Zeste de savoir

Règles de codage — 97 choses à savoir quand on est un programmeur

4 mai 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 Automatiser les règles de codage par Filip van Laenen.

2. Automatiser les règles de codage

Tu dois connaître ça. Au commencement d'un projet, tout le monde est bourré de bonnes intentions — appelle ça des « résolutions de nouveaux projets ». Bien souvent, beaucoup d'entre elles sont inscrites dans des documents, ceux dédiés aux règles de codage du projet. Lors de la réunion de démarrage, le *lead developer* parcourt le document et dans le meilleur des cas, tout le monde est d'accord pour suivre ces règles. Mais une fois que le projet est lancé, **ces bonnes intentions sont abandonnées**, une par une. Quand le projet aboutit et est livré, le code ressemble à un beau boxon et personne ne comprend comment on a pu en arriver là.

Quand est-ce que les choses ont dérapé ? Sans doute dès la réunion de démarrage. Certains membres ne faisaient pas attention. D'autres n'ont pas compris. Pire, il y en a même qui n'étaient pas d'accord et prévoyaient déjà leur rébellion contre ces normes de codage, pour imposer les leurs. Enfin, certains avaient bien compris et étaient d'accord, mais la pression était telle qu'ils ont du laisser tomber quelque chose. Eh oui, un code bien formaté ne fait pas gagner de points avec un client qui attend plus de fonctionnalités. De plus, suivre un standard peut être vraiment ennuyant si ce n'est pas automatisé. Essaye donc d'indenter une classe bordélique à la main pour tester.

Si c'est un tel problème, pourquoi s'obstiner à vouloir des standards d'abord ? Une raison qui pousse à formater le code de manière uniforme est **d'empêcher que quelqu'un ne « s'approprie » un morceau de code** en le formatant de la façon qu'il lui convient. On veut aussi **empêcher les développeurs d'appliquer des anti-patterns**, dans le but d'éviter certains bugs courants. En fait, un standard de code devrait rendre le travail sur le projet plus facile et aider à maintenir une vitesse de développement constante du début à la fin. Cela va sans dire que tout le monde doit être d'accord avec ces règles — ça ne sert à rien si un développeur utilise trois espaces pour indenter alors qu'un autre en utilise quatre.

Il existe une abondance d'outils utilisables pour produire des rapports sur la qualité du code, ainsi que pour documenter et maintenir le standard, même toute la solution ne réside pas ici. Tout cela devrait être automatisé et imposé chaque fois que possible. Voici quelques exemples.

— Faire en sorte que le formatage du code soit **partie intégrante du build**, de manière à ce que n'importe qui puisse le lancer en même temps qu'une compilation.
— Utiliser des **outils statiques d'analyse de code** pour chercher les anti-patterns et pour empêcher le build le cas échéant.
— Les configurer pour **scanner des anti-patterns spécifiques** au projet.
— Ne pas se contenter de mesurer la couverture de code par les tests, mais également les résultats de ces derniers. Si la couverture est trop faible, le build doit échouer.

Fais ceci pour chaque chose que tu considères comme importante. Tu ne pourras pas automatiser tout ce qui te tiens à cœur. Concernant ces derniers, considère que ce sont des **règles complémentaires** au standard automatisé. Accepte également que toi et tes collègues ne les suiviez pas avec autant de soin.

Enfin, le standard doit être **dynamique** et non statique. En même temps que le projet évolue, les besoins du projet changent et ce qui était une bonne décision pour le début du projet n'est peut-être plus nécessaire quelques mois après.

3. Mon mot à moi

Cela semble être une bonne idée, mais faut-il encore que les règles soient raisonnables et clairement définies. J'ai connu les revues de code avec les rejets parce que `const` était après et non avant le type en C++, pour la simple raison que cela ne plaisait pas au *lead dev*. C'est frustrant et source de mécontentement.

Pourtant, dans le même temps, la PEP 8 de Python ne m'a jamais dérangé, parce qu'elle est clairement définie et largement répandue et acceptée. Paradoxal dis-tu ?

? Votre parole

Et vous, qu'en pensez-vous ?