[Note du traducteur : ceci est une traduction en français libre et non officielle du FFF paru sur le forum]

Friday Facts N°366 - La seule façon de faire vite, c'est de faire bien !

Posté par *kovarex* le 18/06/2021

Bonjour,
Ça fait longtemps qu'on ne s'est pas vu :)

Nous avons évidemment beaucoup de choses à dire sur les changements que nous avons récemment apportés au jeu, ou que nous prévoyons d'apporter bientôt, mais nous ne voulons pas encore en parler ici.

Pourtant, il y a actuellement un sujet très pertinent pour nous et nous pouvons le partager sans révéler de changements spécifiques au jeu. Le sujet d'aujourd'hui sera assez technique et lié à la programmation, donc si vous êtes juste venu pour les nouvelles du jeu, vous pouvez sans crainte passer votre chemin.

Oncle Bob

Maintenant qu'il n'y a plus que des développeurs ici, je peux partager ma nouvelle découverte de l'Oncle Bob et son explication vraiment sympa de certains des principes fondamentaux liés à la gestion de projets de programmation, et bien plus encore. Si vous avez 8,5 heures de libre devant vous, je vous propose de regarder cette vidéo, car il y aura plus tard quelques références à ce qu'il mentionne.

Ma pensée générale était que nous maintenons la qualité du code à un niveau assez élevé, et que nous avons une méthodologie de travail raisonnablement bonne. Mais nous avons été victimes d'un aveuglement sélectif à de nombreux endroits. Il est intéressant de voir comment certains morceaux de code étaient bons dès le départ et sont restés assez bons au fil des ans, alors même qu'ils ont été beaucoup développés... alors que d'autres se sont fortement détériorés.

Et la réponse s'explique par la métaphore du rayon de cire.

Qu'est-ce qu'un rayon de cire et quel est le rapport avec la programmation, me direz-vous ? Mon grand-père était un apiculteur très enthousiaste. J'ai passé mon enfance dans notre jardin, où il fallait faire attention où l'on mettait les pieds, où l'on s'asseyait, et où il ne fallait jamais laisser traîner quoi que ce soit de sucré, car on y trouvait très vite un gros tas d'abeilles dessus. Je devais l'aider et apprendre à connaître les abeilles de temps en temps, ce que je détestais honnêtement, car je savais que je n'aurais jamais d'abeilles chez moi plus tard. Mais il avait raison sur un point : tout ce que vous apprenez vous sera utile d'une manière ou d'une autre.

L'une des tâches que l'on effectue avec des abeilles, c'est que lorsque le miel a été récolté, on met un rayon de cire dans la ruche, ce qui ressemble à ceci :

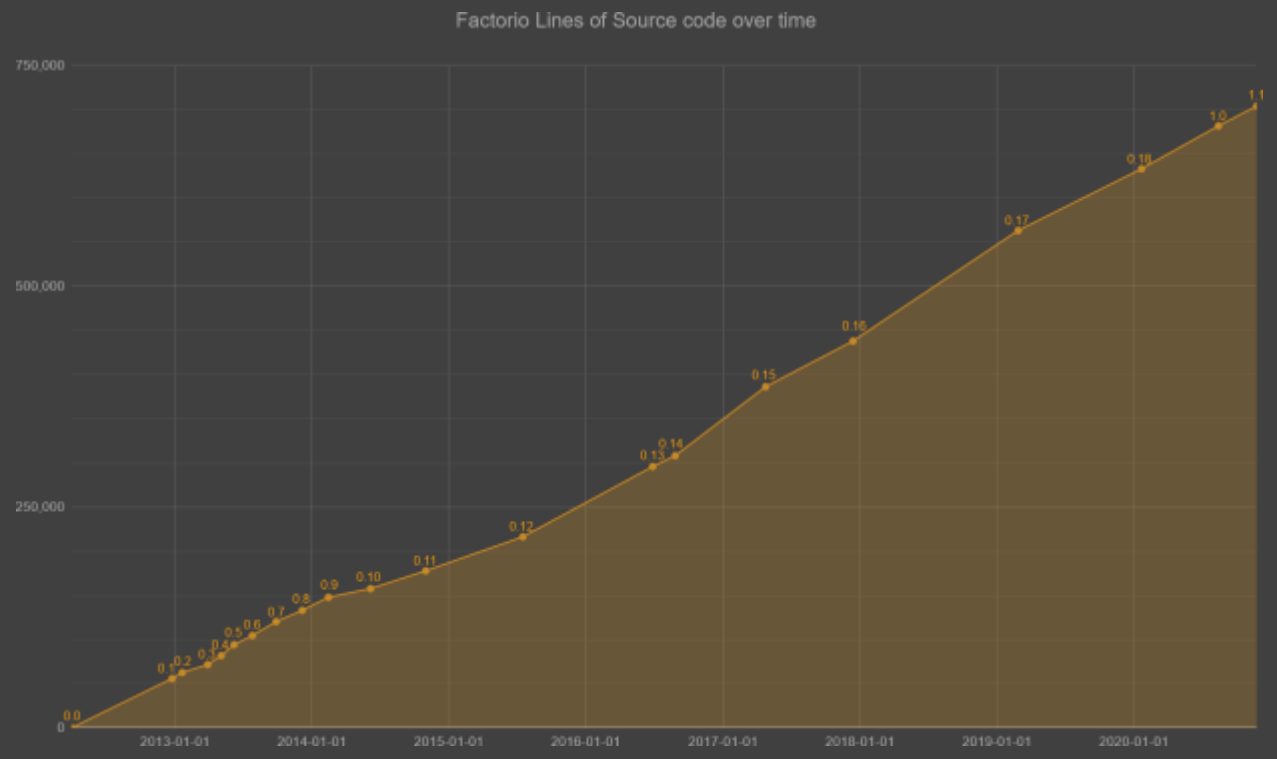
Sa fonction première est de permettre aux abeilles de construire leurs alvéoles de manière régulière et assez rapide, car il est tout à fait naturel de suivre la structure optimisée qui est déjà là. Et c'est exactement ce qui se passe avec un code dont la conception est bonne et extensible dès le départ.

D'un autre côté, il y a du code dont la conception originale était paresseuse, ou qui n'était pas censé devenir aussi complexe, et chacun des changements n'était qu'un petit ajout au désordre initial. Nous avons fini par nous habituer à l'idée que cette partie du code est tout simplement infernale, et qu'y faire de petits changements est ennuyeux. Cela implique que nous n'aimons tout simplement pas cette partie du code, et que nous voulons passer le moins de temps possible à travailler avec elle. Et le résultat est que le problème devient lentement incontrôlable.

Lorsque j'ai pris les lunettes de l'*Oncle Bob* et commencé à regarder autour de moi, j'ai rapidement identifié plusieurs endroits problématiques comme celui-ci. Ce n'est pas une coïncidence si ces endroits absorbaient une part disproportionnée du temps de développement, non seulement parce qu'il est difficile d'y effectuer des changements, mais aussi parce qu'ils sont remplis de bugs de régression et qu'ils constituent généralement une source inépuisable de problèmes.

C'est l'avantage d'avoir une entreprise qui n'est pas cotée en bourse. Imaginez que vous ayez une entreprise qui va de plus en plus lentement chaque trimestre, et que vous disiez aux actionnaires que la façon de résoudre le problème est de ne faire absolument aucune nouvelle innovation pendant un trimestre ou deux, de remanier le code, d'apprendre de nouvelles méthodologies, etc. Je doute que les actionnaires acceptent cela. Heureusement, nous n'avons pas d'actionnaires, et nous comprenons l'importance vitale de cet investissement à long terme. Non seulement dans le projet, mais aussi dans nos compétences et nos connaissances, afin de faire mieux la prochaine fois.

Voici la chronologie des lignes de code dans Factorio



Cela semblerait assez raisonnable s'il y avait le même nombre de personnes travaillant du début à la fin, mais ce n'est pas le cas. Il n'y avait que moi au tout début, et maintenant il y a 9 programmeurs. Cela pourrait s'expliquer par le fait que le jeu prend de l'ampleur et comporte de nombreuses mécaniques interconnectées, ce qui est plus difficile à maintenir. Ou cela pourrait aussi signifier que la densité du code s'est nettement améliorée. Ces deux explications ne suffisent pas à expliquer pourquoi le fait d'avoir plus de programmeurs n'entraîne pas un développement plus rapide.

Cela indique que les problèmes décrits par Oncle Bob sont pertinents aussi pour nous, et que la solution consiste en fait à améliorer la façon dont nous développons plutôt que de simplement augmenter le nombre de personnes. Une fois que nous disposerons d'une base propre, il sera beaucoup plus rapide de recruter de nouveaux programmeurs et de leur faire maîtriser le code.

Permettez-moi maintenant d'expliquer quelques exemples typiques des problèmes que nous avons rencontrés, et comment nous avons procédé pour les résoudre :

Cas n°1 : l'interaction avec l'interface graphique

Nous avons beaucoup écrit au sujet de l'interface graphique (par exemple dans le FFF-216) et de la façon dont nous avons itérativement relevé la barre de ce que nous trouvons acceptable tant du point de vue de l'utilisateur que du programmeur. Le point commun entre le FFF et le codage est que nous avons toujours sous-estimé la complexité de la logique, des styles, de la mise en page, etc. de l'interface graphique. Cela implique que le fait d'améliorer la façon dont l'interface graphique est écrite présente des gains potentiels importants.

Nous sommes satisfaits de la façon dont les objets de l'interface graphique sont structurés et disposés depuis la mise à jour 0.17. Mais au niveau du code, elle apparaît toujours beaucoup plus massive qu'elle ne devrait l'être. Le principal problème était la quantité d'endroits qu'il fallait toucher pour ajouter un élément interactif. Laissez-moi vous montrer un exemple, un simple bouton utilisé pour réinitialiser les préréglages dans la fenêtre du générateur de cartes.

Dans l'en-tête de la classe :

```
class MapGeneratorGui
{
  ...
```

Nous avions une définition de l'objet bouton.

```
...
IconButton resetPresetButton;
...
```

Dans le constructeur de *MapGenerator*, nous devions construire le bouton avec ses paramètres.

```
...
, resetPresetButton(&global->utilitySprites->reset, // normal
                    &global->utilitySprites->reset, // hovered
                    &global->utilitySprites->resetWhite, // disabled
                    global->style->toolButtonRed())
...
```

Nous devions nous enregistrer comme *listeneur* de ce bouton.

```
...
this->resetPresetButton.addActionListener(this);
...
```

Ensuite, nous devions substituer la méthode de l'*ActionListener* dans notre *MapGeneratorClass*, afin de pouvoir écouter les actions de clics.

```
...
void onMouseClick(const agui::MouseEvent& event) override;
...
```

Et enfin, nous pouvions implémenter la méthode, où nous faisons un *if/else* à travers les éléments qui nous intéressent, pour faire la logique réelle.

```
void MapGeneratorGui::onMouseClick(const agui::MouseEvent& event)
{
  if (event.getSourceWidget() == &this->resetPresetButton)
    this->onResetSettings();
  else if (event.getSourceWidget() == &this->randomizeSeedButton)
    this->randomizeSeed();
  ...
```

C'est beaucoup trop de texte standard pour un seul bouton, avec une action simple. Il y a plus de 500 endroits où nous avons enregistré des *actionListeners* dans le code, alors imaginez la quantité de texte superflu.

Nous avons remarqué que lorsque nous utilisons des fonctions anonymes pour signaler des rappels et d'autres choses similaires dans l'interface graphique, cela tend à être beaucoup plus agréable à utiliser. Et si nous en faisions le principal moyen d'utiliser l'interface graphique ?

Nous avons alors décidé de réécrire complètement la façon dont cela fonctionne, donc au lieu d'ajouter des *listeners* et un filtrage à partir des fonctions de capture d'événements, nous pouvons simplement spécifier :

```
this->resetPresetButton.onClick(this, [this]{ this->onResetSettings(); });
```

C'est déjà une grande amélioration, car l'ajout et la maintenance de la nouvelle logique ne nécessitent de regarder qu'à un seul endroit au lieu de plusieurs, et cela le rend généralement plus lisible et moins sujet aux erreurs.

Et puisque nous n'avons pas besoin de conserver l'objet pointeur pour les comparaisons, nous pouvons complètement supprimer sa définition de la classe, et la rendre anonyme à de nombreux endroits, de cette manière :

```
*this << agui::iconButton(&global->utilitySprites->reset,
                          global->style->toolButtonRed(),
                          this, [this]{ this->resetPreset(); })
```

La réécriture de tous les éléments internes de l'interface graphique (encore une fois) a été une tâche importante, mais au final, cela en valait vraiment la peine, et maintenant je ne peux plus imaginer comment nous pourrions continuer à le faire à l'ancienne. Cela a également permis de supprimer plusieurs milliers de lignes de code.

La seule façon de faire vite, c'est de faire bien !

Cas n°2 : la construction manuelle

Il y a plusieurs objectifs principaux à poursuivre lorsque vous essayez de rendre le code plus propre. La suppression des doublons dans le code est la première et la plus grande priorité. C'est raisonnablement facile à résoudre lorsque le code n'est pas bien structuré, que les fonctions sont trop longues ou que les noms sont bizarres, mais si vous avez 5 versions du même tas de code avec de légères modifications ici et là, c'est la pire des choses. C'est juste une question de temps, jusqu'à ce que les corrections de bugs/modifications ne soient appliquées qu'à certaines versions, et qu'il devienne de moins en moins évident de savoir si les différences entre celles-ci sont voulues ou fortuites.

La logique de la construction manuelle est un monstre, en raison de toutes les choses qu'elle prend déjà en charge :

[NdT : cliquez sur chacune des images pour voir l'animation]

Ensuite, toute cette logique doit être multipliée par 2 (quand on est paresseux et qu'on fait un copier-coller), car on peut avoir un bâtiment normal ou un bâtiment fantôme.

Et ensuite, vous multipliez encore par 2 cette abomination du code. Pourquoi ? Parce que nous devons aussi faire toute cette logique dans le mode de dissimulation de la latence. Cela semble déjà mauvais, mais ce n'est pas tout, puisque cette logique a été continuellement remaniée et manipulée par différentes personnes au cours de l'histoire, le cœur du code était une méthode folle et longue dont le code ressemblait à l'horizon mentionné par l'Oncle Bob.

Imaginez maintenant que vous deviez modifier quelque chose dans ce code, surtout si vous tenez compte du fait que ce code comporte naturellement de nombreux cas particuliers erronés ou corrigés uniquement dans certaines versions du code. Il s'agit d'un excellent exemple de la façon dont une conception paresseuse conduit à long terme vers une productivité plus faible.

Pour faire court, ce projet a été abordé comme un projet secondaire de mon hobby, qui a pris des semaines à être terminé, mais au final, tous les doublons ont été fusionnés, le code est bien structuré et entièrement testé. La gestion du code ne demande plus qu'une petite fraction du temps par rapport à la situation précédente, car le programmeur n'est pas obligé de lire un énorme tas de code juste pour avoir une vue d'ensemble et pouvoir modifier quoi que ce soit.

Cela me rappelle une citation de Lou après un type de restructuration similaire : "*Une fois qu'on en aura fini avec ça, ce sera vraiment un plaisir d'ajouter des choses à ce code.*". N'est-ce pas magnifique ? Ce n'est pas seulement plus efficace et moins buggé, c'est aussi plus amusant de travailler avec, et travailler sur quelque chose d'agréable permet d'aller plus vite, indépendamment des autres aspects.

La seule façon de faire vite, c'est de faire bien !

Cas n°3 : les tests d'interfaces graphiques

Non, nous ne sommes évidemment pas arrivés à ce résultat sans tests automatisés, et nous les avons déjà mentionnés plusieurs fois (FFF-29, FFF-62, FFF-288, etc.). Nous essayons continuellement d'élever le niveau des zones de code couvertes par les tests, et cela nous a conduit à couvrir une autre zone, l'interface graphique. Cela correspond à la sous-estimation constante de la quantité de précautions techniques dont l'interface graphique a besoin. Le fait qu'elle ne soit pas testée du tout fait partie de cette sous-estimation. Combien de fois il est arrivé que nous publiions une version et qu'elle se plante sur quelque chose de stupidement simple dans une interface graphique, juste parce que nous n'avions pas de test pour cliquer sur les boutons. Et finalement, il s'est avéré qu'il n'était pas difficile d'automatiser tous ces tests.

Nous avons juste un mode dans lequel l'environnement de test est créé avec des interfaces graphiques (même si les tests sont exécutés sans graphiques). Nous avons déclaré quelques méthodes d'aide, qui permettent une définition très simple de l'endroit où nous voulons que le curseur se déplace ou sur quoi nous voulons cliquer, comme ceci :

```
TEST(ClearQuickbarFilter)
{
  TestScenarioGui scenario;
  scenario.getPlayer()->quickBar[0].setID(ItemFilter("iron-plate"));
  CHECK EQUAL(scenario.player()->getQuickBar()[0].getFilter(), ItemFilter("iron-plate"));
  scenario.click(scenario.gameView->getQuickBarGui()-
>getSlot(ItemStackLocation(QuickBar::mainInventoryIndex, 0)),
                 global->controlSettings->toggleFilter);
  CHECK EQUAL(scenario.player()->getQuickBar()[0].getFilter(), ItemFilter());
}
```

La méthode de cliquage appelle ensuite les événements de bas niveau de l'entrée, de sorte que toutes les couches du traitement des événements et de la logique de l'interface graphique sont testées. C'est un exemple de test de bout en bout, qui est un sujet controversé, parce que certaines "écoles" de méthodologie de test disent que tout doit être testé séparément. Donc, dans ce cas, nous devrions théoriquement tester uniquement le fait de cliquer sur le bouton en premier, ce qui crée une *InputAction* à traiter, et ensuite, avoir un test indépendant de l'*InputAction* fonctionnant correctement. J'aime cette approche dans certains cas, mais la plupart du temps, j'aime vraiment pouvoir pénétrer toutes les couches de la logique avec seulement quelques lignes de code. (Plus d'informations dans la partie sur les dépendances de tests)

La seule façon de faire vite, c'est de faire bien !

Cas n°4 : le développement piloté par les tests (*Test driven development* ou TDD)
Je dois admettre que je ne savais pas vraiment ce que c'était jusqu'à récemment. Je pensais qu'il s'agissait d'une absurdité, car il semble vraiment peu pratique et irréaliste d'écrire d'abord tous les tests pour une fonctionnalité (sans pouvoir l'essayer ou même la compiler), puis d'essayer d'implémenter quelque chose qui la satisfasse.

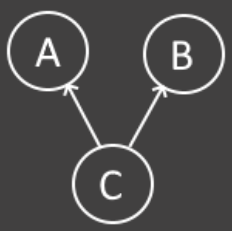
Mais cela ne correspond pas au TDD, et il a fallu qu'on me le montre à la façon "TDD pour les nuls" pour que je réalise à quel point j'avais tort.

Le TDD est en fait le passage rapide et constant entre l'extension des tests et leur réussite continue. Ainsi, lorsque vous écrivez des tests, vous écrivez du code pour les satisfaire pratiquement en même temps. Cela vous permet de tester instantanément ce que vous écrivez, et d'utiliser principalement les tests comme spécification de ce que le code devrait réellement faire, ce qui guide le processus de réflexion pour vous faire penser à la direction que vous prenez, et pour écrire du code qui est plus structuré et testable dès le début.

Après avoir réalisé ce qu'est vraiment le TDD, j'ai commencé à en être immédiatement fan. Je fais maintenant beaucoup d'efforts pour essayer de suivre la méthodologie TDD autant que possible, et pour l'imposer aux autres membres de l'équipe également. Cela semble plus lent d'écrire des tests, même pour de simples éléments de logique qui ne peuvent être que corrects, mais le test m'a prouvé que j'avais tort plusieurs fois déjà, et a évité des sessions de debugging de bas niveau ennuyeuses à brève échéance.

La seule façon de faire vite, c'est de faire bien !

Cas n°5 : les dépendances de tests
Il s'agit de la suite du sujet sur la dépendance de tests par rapport aux tests des interfaces graphiques.



Si ces tests doivent être réellement indépendants, le test C devrait comporter des copies de A et de B, de sorte que le test de C soit indépendant du système A + B fonctionnant correctement. Le consensus semble être que cela conduit à une conception plus indépendante, etc.

Cela peut s'appliquer dans de nombreux cas, mais je pense qu'il est pratiquement impossible d'adopter cette approche partout et que cela créerait beaucoup de désordre. Je ne suis pas le seul à avoir ce problème.

Par exemple, disons que nous avons un test sur les poteaux électriques qui se connectent correctement sur la carte. Mais je peux difficilement le tester si je ne sais pas auparavant si la recherche d'entités sur la carte fonctionne correctement.

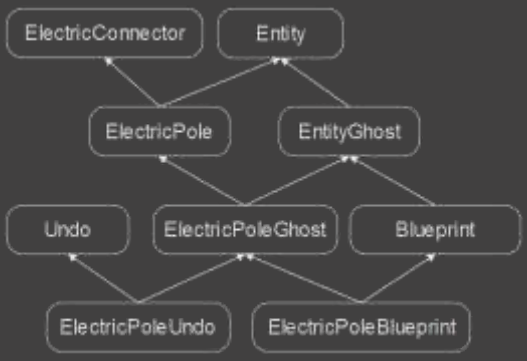
Ma conclusion est qu'avoir des dépendances de ce type est bien, tant que les dépendances sont également testées, mais le problème vient quand vous cassez quelque chose et que beaucoup de tests commencent à échouer soudainement. Lorsque vous effectuez de petits changements, l'indication oui/non des tests est suffisante, mais ce n'est pas toujours le cas, en particulier lorsque vous remaniez une structure interne, auquel cas vous vous attendez à casser beaucoup de choses, et vous devez avoir un moyen de les corriger étape par étape une fois que la compilation est à nouveau possible.

Si les tests n'ont pas de structure particulière, la situation où 100 tests échouent tous en même temps est très malheureuse, il ne vous reste plus qu'à essayer de choisir un test de façon semi-aléatoire et de commencer à le débugger. Mais c'est vraiment trompeur lorsqu'un cas de test compliqué échoue au milieu, et que vous passez un long moment à le débugger, pour vous rendre compte que c'est un bug très simple de bas niveau qui est à l'origine de l'échec.

L'objectif est assez simple, il faut que l'on me donne le cas d'échec le plus simple suite à mon changement.

Pour cela, j'ai mis en place un système simple de dépendance de tests. Les tests sont exécutés et listés de telle manière que lorsque vous débuggez et vérifiez un test, vous savez que toutes ses dépendances fonctionnent déjà correctement. J'ai essayé de chercher si d'autres personnes utilisent aussi les dépendances, et comment ils le font, et étonnamment je n'ai rien trouvé.

Voici un exemple de graphique de dépendance de tests lié aux poteaux électriques :



J'ai construit et utilisé cette structure lors du remaniement de la double logique de connexion des poteaux fantômes/réels et cela a certainement accéléré le processus pour le faire fonctionner correctement, et je suis convaincu que c'est pour nous la meilleure façon de structurer les tests dans un avenir proche. Non seulement cela rend les résultats des tests plus utiles, mais cela nous oblige à diviser les suites de tests en unités plus petites et plus spécialisées, ce qui est certainement utile aussi.

La seule façon de faire vite, c'est de faire bien !

Cas n°6 : la couverture des tests

Lorsque Boskid a rejoint l'équipe en tant que responsable du contrôle qualité, l'un de ses principaux rôles était de s'assurer que tout bug découvert était d'abord couvert par un test avant d'être corrigé, et plus généralement d'améliorer la couverture des tests de notre code. Cela a rendu les versions beaucoup plus sûres, et nous avons eu moins de bugs de régression, ce qui se traduit directement par une efficacité à long terme. Je crois fermement que cela indique et soutient clairement ce que dit l'Oncle Bob. Travailler avec des tests semble plus lent, mais c'est en fait plus rapide.

La couverture des tests est un indicateur des parties du code qui sont exécutées lorsque l'application tourne (ce qui signifie généralement l'exécution de tests dans ce contexte). Je n'avais jamais utilisé d'outil pour mesurer la couverture des tests auparavant, mais comme c'était l'un des sujets abordés par Oncle Bob, j'ai essayé de l'utiliser pour la première fois. J'ai trouvé un outil qui ne fonctionne que sous Windows, mais qui nécessite le moins d'installation possible. Il s'agit de OpenCppCoverage, qui fournit un résultat au format HTML comme celui-ci :

```
450. bool ElectricPole::disconnectRegular(ElectricPole* targetPole)
451. {
452.   if (!this->electricConnector.isConnected(&targetPole->electricConnector))
453.     return false;
454.
455.   targetPole->electricConnector.disconnect(&this->electricConnector);
456.   this->electricConnector.disconnect(&targetPole->electricConnector);
457.   this->setupOrientation();
458.   if (!this->isSimulation() && !targetPole->isSimulation())
459.   {
460.     CompactDeque<ElectricPole*> queue;
461.     this->setupNetworksAfterDisconnect(targetPole, queue);
462.   }
463.   return true;
464. }
465.
```

Il est immédiatement visible, que les deux commandes conditionnelles ne sont pas déclenchées dans les tests. Cela signifie essentiellement que soit le code n'est pas testé, donc il devrait être couvert, soit c'est du code mort, donc il devrait être supprimé. Je suis tout à fait confiant (encore une fois), que l'utilisation de cette méthode peut considérablement nous aider à écrire du code propre et de haute qualité.

La seule façon de faire vite, c'est de faire bien !

Conclusion

La seule façon de faire vite, c'est de faire bien !

Si cela vous touche, si votre réaction en lisant ceci est "*J'aimerais que mon patron ait ces priorités*". Pensez à postuler pour un emploi chez Wube !

Discutez sur nos forums

Discutez sur Reddit

S'abonner par e-mail

[NdT : Traduit avec l'aide de www.DeepL.com/Translator (version gratuite)]