Zeste de savoir

Programmation fonctionnelle — 97 choses à savoir quand on est un programmeur

20 mars 2019

Table des matières

1. Introduction

Introduction

Il existait, il y a de cela deux-trois ans, un site dont le nom était « 97 Things Every Programmer Should Know » mais qui aujourd'hui semble avoir disparu (bien qu'on puisse trouver le Gitbook). Ce site était composé de **conseils divers et variés sur la programmation**, écrits par différents programmeurs et librement diffusé sous licence Creative Commons Attribution Non Commercial Share Alike 3.0 . Je me permets donc de partager ici avec la communauté de Zeste de Savoir des traductions de ces différents articles. N'hésitez pas à commenter.

L'article traduit aujourd'hui est Applique les principes de la programmation fonctionnelle par Edward Garson.

2. Applique les principes de la programmation fonctionnelle

La programmation fonctionnelle a, depuis peu (NdT : au moment où l'auteur a écrit), regagné l'intérêt du grand public programmeur. Une des raisons à cela est que le paradigme fonctionnel est bien adapté pour résoudre les défis posés par une industrie se tournant de plus en plus vers le multi-cœurs. Mais bien ce que soit une application importante, ce n'est pas la raison qui devrait te pousser à l'apprendre.

La maîtrise du paradigme fonctionnel peut grandement améliorer la qualité du code que tu écris, même dans d'autres contextes. Si tu le comprends vraiment et l'appliques, cela se ressentira dans tes designs, qui montreront un plus haut niveau de transparence référentielle (ou *referential transparency*).

La transparence référentielle est une propriété très désirable : elle implique qu'une fonction retourne un résultat constant pour une entrée constante, peu importe quand et où a été invoquée cette fonction. L'idée est que l'évaluation d'une fonction dépende moins, idéalement pas du tout, des effets de bords et autres états *mutables*.

Une des causes majeures de problème avec le code impératif est dû aux variables *mutables*. N'importe quel lecteur a déjà du chercher à comprendre pourquoi il n'obtenait pas le bon résultat dans une situation particulière. Une bonne sémantique de visibilité (NdT : la façon dont un langage gère la portée des variables par exemple) aide à limiter ces effets pervers, ou, tout du moins, réduire leur localisation dans le code. Mais le vrai problème serait plutôt les designs qui emploient la mutabilité de façon exagérée.

Ce n'est pas dans notre métier que nous verrons les bonnes pratiques se propager. Beaucoup d'introduction à l'orienté objet promeuvent de tels designs, montrant des exemples de graphes d'objets à longue durée de vie qui s'appellent leurs mutateurs les uns sur les autres, ce qui est dangereux. Cependant, avec l'aide du *Test Driven Development* , notamment quand on s'efforce de *Mock Roles, not Objects* , on peut éliminer toute mutabilité non nécessaire.

Le résultat est un design dont les **responsabilités sont bien réparties** entre les différents composants, dont les **fonctions sont plus petites** et agissent sur des arguments qu'on leur passe et non sur des variables globales mutables. De cette manière, il y aura moins de problèmes et ils seront plus faciles à corriger, parce qu'il sera plus facile de localiser où une valeur problématique a bien pu être introduite plutôt que de déduire le contexte particulier dans lequel cette valeur étrange est apparue. Aucune méthode ne pourra t'inculquer ces idées aussi bien que le fait d'apprendre un langage fonctionnel, puisque c'est la norme qui y règne.

Bien entendu, cette approche n'est pas la meilleure dans toutes les situations. Par exemple, dans un système orienté objet, cette façon de faire donne de meilleurs résultats avec le *Domain Driven Design* , dans le cadre du développement des *business rules*, que pour le développement d'une interface utilisateur.

Maîtriser la programmation fonctionnelle te permettra d'appliquer les leçons apprises dans d'autres domaines. Ton système orienté objet sera ainsi plus proche de sa contre-partie fonctionnelle, plus qu'on aurait pu le penser. Certains pourraient même trouver que l'orienté objet et le fonctionnel se complètent à la manière du *yin* et du *yang*.

3. Mon mot à moi

En écrivant cet article, j'ai pensé à la variable `errno` en C, qui est une variable globale. Typiquement et comme le signalait déjà *The Pragmatic Programmer*, cela empêche de l'utiliser sur une application multi-process. La programmation fonctionnelle bannit ce genre de pratique, donc le design d'une fonctionnalité similaire aurait été plus efficace avec les idées de ce paradigme en tête. Bien entendu, je sais qu'on parle d'un autre temps, que les jeunes de 20 ans ne peuvent connaître.

? Votre parole

Et vous, qu'en pensez-vous ?