Zeste de savoir

TPP — Chapitre III

20 mars 2019

Table des matières

1. Introduction

i Un mot d'abord

Lire fait partie de ce qu'on appelle la veille technologique. C'est pour cela que j'essaye, régulièrement, d'avancer dans plusieurs livres réputés du domaine informatique. Aujourd'hui, c'est le chapitre III «Les outils de base» du livre *The Pragmatic Programmer* que je vais résumer à ma façon, en me basant sur la compréhension que j'en ai. En espérant que ça vous plaise, bonne lecture.

Si l'outil de base du menuisier est le bois, pour nous autres développeurs, il s'agir du texte. En effet, n'est-ce pas là l'outil avec lequel est écrit tous les autres ? Il est donc important de bien l'utiliser et d'être efficace avec.

2. Au sommaire

— **The Power of Plain Text**. Un aperçu de ce matériau brut, pour mieux le comprendre et l'utiliser.
— **Shell Games**. Comment en tirer le meilleur parti en utilisant son ordinateur ?
— **Power Editing**. Quitte à manipuler du texte, autant le faire dans de bonnes conditions, avec des outils maitrisés sur le bout des doigts.
— **Source Code Control**. Une publicité pour les logiciels de gestion de version, bien avant l'heure qu'ils ne soient universellement répandues et utilisés.
— **Debugging**. Parce que tout le monde y passe à un moment ou à un autre.
— **Code Generators**. Tout comme le menuisier a ses plans, nous avons des générateurs de code pour gagner du temps et de l'efficacité.

3. Le pouvoir du format texte

Qu'est-ce que le format texte ? N'importe quel caractère affichable peut prétendre à ce titre. Il faut donc non seulement que le texte soit affichable, mais également compréhensible par l'utilisateur. Des exemples typiques sont HTML et XML, ou encore JSON.

L'avantage, par rapport aux données binaires, c'est qu'il est directement compréhensible, sans aide ni programme. Au contraire, les données binaires **séparent la donnée de son interprétation**. Sans une application dédiée, les données n'ont aucune signification. D'où le premier conseil du chapitre.

> Keep knowledge in plain text.
>
> *Astuce 20*

Les seuls inconvénients du format texte sont la taille et le temps pour interpréter les données par l'ordinateur qui augmentent. Peut-être est-ce qu'une des deux contraintes, voire les deux, sont inacceptables pour un certain programme, comme le format interne d'une base de données. Mais bien souvent, ces contraintes sont minimes voire nulles.

Contrairement à ce qu'on pense, **les données binaires ne sont pas plus sécurisées**. Si quelqu'un veut les modifier, il y arrivera. Pour empêcher ça, il faut chiffrer, ou bien vérifier l'intégrité des données avec un *checksum*, ou toute autre technique.

Les auteurs citent trois avantages supplémentaires à utiliser un format texte simple.

— **Garantie contre l'obsolescence**. Même si le programme de base est supprimé, ou obsolète, même si l'on ne fait plus rien des données, elles sont toujours là et prêtes à être utilisées. Pas besoin d'avoir leur format ou l'algorithme pour extraire les données d'une masse de bits.
— **Facile à manipuler**. Presque tous les outils que nous manipulons font fait pour le texte. On peut ainsi très facilement comparer plusieurs fichiers, en garder des traces, les manipuler en un clin d'œil avec des outils puissants comme le shell, etc.
— **Tests plus faciles**. Comme il s'agit de texte brut, pas besoin de passer par des logiciels intermédiaires pour modifier des données de test. On gagne ainsi en temps et en efficacité.

i Note personnelle

Je me rappelle effectivement la galère d'avoir à ouvrir le SGBD pour modifier des valeurs d'un fichier de configuration, car celles-ci étaient stockées dans la base de données. Un fichier *.conf* ou *.ini* eut été plus simple. J'avais même du développé seul un petit utilitaire pour éviter d'avoir à ouvrir Visual Studio 2005. C'était bien relou en tout cas.

4. Shell games

Le shell est **un outil formidable**. Je ne crois pas qu'il existe beaucoup (au moins sous GNU/Linux et autres UNIX) de programmes graphiques qui ne soit pas utilisables en console

également. Souvent, c'est même l'inverse, il s'agit de surcouches graphiques à des applications du shell.

> Use the power of commands shells.

Astuce 21

De plus, la combinaison de plusieurs programmes avec les *pipes* décuple sa puissance. L'auteur, pour illustrer, compare la façon de faire deux tâches différentes (même plus), l'une en passant par la GUI, l'autre par le shell. Ainsi, la liste de tous les fichiers *.c* modifiés plus récemment que le *Makefile* s'obtient facilement.

```
find . -name '*.c' -newer Makefile -print
```

Il en est de même pour obtenir la liste de tous les *packages* Java utilisés dans un projet.

```
grep '^import ' *.java |
  sed -e's/.*import *//' -e's/;.*$//' |
  sort -u > list
```

Concernant Windows, le livre donnait dans les années 1990 la solution de Cygwin. Celle-ci est peut-être toujours valable en 2019, mais Powershell et *Linux for Windows* ont apportés leurs lots d'améliorations à la situation.

Note personnelle

Des fois, le shell fait gagner beaucoup de temps. La commande `make` refuse mon *Makefile* parce que j'ai des espaces et non des tabulations ? Qu'à cela ne tienne, StackOverflow a la solution.

```
perl -pi -e 's/^  */\\t/' Makefile
```

5. Power editing

Comme déjà dit plus haut, le texte étant notre matériau de base, il faut des outils pour le manipuler sans effort. Le livre recommande ainsi de bien s'approprier un éditeur et s'y tenir. Le maîtriser par cœur, connaître tous les raccourcis et ses possibilités.

> Use a single editor well.

Astuce 22

Il faut que celui-ci présente plusieurs caractéristiques, selon les auteurs.

— **Configurable**. Les fonts, les couleurs, les raccourcis clavier, etc.
— **Extensible**. On doit pouvoir l'utiliser même avec de nouveaux langages ou de nouvelles versions de langages existants. Formulé autrement, le produit ne doit pas devenir obsolète trop vite.
— **Programmable**. À l'aide de macros ou d'un langage *built-in*.

Il rajoute également des caractéristiques propres à des langages particuliers.

— Coloration syntaxique.
— Auto-complétion.
— Auto-indentation.
— Template de documents.
— Des *features* d'IDE (compilation, debug et tout ça).

i Note personnelle

En 2019, tout ça est de base. Personne n'utilisera un IDE qui ne fournit pas ce genre de fonctionnalités. Mais gardons à l'esprit que ce livre a été écrit il y a plus de 20 ans.
De même, aujourd'hui, on assiste plutôt à une multiplication d'IDE spécialisés qu'à un IDE global sachant tout faire, même si les produits Jetbrains, par exemple, sont assez semblables entre eux, ou encore que Visual Studio propose des projets dans des domaines aussi variés que le web et Unity.

Les auteurs insistent sur les avantages et les gains de productivité que ces solutions apportent par rapport à l'usage du *notepad* de Windows, qu'à leur époque certains utilisaient pour leurs développement.

6. Gestionnaire de code source

Cette section en fera sourire plus d'un aujourd'hui, tellement les gestionnaires de code source sont devenus des réflexes. Je ne citerai donc que le conseil du moment.

> Always use source code control.
>
> *Astuce 23*

7. Debugging

On passe tous par là, à un moment ou un autre. Et parfois, il arrive que certains, ayant trouvé un bug causé par quelqu'un d'autre, vont plus se concentrer à les blâmer qu'à corriger le problème. Cependant, peu importe le responsable, il y a un problème qui demande à être corrigé.

> Fix the problem, not the blame.
>
> *Astuce 24*

Le livre recommande aussi de mettre de côté son orgueil, son égo et ses contraintes liées au projet (temps, deadline, etc) pour bien se préparer au débug. Les auteurs proposent donc une règle à se rappeler.

> Don't panic.

Astuce 25

Il est important de bien **comprendre les causes du problème** pour ne pas seulement corriger les symptômes, mais aussi la cause profonde. Pour ça, on fait du pas-à-pas, on fait des schémas, on peut expliquer à un canard en plastique aussi (ceci dit, un vrai fait l'effet également). Visualiser ce qui se passe en mémoire, l'état des données à chaque instruction est aussi d'une grande aide.

Note personnelle

Ça vous rappelle quelque chose ?

Il faut ensuite **procéder par élimination**. Les bugs causés par la base de données, le compilateur ou le système d'exploitation sont très rares, alors que la probabilité que le problème vienne de notre code est élevée. Les auteurs racontent ainsi qu'un ingénieur a passé des jours entiers à faire des *fixtures* autour d'un *SELECT* parce que celui-ci était cassé. Finalement, ils lui ont prouvé que le problème venait de son code à lui.

> "Select" isn't broken.

Astuce 26

Enfin, dans le cas des « bugs impossibles » qui n'auraient jamais du venir, le réflexe est de ne pas faire d'hypothèse mais bien **de prouver qu'un morceau de code est correct**, dans ce contexte, et que le bug ne peut pas venir de là. Il est possible, en effet, qu'on ait oublié un cas limite ou de vérifier telle ou telle valeur. Reproduire le test avec les mêmes données et le même contexte permettra de savoir si une portion de code est responsable du bug ou non.

> Don't Assume It — Prove It.

Astuce 27

Note personnelle

Une autre chose utile est d'avoir une base de données de bugs. Bien entendu, il faut que celle-ci soit de qualité, avec les contextes, les données ayant causé le problème, la portion de code incriminée, etc. Cela peut faire gagner beaucoup de temps quand un même bug revient.

8. Manipulation de texte

Maintenant, le conseil suivant, développé dans cette partie, est celui d'**apprendre à utiliser un langage de manipulation du texte** (en anglais, *Text manipulation language*). On citait déjà à l'époque Ruby, Perl et Python, ainsi que Tcl. Force est de constater que ce conseil est toujours d'actualité.

> Learn a text manipulation language.
>
> *Astuce 28*

Ils permettent un prototypage rapide, des PoC, car ils sont en général plus concis, plus fournis et plus rapide à utiliser que les autres type C++ ou Java. En guise d'exemple, les auteurs parlent d'un script Perl qui, à partir d'un fichier texte contenant un schéma de base de données, génère plusieurs choses.

— Les requêtes SQL pour créer la base.
— Des données plates pour remplissage.
— Du code C pour accéder à la base.
— Des scripts pour vérifier son intégrité.
— Des pages web contenant des schémas descriptifs et des diagrammes.
— Une version XML du schéma.

D'autres exemples encore sont cités, que je vous laisse découvrir.

9. Générateurs de code

La génération automatique réduit le risque d'erreur et fait gagner du temps. C'est pour ça que les auteurs recommandent de l'utiliser.

> Write code that writes code.
>
> *Astuce 29*

On en distingue deux types.

— Les **générateurs passifs** se lancent une fois et produisent quelque chose qui n'est pas lié au générateur. Quand un IDE génèrent les fichiers `.h` du projet, c'est passif. Quand on précalcule des tables trigonométriques aussi. Quand on convertit du Markdown en HTML également.
— Les **générateurs actifs** produisent des résultats qui dépendent de la configuration du générateur. Quand on utilise gRPC, le résultat dépend de la configuration, c'est actif. Quand on génère des classes Java correspondant au modèle de la base de données, c'est aussi actif.

Un générateur n'a pas besoin d'être compliqué. En fait, il n'est même pas obligé de produire du code. On peut très bien avoir du XML, du JSON ou encore du texte plat en sortie.