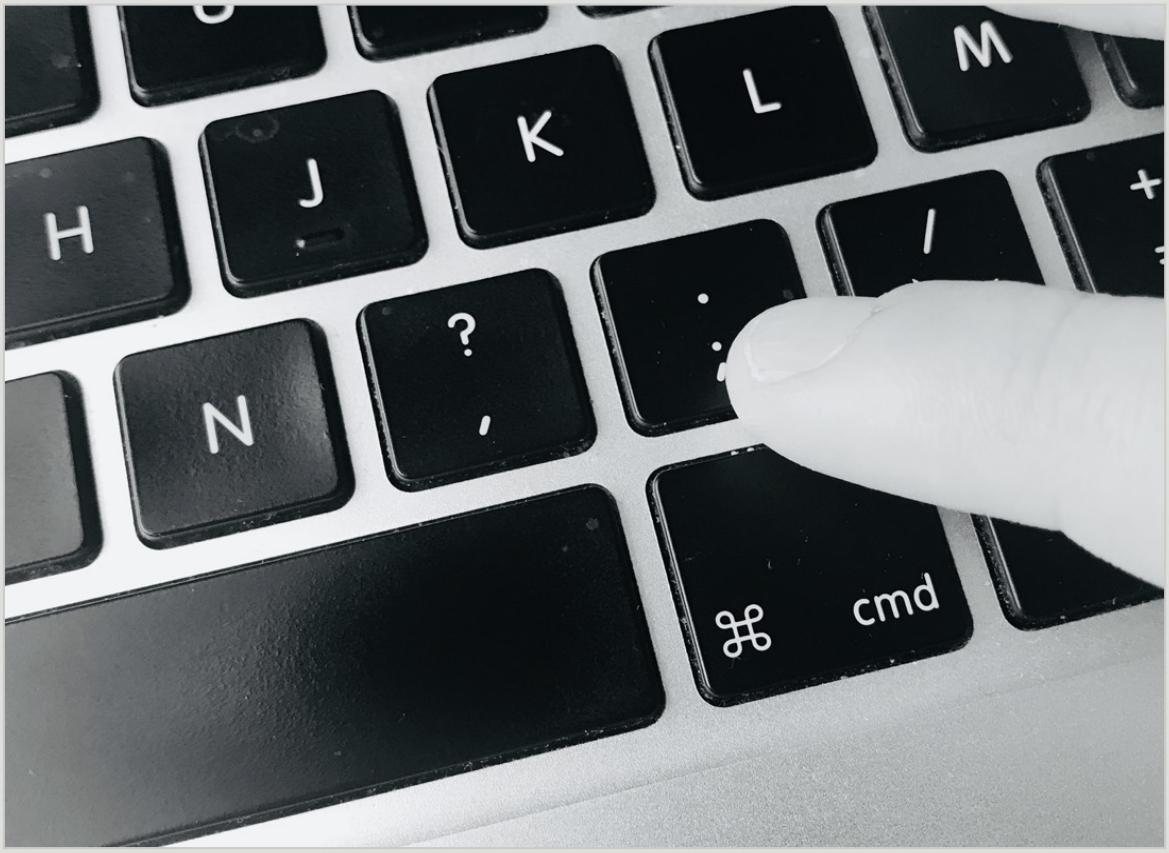

Le Bon Code

L’art délicat et sensoriel de la programmation, et sa conjugaison avec le monde de l’entreprise.

Avant-Propos Et Prérequis

La programmation est toujours présente dans nos vies. Même si vous n'êtes pas un programmeur né, vous utilisez certainement tous les jours des appareils divers: télévision, ordinateur, téléphones, machines à laver, cafetière, voitures, télécommandes, et plus encore.

Chacun de ces objets utilise en son sein de la programmation. La programmation est aux ordinateurs ce qu'est l'ADN aux humains. Le code qui décrit comment ça doit fonctionner.

Mais il existe plusieurs façons d'écrire un programme, et plusieurs langages de programmation pour le faire. Tout comme il existe énormément de façons de construire une voiture, avec une multitude d'outils.

Cet ouvrage n'a pas pour vocation de vous enseigner un langage de programmation.

L'ensemble de ce que vous lirez va pouvoir s'appliquer à tous les programmes que vous aurez peut être un jour à concevoir et développer.

Ce livre s'adresse donc aux curieux, aux programmeurs débutants ou un peu moins débutants, pour partir du bon pied! Notez qu'il est recommandé de connaitre les concepts de fonction, variable, et types de données, même si ce n'est pas obligatoire.

Et même si vous n'envisagez pas de programmer tout comme je n'envisage pas de devenir astrophysicien, ça ne vous empêche pas de lire ce livre tout comme ça ne m'empêche pas d'aller me documenter sur les trous noirs ;-)

CHAPITRE 1

Changelog

Si vous avez lu des versions précédentes du Bon Code, voici les changements apportés. Le changelog (de l'anglais qui veut dire «journal des changements») est un document, quasi exclusivement numérique, fourni avec les mises à jour d'un programme pour indiquer ce qui a changé afin que l'utilisateur sache où regarder.

La version originale du Bon Code était la 1.0. Voici la liste des changements apportés par chaque version.

Version 1.5 {Celle que vous êtes en train de lire}

-Ajout du Précepte «L'indentation»

-Ajout du Précepte «Le typage des variables»

-Ajout du Précepte «Le nommage des variables»

-Ajout du Précepte «L'initialisation des variables»

-Compléments sur le Concept «Passage par adresse, par valeur»

-Ajout du Concept «Isoler pour tester»

CHAPITRE 2

C’est Quoi Un Programme

Bon, avant d’entrer dans le vif du sujet, je vais définir un programme.

Si vous voulez un exemple de programme, prenez une notice de montage de meuble: vous tenez un programme. Un programme est une suite d’instructions. On appelle cela aussi un algorithme.

Dans la même veine, une recette de cuisine est l’exemple parfait d’un algorithme. Vous voulez un gâteau au chocolat? Suivez la recette.

Dans un programme informatique, c’est pareil. Vous avez une suite d’instructions qui ont été écrites, et que l’ordinateur va lire et exécuter, comme un soldat. Ces instructions, c’est le code. D’ailleurs, on dit aussi quand on veut écrire un programme que l’on va «coder», ou encore «développer».

Les instructions seules ne servent à rien. Dans votre recette de cuisine, qui vous dit «mélangez le chocolat et le beurre», vous ne pouvez rien faire si vous n’avez pas le chocolat et le beurre.

Et quand votre notice vous dit d’emboîter la pièce A dans la pièce B, vous n’emboîterez rien du tout si vous n’avez ni la pièce A ni la pièce B.

Dans un programme informatique, c’est pareil. Et les éléments dont vous avez besoin et que vous allez manipuler se nomment des variables.

Et sachez que la notice comme la recette de cuisine peuvent s’écrire en français, en anglais, en chinois...et c’est pareil pour un programme informatique. Il existe plusieurs langages de programmation, fournissant des instructions différentes qu’on peut utiliser.

Un exemple extrêmement simple maintenant pour illustrer tout ça:

```
char monTexte[] = "Coucou";

printf("%s", monTexte);
```

Votre chocolat, votre variable, c’est monTexte, que vous créez avec la première ligne.

Et vous demandez à l’ordinateur à la seconde ligne, d’afficher monTexte à l’écran en utilisant l’instruction printf.

Et comme «monTexte» est égal à «Coucou», votre écran va afficher «Coucou».

«printf» est une des instructions (on dit une fonction) du langage de programmation appelé C. Oui, juste C. Le C est un langage de programmation.

CHAPITRE 3

On Commence Par...

Les Préceptes. En fait, je vais structurer Le Bon Code en 3 parties:

-les Préceptes, qui sont les règles que j’ai accumulées au fil de mon expérience jusqu’ici, qui m’ont été apprises ou que j’ai comprises, et qu’il vaut mieux avoir en tête et suivre lors de vos développements

-les Concepts, qui vous permettront d’élargir votre façon de penser et concevoir vos programmes, et leurs interactions avec les utilisateurs ou avec d’autres programmes

-les Pensées, qui vous n’ont pas directement un lien dans la programmation mais qui vous seront fort utiles face à vos partenaires: les autres développeurs, les utilisateurs, les décideurs...

CHAPITRE 4

Les Préceptes

Que ces Préceptes vous hantent.

Que leur ombre plane au-dessus de vous quand vous maltraitez le code.

SECTION 1

DRY

Sans conteste l’un des Préceptes de base. D’ailleurs c’est pour ça que je commence par lui.

DRY est l’acronyme de Don’t Repeat Yourself. Et c’est l’une des premières règles qui m’a été enseignée.

En français: Ne vous répétez pas.

Le copier-coller ne devrait pas avoir sa place dans un code. Pour appliquer correctement ce Précepte, c’est simple: à chaque fois qu’il vous vient l’idée, l’envie, l’impulsion de faire un copier d’un morceau de code pour aller le coller plus loin, tremblez.

Posez-vous la question: pourrais-je faire autrement?

Car copier-coller un code possède un avantage aussi énorme que les inconvénients que cela procure.

Vous irez très vite: copier coller est très rapide. Et votre code sera bien vite opérationnel. Et sur le moment, ça vous semblera très simple à comprendre.

Mais quand vous devrez mettre à jour ce code (et vous le devrez, un jour ou l’autre, dans 1 mois, dans 1 an, dans 10 ans), vous devrez donc repasser partout ou vous l’avez dupliqué. Il suffira d’oublier un seul endroit, pour que la cohérence de votre programme se casse. C’est particulièrement insidieux car souvent, cet oubli ne se verra même pas, sauf quand l’utilisateur final va vous rapporter un comportement étrange, un calcul incorrect. Lors du débug, vous vous rendrez compte que «ah oui, j’ai aussi oublié de modifier là....»

Parlons en du débug. Le débug, c’est l’action qui pour un développeur, consiste à exécuter le programme ligne par ligne, afin de vérifier chaque valeur des variables, pour trouver où un bug se situe; plus largement cela désigne aussi l’ensemble de ce qu’il fait quand il est à la recherche d’un bug, prouvé ou supposé (euh, sans

inclure le café qu'il prend à la pause). Quand vous disposez d'une dizaine de lignes à un endroit de votre code, et que c'est le seul endroit où elles se trouvent, cela reste assez simple. Quand les lignes font partie d'un gros projet, c'est déjà plus compliqué. Mais si en plus vous les avez copié coller partout, alors là, le débug vous prendra bien plus de temps. Si vous n'avez qu'un endroit où chercher, vous trouverez plus vite, et vous corrigerez plus vite, une fois pour toutes.

Le copier coller est donc un fléau. Vous corrigerez un endroit, en oublierez un autre, et l'utilisateur se rendra compte qu'un endroit de son programme ne fonctionne pas comme un autre.

Vous devez donc rassembler les copier coller, on dit factoriser.

Faites cependant attention, ne factorisez pas du premier coup! Vous pouvez d'abord faire un copier coller, et factoriser plus tard (bon, dans la même journée j'entends, pas 4 ans après). Lorsque vous débutez, ne cherchez pas à faire un code parfait du premier coup. C'est plus difficile, et en plus vous ne parviendrez jamais à voir où sont les erreurs si vous ne les commettez pas. Faites le code tel que vous le sentez, et ensuite, posez-vous la question de son amélioration. C'est très souvent comme ça que je procède: je codes, pour que ça marche, et ensuite, je factorises, une fois, puis deux, puis trois, davantage si nécessaire et au final, le code est factorisé, et on ne trouve plus de répétitions.

SECTION 2

Les Variables Globales

Dans un programme, vous allez coder des fonctions, aussi appelées méthodes. Ce sont des morceaux de programme qu'on peut appeler à volonté. A l'intérieur, comme dans tout programme, vous allez disposer de variables, dites locales. Elles ne sont connues que du traitement dans lequel vous les avez créées, c'est à dire que l'ordinateur ne pourra les utiliser que dans la fonction où elles sont présentes.

Mais la visibilité d'une variable à travers un programme (on dit sa portée) n'est pas la même pour toutes. Il existe des variables qui peuvent être vues de toutes les fonctions, qui sont accessibles à n'importe quel endroit du programme: ce sont des variables globales. On les appelle aussi des globales, tout simplement.

C'est un fléau pour d'autre raisons que le copier coller, mais c'est un fléau quand même.

Imaginez qu'une variable globale soit tranquillement assise dans son coin, attendant qu'on vienne la modifier. Si une fonction s'en charge et modifie la valeur de cette variable, pas de problème.

Mais si une autre fonction s'exécute et utilise cette variable sans la modifier, alors elle ne sait pas dans quel état la variable va se trouver. Elle peut valoir tout et n'importe quoi, et c'est ce qui conduit à des comportement incorrects aussi appelé «effets de bord».

Plus vous ajoutez de globales à un projet, plus vous augmentez le risque d'effets de bord. Et ce n'est pas le pire.

Imaginez dans le scénario précédent que toutes les fonctions qui peuvent modifier la variable globale soient en fait une même fonction mais qui a été copiée collée et à laquelle on a apporté à chaque fois des modifications mineures. Si les modifications concernent la façon dont la globale a été lue ou modifiée, alors si vous souhaitez factoriser cette fonction, vous serez obligé de vérifier chaque utilisation de cette globale. A chaque fois qu'elle est lue, vous devrez

vérifier si la valeur attendue est la bonne, à chaque fois qu’elle est écrite, vous devrez vérifier si elle l’est correctement et pas avec une valeur qu’une autre fonction ne saura pas gérer. C’est difficilement factorisable, c’est d’ailleurs, sur les anciens projets sur lesquels j’ai travaillé, l’un des travails les plus longs et les plus minutieux que j’ai eu à faire. Quand vous faites des copier coller, des variables globales, vous accumulez un retard de conception: de la dette technique. Comme toute dette, vous devrez la payer. On verra plus tard comment et pourquoi.

SECTION 3

Le Gestionnaire De Sources

Quand vous codez, vous écrivez du code. Du code source. La gestion de ces sources est un point crucial d'un développement efficace.

Vous connaissez sûrement les logiciels de gestion de sources, ce qu'on appelle des logiciels de gestion de version, ou logiciels de versionning. Car leur principale fonction est de gérer la version de vos sources. Explications.

Quand vous écrivez une lettre sur un traitement de texte, vous sauvegardez la lettre sous la forme d'un fichier.

Vous pourriez prendre ce fichier et le donner à un gestionnaire de sources. Il va enregistrer le fichier, et lui attribuer la version 1.

Si vous modifiez la lettre le lendemain, et que vous la redonnez au gestionnaire de sources, il va voir que le fichier a été modifié par rapport au précédent (comment? ce n'est pas l'objet ici, mais il le sait, vous en faites pas!). Il va donc enregistrer à nouveau le fichier, mais va lui attribuer la version 2. Et ainsi de suite, il va enregistrer un historique de tous vos fichiers en leur attribuant à chaque fois un numéro de version qui augmente.

Si vous vous êtes trompé et qu'au final vous ne souhaitiez pas modifier ce fichier, vous pourrez demander au gestionnaire de sources de vous rendre la première version de ce fichier: il vous restaurera donc la version 1 de cette lettre.

Et bien pour la gestion de votre code source, c'est pareil, enfin presque: le logiciel de gestion de version va même pouvoir savoir quelles sont les lignes qui ont été modifiées, par qui, et quand. Pas seulement un fichier entier, mais chaque ligne de ce fichier.

Vous pourrez donc:

-revenir en arrière en cas d'erreur

-tester des modifications qui seront sans impact sur la version en cours de vos fichiers

-garder une copie de sauvegarde de vos codes sources ailleurs, car souvent, les logiciels de gestion de version sont installés sur des serveurs, qui hébergent donc une copie complète de vos codes sources.

Même si on est tenté de penser que ce n’est pas utile lorsque vous êtes tout seul car j’entends souvent que seul, «on en a pas besoin», je m’insurges, et je dis même le contraire: plus vous êtes seul à développer et plus il est indispensable de disposer d’un logiciel de versionning! C’est un atout majeur pour votre progression.

Les Commentaires

Quand vous écrivez du code, vous écrivez une suite d'instructions qui portent parfois des noms barbares. Au début de ce livre, on a vu une instruction du langage C: printf. Cette instruction permet d'afficher un texte à l'écran.

Il en existe des milliers d'autres dans plein de langages différents. Dans le langage PHP par exemple, printf existe et permet l'affichage d'une chaine de caractère, mais on utilise souvent la fonction echo à la place, qui remplit presque la même fonction.

Quand vous multipliez les fonctions et les variables au sein de votre code cela peut vite devenir compliqué de lire naturellement ce qu'il est censé faire, si vous avez un algorithme complexe à implémenter. Et vous pouvez avoir besoin d'apporter des précisions qui sont en lien avec le code mais qui ne sont pas du code. C'est là qu'interviennent les commentaires.

Un commentaire commence par une suite de caractères identifiables (généralement «//») et permet d'ajouter du texte pour expliquer ce que le code fait.

On distingue plusieurs sortes de commentaires que j'ai rencontré au cours de mes développements.

Premièrement, l'aplat de code commenté. Il s'agit de code source, mais qu'on utilise plus et qu'on a donc entièrement commenté. Je vais être très direct: ça ne doit jamais exister. Le code qui n'est plus utile doit être effacé et de toute façon, il pourra être récupéré par votre gestionnaire de versions qui en conserve une copie dans l'historique de ses fichiers (vous avez bien un gestionnaire de versions hein?). Si vous laissez des portions complètes de code en commentaire, vous vous retrouverez avec des fonctions longues, difficiles à débugger. Le code commenté n'est en général jamais plus décommenté car il est souvent vieux, et qu'on ne sait pas quel effet ça aurait si on le réactivait: il n'a donc rien à faire là.

Deuxièmement, voilà le genre de commentaire que j’ai pu croiser, je reprends le code en début de ce livre et je le commente:

```
char monTexte[] = "Coucou";

printf("%s", monTexte); //On affiche monTexte
```

Voilà typiquement l’exemple du commentaire inutile. La fonction printf sert à afficher une variable, ici la variable monTexte, il n’y a donc aucune utilité de le préciser en commentaire.

Troisièmement, le commentaire utile. C’est lui qui nous intéresse. Celui qui vous permettra d’expliquer la façon dont un algorithme doit globalement fonctionner, ou qui pourra apporter un éclairage particulier sur une portion du code qu’il n’est pas évident de comprendre à première vue. Ne commentez donc pas à outrance, commentez utile.

N’oubliez jamais qu’un code clair se suffit à lui même. Un code clair n’a pas besoin d’être commenté. Personnellement, je commente pour décrire des algorithmes, dans des commentaires un peu plus longs, qui décrivent un fonctionnement global, mais je commente rarement les fonctions elle même, en raison des deux Préceptes suivants vers lesquels nous allons immédiatement.

SECTION 5

Le Principe De Responsabilité Unique

Bon, si vous le voulez en anglais c’est le Single Responsibility Principle. Retenez bien: une fonction ne fait qu’une seule chose, et elle la fait bien.

Pour exemple, prenons le cas d’une facture, qu’on doit calculer puis imprimer.

On a une fonction qui va se charger du calcul, et une autre qui va se charger de l’impression. Si vous codez une seule fonction pour ces deux actions, vous obtiendrez les résultats suivants:

-une fonction à rallonge, dont le code sera moins facile à lire

-une fonction difficile à débugger

-une fonction plus sujette aux bugs au vu de son périmètre plus étendu

-une fonction plus difficile à factoriser si ça devait être nécessaire

-une fonction plus difficile à tester

-une fonction plus difficile à faire évoluer

-des effets de bords possibles à l’intérieur de la fonction (recoder la partie du calcul de la facture peut influer sur la partie liée à son impression)

Vous ne devez donc jamais regrouper tous les traitements au sein de la même fonction, mais les éclater en plusieurs traitements simples. C’est l’un des fondements de base de l’algorithmique: quand on vous demandera quelque chose à coder, vous serez face à un gros problème, que vous devrez découper en petits problèmes, plus simples à résoudre. Lorsque vous aurez résolu chacun des petits problèmes qui constituent le gros problème, vous aurez résolu le gros problème.

Pensez donc toujours à imaginer comment vous pouvez éclater le gros problème en briques unitaires, et réutilisables (la fonction qui calcule la facture pourrait servir toute seule, si on veut juste recalculer une facture mais sans forcément l'imprimer). Cela facilitera par la suite l'évolution de votre programme, permettra de le tester plus simplement, plus rapidement, et d'en déceler les bugs plus vite. Si votre programme est amené à évoluer vous pourrez en plus créer de nouvelles briques, sans casser les briques existantes.

1 Entrée, 1 Sortie

Lorsque vous ne respecterez pas ce Précepte, vous obtiendrez un code qui est un cauchemar à factoriser.

Une fonction possède un point d'entrée. C'est, pour faire très simple, l'endroit où elle commence. Elle possède également un point de sortie: c'est l'endroit où elle finit et ou elle renvoie une valeur au code qui l'a appelé.

On peut, sous certaines conditions, avoir des points d'entrée multiples. Une fonction peut être appelée de plusieurs façons (certaines peuvent recevoir 2 paramètres, ou plus, ou moins, mais la fonction porte toujours le même nom, il s'agit en fait de la même fonction). Cependant, je n'ai pour le moment quasi-jamais eu à me servir de cette possibilité. En tant que débutants et pour simplifier les explications, nous allons considérer que ce n'est pas possible pour vous et qu'une fonction ne peut s'appeler que selon une seule façon (on dit qu'elle n'a qu'un seul «prototype» ou une seule «syntaxe»)

Il en va d'une toute autre façon concernant les points de sortie. L'un des exemples les plus simples pour illustrer ce Précepte serait une fonction qui doit vérifier si votre disque dur est plein. Si c'est le cas, la fonction renvoie oui, sinon, elle renvoie non. Un disque plein, c'est pas bon.

Vous avez deux fois le verbe «renvoyer» dans la phrase ci-dessus. Dans le traitement que vous allez coder, vous aurez donc deux fois l'instruction permettant de sortir de la fonction en renvoyant une valeur.

Sur une fonction aussi simple, on pourrait penser que ce n'est pas grave. C'est presque vrai...

Supposez maintenant une fonction qui peut renvoyer plusieurs valeurs, parfois identiques, à plusieurs endroits. Vous aurez donc de multiples fois l'instruction de renvoi, qui stoppera la fonction.

C’est déjà difficile à débugger. Mais par dessus tout, c’est encore plus difficile à factoriser. Car quand on renvoie la valeur, la fonction s’arrête et le code qui suit la fonction de renvoi n’est pas exécuté. Cela change complètement la manière dont vous allez devoir recoder le traitement.

Pour faire mieux, il ne faut pas penser que la valeur que vous voulez renvoyer doit être renvoyée directement. Vous devez affecter cette valeur à une variable, et modifier cette variable au cours de la fonction si c’est nécessaire. A la fin de la fonction, vous avez une seule instruction de renvoi, qui va renvoyer la variable en question. Vous avez donc un seul point de sortie.

Reprenons l’exemple précédent, la fonction qui renvoie oui si le disque dur est plein. Au début de la fonction, vous déclarez une variable (de type booléen, qui ne peut renvoyer que 2 valeurs, oui ou non), par exemple bPlein. Si le disque est plein, vous affecterez oui à bPlein, sinon vous affecterez non. A la fin de la fonction, vous faites un renvoi de bPlein qui contient la réponse à la question : le disque est-il plein? Et vous n’avez qu’un seul endroit où cette valeur est renvoyée.

Mieux, vous pouvez déclarer bPlein et immédiatement affecter Oui à cette variable. Si le disque n’est pas plein, vous changez la valeur de bPlein en l’affectant à non, sinon vous ne faites rien et vous renvoyez bPlein. Cela permet d’une part de réduire la taille de votre code, mais le plus grand intérêt réside dans le fait que programmer de cette façon consiste à faire de la programmation défensive. Avant même que vous ayez vérifié le taux de remplissage du disque, votre variable bPlein vaut déjà oui puisque vous lui avez affecté cette valeur. Vous vous attendez donc déjà au pire, et si pour une raison inconnue vous ne pouvez pas vérifier le taux de remplissage de votre disque, alors votre fonction va considérer le pire scénario et dire que le disque est quand même plein à celui qui l’a appelé.

La programmation défensive consiste donc à faire en sorte que votre valeur de retour renvoie par défaut le pire cas qui puisse se produire, et c’est dans le traitement que vous modifiez cette valeur si c’est nécessaire. Cette valeur ne sera renvoyée qu’une seule fois, à la fin

du code, pour respecter le Précepte 1 point d’entrée, 1 point de sortie.

SECTION 7

Respectez Le Flow

Tout code qui s'exécute va suivre un chemin. En général, il va de haut en bas (sauf si l’écran est à l’envers...)

Vous vous devez de respecter le flow, sinon le débug d’une fonction sera horrible à assurer. Je ne vous parle même pas de sa factorisation. En ce sens, vous croiserez peut-être un jour une instruction «goto».

Le goto est une instruction un peu particulière dans les langages de programmation. Elle permet de demander au code de sauter une partie du code pour aller à un autre endroit, ou de revenir en arrière et d'exécuter à nouveau une partie du code. A tout instant, vous êtes donc capable de casser le flow du code et de lui dire d’aller ailleurs, et de cet ailleurs, vous pouvez même revenir au début, pour ensuite repartir au milieu. Un vrai sac de noeud qui vaut d’ailleurs à cette instruction d’être considérée comme l’un des éléments de la «programmation spaghetti», en ce sens qu’elle va permettre au code de sauter d’un bout à l’autre et de s’entremêler comme des spaghettis.

Je n’ai pas d’autre conseil à vous donner que de bannir cette instruction de votre code. Elle n’a rien à y faire.

Si vous devez sauter à un endroit plus loin pour exécuter du code, alors vous pouvez mettre le code en question dans une fonction et appeler la fonction. De même, si vous devez revenir en arrière pour exécuter à nouveau une portion de code, ça veut dire que cette portion devrait être dans une fonction que vous pourrez appeler. Si vous utilisez des goto, vous n’avez certes pas besoin de créer des fonctions. Mais vous vous retrouverez avec une fonction très longue (voir le Précepte du Principe de responsabilité unique), difficile à débugger et à tester, et très difficile à factoriser, surtout si elle possède des points de sortie multiples.

Les Tests Unitaires

Tout programme que vous allez coder devra être testé. Je ne discuterais pas sur le testing! C’est un métier à part, souvent assuré par une personne qui ne fait pas que ça a fortiori dans une petite entreprise, ou par une équipe entière, mais c’est un passage nécessaire et obligatoire. Mais ça, c’est pour votre programme lorsqu’il est «terminé» et qu’il répond au besoin initial.

Ce n’est pas de ça dont je vais parler dans ce Précepte. Il est ici question de tests unitaires.

Chaque fonction que vous allez coder devra être testée. La plus petite des fonctions que vous allez écrire devra être testée. C’est ça un test unitaire. Il porte bien son nom car il porte sur une unité: une fonction, un module que vous aurez conçu. Avant d’envoyer votre programme aux testeurs, vous devez vous assurez qu’il fait déjà ce qu’il est censé faire dans les conditions prévues. Si votre programme doit dire bonjour quand vous appuyez sur Entrée, alors, testez qu’il dit bien bonjour lorsque vous appuyez sur Entrée avant de l’envoyer en test.

Ça parait absurde dit comme ça, mais ils ne sont pas rares les cas où j’ai vu atterrir en production des programmes non testés. «Il est simple, pas de test ça ira plus vite». Peu importe la complexité du programme, sachez qu’il ne sera jamais plus rapide de le diffuser sans le tester. Non seulement les remontées de bugs devront être analysées et corrigées ce qui va déjà vous consommer pas mal de temps, mais en plus ce sont vos utilisateurs qui vous feront ces remontées, ce qui ne va pas manquer d’écorner votre image de marque. Le mot d’ordre est donc simple: testez toujours tout. Vous serez parfois surpris après avoir songé que «ce code est tellement simple qu’il ne peut pas échouer», de le tester et de voir que la programmation révèle bien des surprises!

Les tests unitaires incombent au développeur. Ne cherchez pas à vous y soustraire: votre métier n’est pas seulement de développer mais également de tester ce que vous développez. Pas aussi largement qu’un testeur pro dont c’est le métier, mais suffisamment pour ne pas faire perdre de temps aux équipes de test qui s’apercevront vite que les fonctions élémentaires de votre programme ne sont pas remplies. Un mécanicien ne se contente pas seulement de changer la courroie de distribution: il démarre le moteur pour vérifier que tout fonctionne correctement. Vous devez le vérifier aussi (le programme hein, pas votre moteur!)

Vous pouvez même coder des tests: cela vous permettra de lancer des tests automatiques. Je m’en sers pour développer des tests qui permettent de vérifier que les fonctions basiques du programme telles que les accès aux données, les connexions aux systèmes tiers, les moteurs de calculs les plus élémentaires, fonctionnent toujours malgré toutes les modifications apportées au reste du code source.

L’indentation

L'indentation, c'est le fait d'ajouter en début de ligne des espaces ou des tabulations. Dans votre code, l'indentation est primordiale et permet une meilleure lisibilité du programme. Il y a même des langages de programmation qui vous impose d'indenter correctement le code, sinon, il ne fonctionnera pas.

Et personnellement, l'indentation est pour moi un concept encore plus large: les sauts de ligne et les espaces dans le code lui même et pas seulement en début de ligne ont leur importance.

Voilà un exemple de code, qui réalise la chose suivante: déclarer 2 variables, leur affecter la valeur 5, faire une boucle «for» de 10 tours pour augmenter chaque variable de 10 unités, et afficher ces 2 variables à l'écran. Le langage utilisé est toujours le langage C:

```
int toto=5;int titi=5; for(int i=1;i<=10;i++){toto++;titi++;};
printf("%d",toto);printf("%d",titi);
```

Voilà le même code, mais indenté:

```
int toto = 5;

int titi = 5;


for(int i=1;i<=10;i++)

{

	toto++;

	titi++;

}


printf("%d",toto);
```

```
printf("%d",titi);
```

Avouez quand même que c'est plus simple à lire comme ça.

Les déclarations de variables se font les une à la suite des autres avec un retour à la ligne entre chacune des déclarations.

Lorsqu'une boucle démarre, elle est sur sa propre ligne, et le code qu'elle contient est indenté vers la droite, en général avec une tabulation.

Dans l'exemple ci-dessus vous voyez également qu'entre les déclarations de variable, la boucle for, et l'affichage, j'ai inséré un saut de ligne. Cela permet d'aérer le code, et d'en dégager les parties principales. De même, dans les déclarations de variable, le signe = est entouré de 2 espaces, pour rendre plus lisible la variable, et la valeur que je lui affecte.

Et de manière tout à fait générale, une ligne de code = une instruction et une seule.

Ce que fait votre code est bien plus simple à comprendre lorsqu'il est indenté! Pensez-y, pour vous, comme pour ceux qui le retoucheront ou le liront. Vous gagnerez du temps, et eux aussi.

SECTION 10

Le Typage Des Variables

Bon, là, c’est un gros morceau. Et il va y avoir 2 écoles: le typage fort et le typage faible. Vous entendrez également parler de typage statique, ou dynamique.

Sachez tout d'abord que les expressions "typage fort" et "typage faible" ont été souvent utilisées, et vidées de leur sens, car il est en fait assez difficile de définir à partir de quand un langage possède un typage fort ou faible; et il en va de même pour la notion de typage statique ou dynamique. Un langage peut être à typage fort dynamique ou faible statique, et fort statique, ou faible dynamique. On verra ça plus tard, mais en attendant, arrêtons-nous un instant sur la notion de typage pour comprendre la base.

Les variables que vous manipulez dans vos programmes sont stockées dans des cases de la mémoire. Or, toute variable est typée. Il existe différents types de données que peut prendre une variable.

Par exemple, en langage C, on peut écrire:

int toto = 5;

Cela signifie qu’on déclare une variable nommée toto, de type int. Int, c’est de l’anglais, c’est l’abréviation de integer. Un mot anglais qui veut dire entier.

toto est donc une variable qui ne pourra stocker que des nombres entiers.

Maintenant, voici un exemple de fonction (aussi appelée procédure), qu’on va faire en WLangage:

```
PROCÉDURE _multiplier(parametre1, parametre2)
RENVOYER parametre1*parametre2
```

Elle est très simple. Vous passez 2 paramètres à cette fonction, elle les multiplie (l'étoile indique une multiplication) et elle vous renvoie directement le résultat de la multiplication.

Je vais maintenant appeler cette fonction de deux manières différente:

```
unEntier est un entier = 4
uneLettre est une chaîne = "A"

_multiplier(unEntier,unEntier)
_multiplier(uneLettre,uneLettre)
```

Je déclare tout d'abord en WLangage (qui s'écrit essentiellement en français d'ailleurs, même si on peut l'écrire en anglais, ce qui est le cas de beaucoup de langages de programmation), un entier, et une chaine, c'est à dire une variable qui peut contenir des lettres et/ou des chiffres. L'entier vaut 4, la chaine vaut A.

Dans le premier cas, je demande à la fonction _multiplier de multiplier l'entier 4 par lui même. Le résultat obtenu est bien 16.

Dans le deuxième cas, je demande à la fonction de multiplier la lettre A par elle même, et le résultat est…que vous n'aurez pas de résultat car le programme a planté! Multiplier deux lettres n'est pas possible pour un ordinateur (alors, pour un matheux, ça se discute certes…).

Tenez, je vous donne le message d’erreur qu’il a renvoyé:
« L'opération '*' est interdite entre un élément de type 'chaîne ANSI' et un élément de type 'chaîne ANSI’. »

Et dans ce cas, vous n’avez su qu’il y avait un problème que quand vous avez exécuté le programme. Lorsque vous développez, votre ordinateur pourrait pourtant vous avertir de votre erreur! Pourquoi ne l’a t il pas fait, voyant que vous alliez essayer de multiplier des lettres?

Parce que vous ne lui avez pas dit. Les deux variables que la fonction attend, parametre1, parametre2, ne possèdent pas de type. On dit que ces variables ne sont pas typées. Pour vous avertir de votre erreur avant même que vous ayez lancé le programme, l’ordinateur doit connaitre le type des variables que vous manipulez.

C’est l’une des erreurs que je vois assez fréquemment, et qui aboutit comme dans cet exemple, à des plantages directement chez le client, car l’erreur n’est pas détectée avant, faute de typage correct.

Changeons donc la tête de notre fonction en indiquant simplement à l’ordinateur, le type des paramètres de la fonction:

PROCÉDURE _multiplier(parametre1 est un entier, parametre2 est un entier)

RENVOYER parametre1*parametre2

A peine ai-je écris ça que l’ordinateur m’indique que cette ligne est en erreur:

_multiplier(uneLettre,uneLettre)

Et l’erreur est la suivante: « Le paramètre 1 de type chaine ANSI ne peut pas être converti en type entier »

Le programme n’est donc même pas encore lancé que déjà, vous savez que le code est incorrect.

Précisez donc toujours le type de vos variables. Systématiquement. Vous devez savoir ce que vous manipulez, c’est l’un des premiers remparts contre les erreurs d’exécution, c’est à dire celles que vous ne pourrez détecter qu’une fois que le programme a été livré aux testeurs, qui se casseront les dents dessus, ou pire, aux clients.

Revenons un instant sur ce que je disais plus haut concernant le typage fort/faible, statique/dynamique. Il est possible, en WLangage par exemple, d’écrire ceci:

soit toto = 1

Vous déclarez donc une variable nommée toto, mais vous n’avez pas précisé son type. Mais comme vous lui avez affecté la valeur 1, l’ordinateur sait que son type va être entier. C’est ce qu’on appelle l’inférence de type. C’est pratique, mais souvent dangereux, car vous ne savez pas exactement ce que fait l’ordinateur justement. Un entier, ça peut être égal à 1, mais une chaine aussi. La lettre 1, le chiffre 1, ce n’est pourtant pas pareil pour un ordinateur. Il y a donc une ambiguïté. Lorsque vous affectez la valeur 1 a une variable censée contenir des lettres, il est possible que le langage convertisse automatiquement cette lettre 1 vers un entier égal à 1 en cas d’opération de multiplication par exemple. Lorsque cela se produit, on parle de typage dynamique. L’opération de conversion effectuée s’appelle un cast. Caster une variable, c’est la changer de type. Ce que parfois, l’ordinateur fait tout seul.

Mais encore une fois, cela laisse une ambiguïté, qui serait facilement levée si vous précisiez le type que vous manipulez, tout en permettant de détecter l'erreur au plus tôt, pendant votre programmation.

L'une des rares exceptions à ce Précepte, je la rencontre souvent dans la programmation objet. Il y a des cas où on ne doit surtout pas préciser de type, c'est même l'atout de la programmation objet car cela permet le polymorphisme. Mais cela représente assez peu de code, même dans mes projets. Le polymorphisme permet des factorisations de code tellement grandes, qu'on en écrit finalement assez peu par rapport au reste du code qui doit être typé. Et si vous prenez l'habitude de typer, il sera encore plus simple pour vous de détecter à quel moment vous ne devez pas typer.

Ne vous attachez pas tant aux terminologies de typage fort/faible, statique/dynamique, et retenez simplement que plus vous typez explicitement vos variables, dans le code, mieux c'est.

Le Nommage Des Variables

J'ai pu croiser dans les différents programmes que j'ai eu à maintenir, des noms de variable...curieux. DarkVador par exemple (si si, je vous jure). Bon, c'était rigolo, et heureusement pas bien grave.

Voici un exemple de code, mais je ne vais pas vous dire ce qu'il fait:

```
int i = 50;

int k = 400;

int d = 0;

int e = 100;


d = (k/e) * i;


printf("%d",d);
```

C'est chiant à lire hein?

Voilà la deuxième version:

```
int pourcentageReduction = 50;

int prixDepart = 400;

int prixArrivee = 0;

int diviseur = 100;


prixArrivee = (prixDepart/diviseur) * pourcentageReduction;
```

```
printf("%d",prixArrivee);
```

Et là, on comprend facilement que ce code servait à calculer un prix d’arrivée en fonction d'un prix de départ et d'un pourcentage de réduction, puis à l'afficher à l'écran.

Il va arriver un moment ou vous devrez reprendre les codes sources d'un autre. Si les variables ne sont pas correctement nommées, cela va vous induire en erreur, et pire, vous fera parfois modifier des variables d'une manière incorrecte, car vous ne savez pas exactement quelles données elles représentaient dans le programme. C'est dangereux! Nommez donc toujours vos variables correctement. Il existe des conventions permettant d'assurer un nommage cohérent de vos variables et de vos fonctions au sein d'un programme. Personnellement, j'utilise le lower camel case (et je vous laisserai creuser le sujet pour vous renseigner à ce propos!).

Concernant la longueur du nom, il va falloir être concis. Supprimez les articles « de », « du », « la » etc…soyez court mais explicites. On ne doit pas douter de ce que votre variable représente, sinon, on doutera du résultat qu’elle peut délivrer; et donc, on perdra du temps à le vérifier.

L'initialisation Des Variables

En langage C, lorsque vous déclarez une variable de cette façon:

int toto = 0;

Vous déclarez un entier, appelé toto, qui contient la valeur 0. Mais bon, quitte à écrire zéro, vous vous dites peut être «autant ne rien mettre du tout».

Si vous écrivez simplement:

int toto;

Pouvez-vous me dire quelle est la valeur de toto? Zéro?

Vous allez voir pourquoi c'est dangereux.

Si je demande à l'ordinateur d'afficher toto, j'obtiens 6356712. C'est un gros entier.

Lorsque vous déclarez une variable, l'ordinateur réserve une case dans la mémoire. Cette case va stocker la valeur de votre variable. Sauf que quand l'ordinateur arrive sur la case, une valeur est déjà présente. Elle a été laissée par un autre programme qui a utilisé la case avant vous.

Vous ne devez donc jamais supposer que la case contiendra zéro, mais systématiquement initialiser vos variables avec une valeur que vous avez choisie. Cela est valable pour n'importe qu'elle variable, pas seulement des entiers. Lorsque vous déclarez la variable, vous devez savoir quelle valeur elle contient, et le meilleur moyen pour ça est de lui en affecter une, on dit qu’on initialise la variable.

Il existe des langages qui fonctionnent différemment. Par exemple en WLangage, je n'ai pas besoin de faire ça. Le WLangage initialise lui même mes variables, et je peux choisir de ne pas m'en occuper.

Ce que je ne fais pas toujours, car un entier est par défaut initialisé à zéro, et parfois, je souhaites une autre valeur. Mais vous savez maintenant ce qui risque d'arriver si vous ne respectez pas ce Précepte: lorsque la valeur de la variable est imprévisible, le résultat du code qui utilisera la variable le sera aussi!

CHAPITRE 5

Les Concepts

Les Préceptes sont des bonnes pratiques, mais les Concepts sont des façons de réfléchir, des éléments auxquels vous devrez faire attention lorsque vous entamerez les phases de conception de vos programmes ou lorsque vous allez les développer.

Passage Par Adresse, Par Valeur

Vous ne maitrisez peut être pas encore cette notion, pas évidente quand on débute. Faites attention quand vous la croiserez...je vais tenter de vulgariser pour vous mettre le pied à l'étrier et je vous invite à potasser le sujet des pointeurs en programmation!

Quand vous déclarez une variable, elle est stockée dans la mémoire (=dans les barrettes de RAM). Si vous déclarez une variable toto et que vous affectez le chiffre 5 à cette variable, il y a une case de la mémoire qui va contenir la valeur 5.

Si vous déclarez une variable titi et que vous faites titi = toto, alors titi vaudra 5 aussi. Mais le 5 de titi sera stocké dans une autre case de la mémoire. Vous aurez donc à cet instant 2 cases dans la mémoire qui valent 5. Si vous changez la valeur de toto et que vous mettez 6, titi vaut toujours 5. Chacune de ces cases est identifiée par son numéro, son adresse dans le jargon. Une case est donc caractérisée par son adresse (le numéro de la case dans la mémoire) et sa valeur (la valeur que contient la case).

Mais il est aussi possible de dire à une variable de ne pas posséder sa propre case, mais d'utiliser la case d'une autre variable. Ainsi, si vous modifiez la valeur de la première variable, la deuxième sera affectée aussi puisqu'elle désigne en fait la même case dans la mémoire de l'ordinateur.

Cela peut devenir problématique si dans votre code, vous ne faites pas attention à la manière dont les variables sont utilisées. Vous risquez involontairement de modifier des valeurs qui sont en fait reliées à plusieurs variables dans le code, et de provoquer des effets de bords qui conduiront à des erreurs, alors que vous auriez souhaité que des variables soient réellement indépendantes.

Dans le même genre, il est possible de coder une fonction qui par exemple va additionner deux nombres et retourner le résultat de

l’addition. Pour appeler cette fonction, vous allez donc lui passer 2 nombres à additionner.

Si le passage de ces 2 nombres, de ces 2 paramètres comme on dit, se fait par adresse, alors ça veut dire que vous n’envoyez pas une copie de ces 2 nombres à la fonction (auquel cas, ce serait un passage par valeur). Vous lui indiquez directement dans quelle case de la mémoire on peut les trouver.

Tant que la fonction n’y touche pas, et qu’elle se contente de lire ces 2 variables, pour les additionner, et qu’elle retourne le résultat, ce n’est pas gênant. Par contre, si dans la fonction, des lignes de code modifient la valeur de ces variables dans la mémoire, alors là, prudence. Le code qui a appelé la fonction ne s’attend sûrement pas à ce que la fonction modifie ce qu’on lui a passé : une fonction n’est pas censée agir sur ses paramètres. Chaque fonction doit renvoyer une valeur, qu’on pourra utiliser en l’affectant à une variable qu’on aura déclarée.

J’ai plusieurs fois constaté que des fonctions étaient appelées dans le code, et qu’on passait à ces fonctions des paramètres qu’elles allaient modifier. Ce n’est pas propre, car coder ainsi viole le précepte du principe de responsabilité unique. En l’occurrence, ma fonction d’exemple doit additionner deux nombres et retourner la somme, on ne lui demande pas de changer les paramètres d’entrée. Une fonction doit être isolée, ce doit être une boite noire qui fait le boulot, sans interférer sur son environnement extérieur. Sur l’exemple de l’addition, c’est simpliste, mais appliquée à des fonctions plus complexes, c’est une source d’erreur particulièrement insidieuse.

Dans les différents langages de programmation, il existe des moyens permettant de passer des variables par adresse ou par valeur à une fonction, et il y a un fonctionnement par défaut. En WLangage par exemple, les paramètres sont toujours passés par adresse, sauf si je spécifie moi même le mot clé LOCAL dans chaque paramètre de la fonction, ce qui la forcera à toujours récupérer ces paramètres par valeur.

Modularité

C’est une notion difficile à mettre en oeuvre lorsqu’on commence l’apprentissage de la programmation, et c’est pourtant l’une des clés qui vous permettra de mettre au point des programmes faciles à faire évoluer, à tester, et à débugger.

Vous avez pu lire précédemment que lorsque vous faites face à un gros problème, vous devez le diviser en plus petits problèmes, et quand vous résolvez tous les petits problèmes, vous résolvez le gros problème.

Quand on débute, on a du mal à trouver le bon niveau de division: parfois on divise trop, parfois pas assez. Vous apprendrez par votre expérience comment diviser au mieux. Cela ne dépend pas d’une règle établie mais du projet que vous menez.

Néanmoins, le Concept de modularité vous impose de ne jamais faire de programmes «énormes». Chaque fonction qu’un programme doit remplir ou dont il aura besoin pour fonctionner doit être isolée.

Prenons un exemple, ce sera plus parlant. On vous demande de créer un programme qui va se connecter à un serveur pour récupérer des données, afin de faire des calculs dessus, pour ensuite les imprimer. Au final, vous ne devez obtenir qu’un seul programme, mais à l'intérieur, vous pouvez déjà vous douter que vous aurez 3 briques indépendantes à développer: le module qui va récupérer les données, le module qui va faire les calculs, le module qui va les imprimer. Vous ne devez pas faire une fonction qui fait tout, mais plusieurs fonctions qui sont isolées les unes des autres et remplissent chacune leur rôle. C’est toujours ainsi que vous devez penser vos programmes. Un programme est constitué de briques.

Votre programme tout entier pourra être lui même considéré comme une brique: imaginez qu’on vous demande de livrer votre programme pour qu’il récupère et calcule les données, mais au lieu de les imprimer, il doit les envoyer à un autre programme, fait par un

autre développeur, qui va lui utiliser vos résultats pour faire autre chose. Vous ajoutez donc à votre programme une brique pour qu'il puisse envoyer les données. Mais cet autre développeur verra votre programme comme une brique unique, dont le seul rôle est de lui envoyer des données. Il ne saura pas comment vous avez fait, il ne saura pas que vous devez les récupérer, les calculer, il ne sait pas quelles briques vous utilisez pour le faire, il sait juste que vous devez lui envoyer des données. Et il verra votre programme comme l'une des briques d'un système plus évolué de traitement de données.

En conception, vous devez donc raisonner ainsi. A l'échelle de votre programme pour commencer, mais rapidement, vous aurez à interagir avec de multiples éléments qui seront autant de briques à prendre en compte. Au final vous serez donc capable de concevoir un ensemble de briques qui feront le boulot, et qui seront plus simples à faire évoluer, à tester et à corriger qu'un énorme parpaing qui fait tout!

SECTION 3

Procédural Et Objet

La programmation procédurale et la programmation objet sont deux façons de faire, deux paradigmes. Les deux permettent d'obtenir des programmes.

Le combat est toujours vif entre les défenseurs de l'un et de l'autre. Le style de programmation procédural est vite qualifié de «pourri» quand celui de la programmation objet est paré d'un modernisme fou et d'un style incontesté.

Navré de décevoir: les deux styles se valent. Il existe des programmes pourris en procédural et tout aussi pourris en objet, de même qu'il existe des programmes propres en procédural et en objet. La différence ne réside pas là.

Le procédural, c'est bien pour certaines choses et moins bien pour d'autres. La conception d'un programme objet nécessite davantage de réflexion, ce qui se paye au prix d'un temps plus long mais qui permet ensuite de développer plus vite.

Le procédural peut être utile pour des projets qui ne seront pas largement diffusés mais se limiteront à quelques dizaines d'utilisateurs, ou pour faire des maquettes de démonstration. Néanmoins, il suffit d'un peu plus de réflexion pour faire des programmes bien plus gros même en procédural. Etant un développeur objet, j'ai naturellement tendance à développer ainsi car l'objet m'amène des avantages absents du procédural. Ces avantages me viennent très vite en tête lorsque je fais du procédural (tiens, en objet, j'aurais fait comme ça..), aussi, il vaudrait mieux que vous cherchiez par vos propres moyens des infos sur ces deux paradigmes afin de vous faire une opinion. Mais ne vous enfermez pas dans un style!

SECTION 4

Dette Technique

Dans le Précepte sur les variables globales, je parlais de dette technique. Chaque fois que vous déviez d'un Précepte, que vous appliquez mal un Concept, vous risquez d'accumuler de la dette technique. C'est un risque que vous pourrez ou parfois devrez prendre, car les impératifs de coûts et de délais ne vous laisseront peut être pas toujours le choix de faire autrement. Néanmoins, vous risquez fort de payer un jour ces erreurs. Tout comme je l'ai payé dans l'exemple que je vais vous raconter.

Il y a longtemps, j'ai du factoriser un morceau de code pour le rendre plus beau, plus propre, plus simple à comprendre. Mais pas plus rapide.

En effet, une partie du code nécessitait un temps plus long pour son analyse et son optimisation. Temps que je n'avais pas, j'ai donc écrit un code certes propre, qui fait ce qu'il doit faire, mais qui ne le fait pas aussi bien qu'il devrait. Et en haut de ce code, j'ai mis un commentaire qui disait «pas le temps de faire mieux, on verra plus tard».

Un jour un client nous a appelé car la partie du programme qui lançait ce code était très lente. La fenêtre mettait des dizaines voire des centaines de secondes à s'afficher.

J'ai donc été forcé pour améliorer ça, de prendre le temps que je ne m'étais pas accordé au départ. J'ai centuplé les performances de la fenêtre qui ensuite s'affichait quasi instantanément. La solution ne prenait pas davantage de lignes de code mais était réfléchie d'une autre façon, plus optimisée. Et j'ai supprimé mon commentaire...

Soyez donc attentif à votre programme. Toute dette technique se paiera, un jour ou l'autre.

SECTION 5

Isoler Pour Tester

Il va vous arriver (plus ou moins régulièrement...) de faire face à des comportements bizarres du code. Vous vous poserez sans cesse la même question : pourquoi?

Afin de pouvoir tester plus vite et plus efficacement un morceau de code, pensez à l'isoler. Quand je dis isoler, je dis isoler dans un projet totalement à part, un programme extérieur dont vous démarrez le codage de zéro.

Une fois le code incriminé isolé, testez-le: parfois, ça marchera très bien dans ce petit programme tout neuf, alors que ça ne fonctionne pas dans votre programme (moins neuf...). C'est le signe que le problème ne vient pas forcément du code en question mais d'un autre endroit du code, qui peut par exemple toucher à des variables dont le morceau de code en erreur à besoin.

Isoler est fondamental. Le projet dans lequel est implémenté le code en erreur est potentiellement une grosse machine. Possiblement remplie de variables (globales ou pas), de fonctions appelés à divers endroits, n'importe quand. Vous ne pouvez pas débugger dans un environnement aussi pollué. Pensez donc à isoler pour voir si vous arrivez à reproduire le problème. Si oui, cela va énormément faciliter le débug, et vous n'aurez qu'à réimplémenter la solution que vous aurez trouvé dans le code initial.

CHAPITRE 6

Les Pensées

Je viens de passer quelques pages à expliquer des notions qui sont purement du domaine du développement, de la conception, de la technique. Mais ce chapitre apporte une nuance supplémentaire.

Il y a plusieurs environnements dans lesquels vous serez amené à développer, mais un programme répond en général à un besoin. Un client, votre patron, une entreprise peut vous demander de coder un programme. Cela va devoir vous amener très vite à réfléchir en terme de conception, de délais, et donc d'argent.

En tant que développeur, si vous êtes salarié, votre travail consistera entre autres à présenter les risques: à votre supérieur de décider si il veut les prendre. Vous aurez en tête de nombreuses méthodologies pour appliquer l'art du métier, art qui n'est malheureusement pas toujours compatible avec le monde de l'entreprise.

Disons le d'emblée: la guerre qui existe entre les décideurs qui veulent des programmes rapides à faire, bien fait, riches de fonctionnalités, pas cher, et les développeurs qui savent qu'un programme bien fait, riche ne peut pas être rapide à faire et pas cher ne sera pas l'objet de ce chapitre.

Néanmoins, inutile d'entamer une guerre de tranchée en campant sur vos positions. Car il existe des façons de répondre à des besoins de programmation sans sacrifier l'art du métier à outrance, tout en respectant les contraintes de temps, d'argent et de technique qui vous seront imposées. Les exemples ne manquent évidemment pas, de la TPE jusqu'à la multinationale, pour vous montrer qu'il est bien possible de sortir en production des programmes viables et évolutifs.

SECTION 1

Les Langages Et Leur Prix

Beaucoup de développeurs ont un langage de prédilection. Me concernant, je suis essentiellement amené à developper en WLangage: c'est mon langage principal.

Cela étant, même si j'ai un niveau en WLangage supérieur à tous les autres langages que j'ai pu pratiquer, je ne me cantonne pas seulement à ça. Un développeur sait développer, il n'est précisé nulle part qu'il ne sait développer que dans un seul langage. J'ai ainsi fait du C, C++, C#, PHP, JavaScript, Java, VB.Net...

Aucun langage de programmation n'est meilleur qu'un autre sinon tout le monde l'utiliserait. Simplement certains langages sont plus adaptés que d'autre à la création de certains types de programme. Le langage n'est qu'une boite à outils, mais les concepts de la programmation s'y appliqueront quand même.

Un mécanicien sait se servir d'un tournevis, peu importe sa marque; mais il sait qu'il existe des tournevis plats, cruciformes, petits, gros, et chaque tournevis est fait pour un usage spécifique.

Néanmoins, sachez que tous les langages ont un prix. Un développeur COBOL (COBOL est un langage de programmation utilisé dans le domaine financier) peut se vendre plus cher qu'un développeur PHP car il est plus rare. Cela compte quand vous concevez un projet, car le choix du langage peut grandement influencer les coûts.

Vous aurez néanmoins une longueur d'avance si vous ne vous réduisez pas à n'avoir vu et utilisé qu'un seul langage. Pour exemple, le WLangage peut ne pas savoir faire une tâche donnée, mais qu'importe: vous pouvez lui incorporer du C# et le faire à sa place. Maitriser le C# vous permet donc de vous adapter. J'ai été amené à patcher (corriger) des morceaux de code Java qui certes ne sont pas du WLangage, mais qui fonctionnait avec lui. Maitriser un peu le Java m'a donc permis de m'en sortir seul, ce qui sera une

compétence appréciable dans une entreprise. Car un développeur Java coûte aussi de l’argent.

SECTION 2

Identifiez Les Profils

L’une des erreurs les plus fréquentes, que j’ai moi même commise, est de considérer que tout le monde comprend de quoi vous parlez quand vous l’expliquez avec vos mots. Or, ceux qui ne sont pas de votre domaine ne peuvent pas le comprendre avec vos mots, mais ils peuvent s’en approcher avec les leurs.

Il est donc inutile de partir dans des considérations techniques poussées, assorties d’un vocabulaire soutenu, pour vous adresser à votre collègue commercial. Si en retour il vous explique son métier de la même façon, vous ne comprendrez probablement pas grand chose non plus. Vous devez donc expliquer votre métier, à un niveau qui lui est accessible, et si il fait pareil de son côté, alors vous trouverez un terrain de communication commun. Qui vous permettra de comprendre les besoins de l’autre et d’y répondre par une solution appropriée, chacun dans vos domaines respectifs.

Cela vaut pour un commercial, mais aussi pour votre supérieur qui n’est peut être pas à votre niveau en développement, ou qui ne développe pas dans le même langage que vous, ou qui ne développe même pas, et auquel vous aurez à expliquer des spécificités du langage que vous utilisez qu’il ne connait pas.

Vous devez donc trouver des moyens de vulgariser, en pensant que si vous l’expliquez d’une manière que vous comprenez, ça veut dire qu’en face on ne vous comprendra pas. Changez de méthode, et expliquez par des moyens que vous n’utilisez pas pour vous l’expliquer à vous même ou à vos pairs du même domaine.

Pour exemple: reprenez le sujet des passages par adresse et par variable, abordé dans les Concepts. Une mauvaise utilisation de ce Concept ou sa méconnaissance peut aboutir à des erreurs sur votre programme. Imaginez qu’une erreur se soit produite: dans une application, vous vous rendez compte que 2 variables différentes (toto et titi) possèdent la même valeur mais que c’est anormal, elles

devraient être indépendantes. L'utilisateur va vous faire remonter cette erreur mais comme elle l'handicape dans l'utilisation du programme, il va demander à être partiellement remboursé.

On va donc vous demander pourquoi ça ne fonctionne pas comme ça devrait et pourquoi l'utilisateur demande un remboursement. Ce peut être un de vos collègue développeur qui vous pose la question mais souvent, c'est de la part d'une personne tierce que la demande émanera: le client va contacter un autre service, demander un remboursement, et c'est une personne de ce service qui vous posera peut être la question. Il y a deux manières de lui expliquer, premièrement:

-j'ai une variable qui a été modifiée par une ligne de code, mais ailleurs un pointeur était déjà déclaré sur cette variable, donc quand le pointeur a touché à la variable ce sont tous les codes utilisant la variable qui ont été impactés, et ça n'aurait pas du arriver. Je l'ai changé.

Là, au mieux on acquiesce poliment et on ira dire au client qu'on a pas trop compris mais que c'est corrigé, au pire, on vous envoie le pot de fleurs à travers le bureau (bon, j'exagères peut être un peu).

Changez de méthode: la personne en face ne connait pas votre métier. Dites plutôt:

-une valeur a été modifiée en même temps par 2 éléments au sein du programme, c'est un bug que j'ai corrigé.

Certes je vous l'accorde, cette deuxième explication peut ne pas être strictement égale à la réalité. Mais pour qu'elle le soit, vous devriez fournir des détails auxquels votre interlocuteur ne sera pas sensible. On accepte donc de diminuer la «véracité technique» de l'explication pour qu'elle soit acceptée par ceux auxquels elle est destinée. Et si c'est accepté, ça passe tout de suite mieux.

La conduite du changement en entreprise se repose sur la même façon de communiquer: au lieu d'expliquer aux standardistes ce que vous allez techniquement changer dans l'infrastructure réseau de l'entreprise, soyez factuels et expliquez avec des éléments qui leur

sont familiers. Si vous augmentez le nombre de serveurs et d'IPBX, et que vous les passez sur fibre optique dans un datacenter, au lieu de les laisser sur la connexion ADSL anémique qui rame comme un kayakiste malade, dites plutôt qu'à terme ils recevront les appels plus vite, avec une meilleure qualité, et sans coupures ni distorsions. La première explication leur fera peur, le changement leur paraitra compliqué, important, et on a toujours peur de ce qu'on ne connait pas. La deuxième leur fera comprendre que les modifications vont dans le sens de leur quotidien, qui s'en trouvera amélioré, et ce sera tout de suite mieux accepté. Pour gérer le changement en entreprise, soyez proches du problème. Les utilisateurs ont des problèmes avec la technologie mais ils ne sont pas sensibles à la technologie. Expliquez leur donc concrètement ce que le changement leur apportera dans leur quotidien.

Et si ça ne change rien, dites le aussi! Le fait de dire «on va changer des choses dans l'infrastructure réseau» sans préciser que certains ne seront même pas concernés par le changement, peut inutilement générer une appréhension de leur part alors qu'ils n'avaient en fait même pas à s'inquiéter, puisque rien ne changera pour eux.

Pensez Coûts

Essentiellement, vous ferez des programmes pour gagner de l’argent. Vous pouvez en faire pour votre plaisir, mais si vous êtes un développeur pro, c’est pour gagner de l’argent que vous coderez.

Que ce soit pour vous ou pour une entreprise, sachez bien que le temps, c’est de l’argent. L’entreprise est là pour gagner de l’argent, et même si vous devenez votre propre patron, vous serez là pour en gagner aussi.

Lorsque vous devez argumenter, mettez donc le prix dans la balance. Plus votre expérience augmentera, plus vous saurez quand et pourquoi le temps passé pourra augmenter, et le coût avec lui.

Les questions sont toujours les mêmes:

-qu’est ce que je veux?

-en combien de temps je le veux?

-combien d’argent puis-je y mettre?

Et la maxime associée ne varie pas: si t’as pas le temps, il te faut l’argent, si t’as pas l’argent, il te faut le temps. Chacun est conscient que les programmes riches, sans bug ou presque, bien codés, évolutifs facilement et rapidement, ne sont pas les moins chers à réaliser. Le tout est de trouver les compromis nécessaires. Pour faire entendre votre voix, sachez apporter de manière juste et adéquate l’argument financier, imparable quand vous vous adressez aux décideurs. N’oubliez pas qu’un langage que vous maitrisez moins coûte plus cher: soit vous serez plus long à coder, soit il faudra vous former.

De plus, les phases de conception où vous écrirez peu de code mais où ce code devra être très bien écrit, ont un certain coût également.

SECTION 4

Soyez Open

Très souvent vous vous retrouverez confronté à des interlocuteurs qui maitrisent d'autres technologies que les vôtres. L'informatique est particulièrement victime des dogmes. Un développeur expert dans un langage pourra vous narrer toutes les raisons qui font de ce langage LE langage. Et se fera une joie de vous dire à quel point tous les idiots de la Terre feraient mieux de se servir de ce langage au lieu de perdre leur temps sur le leur.

Ne tombez jamais dans ce piège. Il existe des milliers de langage, base de données, logiciels différents. J'ai conçu ou participé à la conception et à la programmation de systèmes basés sur mySQL, mariaDB, SQL Server, Hyperfile, avec du WLangage, du PHP, du C#, du Java, parfois tout en même temps, et ça fonctionnait très bien. Bien sûr qu'on aurait pu tout faire dans un seul de ces langages, évidemment. Bien sûr qu'on aurait pu prendre d'autres bases de données. Mais eu égard aux délais, aux compétences disponibles, aux moyens accordés, des solutions ont été trouvées pour qu'elles répondent aux besoins et aux impératifs de coût et de délai. Elles sont peut être imparfaites sur certains points, meilleures sur d'autres, mais qu'importe. L'essentiel, c'est de ne jamais fermer votre esprit à des solutions quand vous aborderez un problème. Le monde de l'informatique va vous donner tous les outils pour concevoir une multitude de solutions, sachez simplement les chercher et les combiner si besoin.

Pour exemple, j'ai eu un programme à coder qui devait récupérer des informations depuis un site internet. Le WLangage ne pouvait pas effectuer une opération dont j'avais besoin pour communiquer efficacement avec le site (bon pour les puristes, je n'arrivais pas à faire fonctionner les MemoryStream en WLangage), et je n'avais pas le temps de coder l'ensemble du programme en C#, même si il aurait pu répondre à tout le besoin client, et même si j'aurais fini par y arriver. Qu'importe, vu que je n'ai pas le temps de le faire, je vais coder un bout de programme C# qui va me permettre de régler ce

problème et revenir au WLangage pour la suite. Je ne m’enferme pas en disant que si je ne peux pas le faire avec un langage, je ne le ferais pas. Au contraire.

CHAPITRE 7

Outils Utilisés

Pour les curieux, voici les outils utilisés pour les démonstrations de code contenues dans ce livre:

-pour le langage C : le logiciel Code::Blocks dans sa version 17.12

-pour le WLangage : le logiciel WinDev® de la société PC SOFT, dans sa version 24, update 4.

CHAPITRE 8

Le Mot De La Fin

C’est déjà fini. Enfin presque.

Rendez-vous à la prochaine mise à jour!

Si vous voulez savoir qui je suis, me contacter, tout est là:

http://anthonylaurito.fr

A bientôt!