
    // code pour la tâche 1
    ...
}
```

```
void task_2(int& value) {
    // code pour la tâche 2
    ...
}

int do_something() {
    int result { 0 };
    task_1(result);
    task_2(result);
    return result;
}

int main() {
    // test de la tache 1
    int result_1 { 0 };
    assert(task_1(result_1) == expected_result_1);

    // test de la tache 1
    int result_2 { 0 };
    assert(task_2(result_2) == expected_result_2);

    // test de la tache 1
    const int result { do_something() };
    assert(result == expected_result);
}
```

Vous voyez qu'avec ce code, selon quelle assertion échoue, vous pourrez retrouver facilement quelle tâche n'est pas correcte.

C'est le second critère de la testabilité : plus il sera facile d'écrire des tests unitaires, plus le code sera testable.

Un code peu testable sera un code qui demandera plus de travail pour vérifier qu'il est correct et le maintenir. Voire ca sera un code qui a plus de risque de contenir des erreurs.

Pour aller plus loin : Software testability.

Classes, modules et bibliothèques logicielles

Un dernier critère important est le découpage du code. Si vous écrivez un code d'une dizaine de lignes, vous (ou un autre développeur) n'aurez probablement pas de mal à relire et comprendre votre code. Si vous écrivez une centaine de lignes de code, cela sera plus difficile. Si vous écrivez plusieurs milliers de lignes de code, vous aurez du mal a comprendre votre code.

Les fonctions permettent de découper un code en partie plus courtes et plus simples a lire. En particulier, en donnant des noms aux fonctions et aux paramètres. Mais si votre code est conséquent, vous pourrez obtenir plusieurs milliers de fonctions et votre code redeviendra peu compréhensible.

De nos jours, les programmes sont de plus en plus important en taille, développés par des équipes nombreuses, maintenu sur des années. Le découpage en fonctions n'est alors plus suffisant, il faut des niveaux supérieurs d'organisation du code :

- une classe regroupe plusieurs fonctions et structures de données ;
- un module regroupe plusieurs classes ayant une thématique commune (gestion des fichiers, réseaux, interface graphique, etc) ;
- une bibliothèque logicielle regroupe un ou plusieurs modules en projets, qui peuvent être développés de façon indépendante ;
- un framework regroupe plusieurs bibliothèques logicielles ;
- un programme va utiliser plusieurs bibliothèques logicielles pour remplir un ou plusieurs tâches spécifiques ;

L'application finale que vous proposerez aux utilisateurs pourra donc être un ensemble complexe d'applications qui interagissent en eux, de bibliothèques logicielles, de ressources (images, fichiers de données), etc.

Ce qui implique qu'il faudra également gérer la création de *distribuables* (c'est-à-dire de “quelque chose” qui permettra aux utilisateurs d'accéder à vos applications : installation, mise à jour, fournir les sources, etc).

Pour terminer, en complément de cette organisation logique de votre code en fonctions, classes, modules, etc. il faudra également gérer l'organisation physique de votre code dans des fichiers et dans des répertoires.

Chapitre précédent	Sommaire principal	Chapitre suivant