NOHAR

CC BY SA

Geste de savoir

Le kata du « Bowling Game »

31 mai 2021

Table des matières

Salut!

Aujourd'hui je voudrais vous montrer une technique d'entraînement au développement que j'ai récemment adoptée. Il s'agit du *code kata*.

Cet exercice qui prend moins de 30 minutes par jour peut vous permettre de progresser dans de nombreux aspects du développement. On peut utiliser les katas pour:

— S'entraîner au *Test-Driven Development* (TDD) et au *Simple Design*,
— Apprendre à utiliser efficacement ses outils de travail,
— Mieux expliquer aux autres la façon dont on raisonne, par exemple en parlant à une peluche sur notre bureau,
— Apprendre à connaître quelqu'un au travers de sa façon de penser, et donc à **mieux coopérer avec cette personne**, en réalisant un kata "en multi-joueur", c'est-à-dire en une courte séance *pair programming*.

Son principe est de prendre un problème, peu importe si dont on en connait déjà la solution, et s'entraîner à le résoudre en réalisant chaque geste **délibérément** de bout en bout. En essayant d'optimiser toutes les micro-décisions et les gestes que nous réalisons pour converger vers cette solution. Pour ce faire, je vais vous montrer un kata très populaire de Robert C. Martin, qui consiste à calculer le score d'une partie de bowling américain.

Je vais vous montrer comment je réalise ce kata en Go en vous détaillant toutes les questions que je me pose au fur et à mesure, mais **on s'en fiche du langage et des outils : vous n'avez pas du tout besoin de connaître Go pour suivre**, juste de savoir déjà programmer dans un langage quelconque. Vous pouvez réaliser cet exercice dans n'importe quelles conditions, de préférence **les conditions dans lesquelles vous désirez vous améliorer**, en Javascript sous iOS, en C# sous Windows, en Rust sous Linux…

En l'occurrence, ce kata est aussi pour moi l'occasion de vous montrer comment on raisonne en TDD et pourquoi c'est avantageux.

1. Énoncé du problème

L'objectif de cet exercice est d'implémenter un composant qui calcule le score d'une partie de bowling américain.

Une partie se déroule en **10 tours**. À chaque tour, le joueur dispose de deux essais pour faire tomber 10 quilles disposées en triangle. Le score de base est le nombre de quilles que le joueur a réussi à faire tomber pendant le tour.

Lorsque le joueur arrive à faire tomber les dix quilles en deux lancers, cela s'appelle un **spare**. Le score d'un spare est de 10, auquel on ajoute le nombre de quilles tombées *au lancer suivant*.

Lorsque le joueur arrive à faire tomber les dix quilles en un seul lancer, cela s'appelle un **strike**. Le score d'un strike est de 10, auquel on ajoute le total de quilles tombées *aux deux lancers suivants*.

Lors du dernier tour:

— si le joueur réalise un *spare*, il dispose d'un lancer bonus pour permettre de calculer son score ;
— si le joueur réalise un *strike*, il dispose de deux lancers bonus pour permettre de calculer son score.

Cela signifie qu'une partie de bowling se termine en 21 lancers maximum (10 spare d'affilée).

Le composant que l'on cherche à implémenter n'est **pas** responsable de valider le nombre de coups joués dans une partie ni que les coups sont "légaux". On part de l'hypothèse que cette validation est réalisée *avant* d'interagir avec ce composant, et donc qu'il sera systématiquement appelé avec une entrée bien formée.

Le **véritable but** de cet exercice est de minimiser le nombre d'efforts à concéder et de questions que nous avons besoin de nous poser pour parvenir au résultat *le plus simple possible* qui satisfasse cet énoncé.

C'est pour cette raison que l'on se moque de connaître la solution et qu'il est bon de répéter ce kata quotidiennement (mais pas plus d'une fois par jour): le plus important est d'être capable d'y parvenir en réalisant délibérément chaque geste, de manière à mémoriser *les questions que l'on se pose* plutôt que *les réponses* que nous leur apportons.

2. Écrire le premier test

Le principe du TDD est de boucler rapidement sur les trois étapes suivantes:

1. RED: On écrit **le minimum de tests** possible pour que les tests échouent,
2. GREEN: On ajoute le code **le plus simple possible** pour faire passer tous les tests,
3. REFACTOR: C'est ici que l'on prend soin du code et que l'on peut factoriser ou renommer les choses, les tests garantissent que nous ne casserons pas le travail que nous avons déjà fait.

2. *Écrire le premier test*

Mais la *toute première* chose à faire est de créer un environnement (ou "projet") pour travailler.

Dans mon cas, je vais réaliser ce kata en Go, et je vais donc taper les lignes suivantes:

```
$ rm -rf ./bowling && mkdir bowling && cd bowling
$ go mod init bowling
```

Puis je vais ouvrir mon éditeur (vim) dans lequel je vais créer un fichier dans lequel je vais écrire mon premier test.

> **Le rôle des tous premiers tests** est hyper important. Ils servent à réfléchir à **l'interface publique** du code que nous allons écrire.

Le premier élément d'interface auquel nous ayons à faire est le constructeur. C'est pourquoi j'ai écrit le test suivant dans un fichier `bowling_test.go`.

```
$ rm -rf ./bowling && mkdir bowling && cd bowling
$ go mod init bowling
```

Évidemment, ce test ne compile pas puisqu'il fait appel à la fonction `NewGame()` qui n'est pas définie. Nous pouvons d'ailleurs le vérifier dans la console (dans la réalité j'utilise la commande `:GoTest` dans Vim) :

```
$ go test
# bowling [bowling.test]
./bowling_test.go:6:10: undefined: NewGame
FAIL    bowling [build failed]
```

Nous sommes donc à l'état **RED**. Notre mission est d'écrire juste ce qu'il faut de code pour que ce test passe au vert. Je vais écrire ce code dans le fichier `bowling.go` que voici:

```
$ go test
# bowling [bowling.test]
./bowling_test.go:6:10: undefined: NewGame
FAIL    bowling [build failed]
```

Je relance les tests. Ils passent avec succès: nous sommes **GREEN**. C'est l'heure de "refactoriser", sauf qu'il n'y a rien à refactoriser pour l'instant, donc notre but est d'écrire un nouveau test qui échoue.

Dans notre cas, nous voulons calculer un score de bowling. Nous savons qu'une partie de bowling se compose d'un nombre variable de lancer et nous savons que nous ne calculerons le score

que de parties finies. Nous savons aussi que nous ne passerons aucune donnée invalide à notre composant.

La partie la plus simple possible au bowling est une partie **nulle** (on envoie la boule dans la gouttière à chaque coup). C'est pourquoi j'ai rajouté le test suivant dans `bowling_test.go` pour la simuler.

```
$ go test
# bowling [bowling.test]
./bowling_test.go:6:10: undefined: NewGame
FAIL    bowling [build failed]
```

Évidemment, ce test ne compile pas, car il utilise 2 méthodes qui ne sont pas définies:

— `game.Roll(int)` qui sert à enregistrer un lancer,
— `game.Score() int` qui sert à récupérer le score final.

Bien qu'il ne compile pas, ce test m'a poussé à *réfléchir* aux fonctions que je vais écrire. C'est l'API la plus simple possible.

Nous voici donc à nouveau à l'état **RED**. Il suffit de définir ces méthodes pour que le test passe:

```
$ go test
# bowling [bowling.test]
./bowling_test.go:6:10: undefined: NewGame
FAIL    bowling [build failed]
```

Je relance les tests. Ils passent: nous sommes **GREEN**.

Regardons nos tests: le test `TestNewGame` est entièrement contenu dans le test `TestGutterGame`. Autrement dit, si ce dernier échoue, `TestNewGame` ne nous donnera aucune information supplémentaire. Ce test est donc **redondant**: il faut le supprimer. Ce que l'on peut faire, par contre, c'est factoriser ce code dans une fonction `newGame` afin de ne pas avoir à récrire cette vérification à chaque test que nous écrirons:

```
$ go test
# bowling [bowling.test]
./bowling_test.go:6:10: undefined: NewGame
FAIL    bowling [build failed]
```

Du reste, il n'y a pas de code à bouger, mais il est idiomatique en Go de documenter toutes les fonctions et types *publics* (dont le nom commence par une majuscule) que nous créons, en écrivant pour cela des commentaires spéciaux, sans quoi `golint` nous fera les gros yeux. Faisons ça dès maintenant pour ne pas avoir à y repenser plus tard.

```
$ go test
# bowling [bowling.test]
./bowling_test.go:6:10: undefined: NewGame
FAIL    bowling [build failed]
```

Et voilà, le projet est démarré.

3. Calculer le score d'une partie "normale"

Il est temps d'écrire un nouveau test.

Jusqu'à présent, nous avons réalisé une partie où on ne tombait aucune quille à chaque coup. Essayons maintenant une partie où nous tombons **une seule** quille à chaque fois. Cette partie devrait être composée de 20 lancers, et son score final devrait être 20.

Pour ce faire, j'ai *copié-collé* notre premier test puis je l'ai modifié pour donner ceci:

```
$ go test
# bowling [bowling.test]
./bowling_test.go:6:10: undefined: NewGame
FAIL    bowling [build failed]
```

Vous connaissez l'adage: «**si tu copies-colles quelque chose, c'est sûrement que tu as quelque chose à refactoriser**».

Sauf que l'on n'en est pas à l'étape où on refactorise les choses. C'est pour cette raison que j'ai laissé ce commentaire `// XXX copy-pasta`. Mon éditeur va faire ressortir ce `XXX` avec un fond coloré pour attirer mon attention et ne pas l'oublier quand il sera temps de refactoriser.

En attendant, les tests échouent, évidemment:

```
bowling_test.go|30| game.Score() should be 20, got 0.
```

Il va donc falloir faire passer ce test. La solution la plus simple est d'accumuler le score dans un attribut de la structure `Game`:

```
bowling_test.go|30| game.Score() should be 20, got 0.
```

Les tests passent, nous sommes **GREEN**. C'est maintenant que l'on refactorise nos tests.

> Le but d'une refactorisation des tests n'est pas de rendre le code le plus court possible, mais **de rendre chaque test le plus lisible possible**.

Pour cette raison, factorisons les trois étapes de chaque test:

— initialisation (`newGame(t)` fait déjà le travail),
— exercice du code (lancer `game.Roll()` dans une boucle),
— vérification (comparaison de `game.Score()` avec la valeur attendue).

Voici ce que cela donne chez moi:

```
1 bowling_test.go|30| game.Score() should be 20, got 0.
```

4. Gérer les "spares"

Ça y est, vous êtes bien échauffés? 🍏

Il faut maintenant que notre code soit capable de compter des *spares*. Pour cela, écrivons un test qui réalise un spare (6, puis 4 quilles) *et un lancer non-nul* (3 quilles), puis une partie nulle le reste du temps. Le score du spare devra être $(6 + 4) + 3 = 13$, et le core total $13 + 3 = 16$.

```
1 bowling_test.go|30| game.Score() should be 20, got 0.
```

Et bien sûr ce test échoue avec le message: `Score should be 16, got 13`.

C'est maintenant que nous allons devoir nous gratter la tête. Commençons par jeter un oeil un peu plus critique à nos méthodes:

```
1 bowling_test.go|30| game.Score() should be 20, got 0.
```

Nous avons:

— une méthode dont le nom et la documentation indiquent qu'elle enregistrent les coups, alors qu'elle **calcule le score**,
— une méthode dont le nom et la documentation indiquent qu'elle calcule le score, alors qu'elle **ne fait rien** si ce n'est retourner le score déjà calculé.

Ceci est indicatif d'un problème de *design*. Nous allons donc avoir besoin de faire des changements relativement profonds dans notre code, alors **commençons par commenter le test `TestOneSpare`**, afin de retourner à l'état **GREEN** précédent, et pouvoir changer de design tout en nous assurant que notre code fonctionne *au moins aussi bien* qu'avant.

4. Gérer les "spares"

Au lieu de ne garder que le score final en mémoire, nous devrions garder chaque coup, *puis* calculer le score final dans la méthode. En Go, on peut faire cela avec une *slice* d'entiers, dont la capacité maximale, 21 éléments, est connue dès le départ.

En modifiant mon code ainsi, le code passe à nouveau tous les tests (sauf `TestOneSpare`, oui):

```
1 bowling_test.go|30| game.Score() should be 20, got 0.
```

Je peux maintenant décommenter `TestOneSpare` et revenir à l'état **RED**.

Pour gérer les *spares*, maintenant, nous avons besoin de compter le score de la même façon que nous comptons manuellement les points au bowling, en considérant la partie comme 10 tours ayant chacun un nombre variable de lancers.

Voici comment je passe **GREEN**:

```
1 bowling_test.go|30| game.Score() should be 20, got 0.
```

Autrement dit, à chaque tour, je regarde si le tour est un spare ou non. S'il l'est, il compte comme la somme des deux lancers du tour courant et du prochain lancer, sinon, il compte normalement comme la somme des quilles tombées en deux coups.

Admettons que ce code n'est pas très lisible. Ça tombe bien, parce que c'est le moment de le refactoriser. D'abord, cette variable `i` n'est pas super bien nommée parce qu'on ne sait pas vraiment à quoi elle fait référence. Elle est utilisée pour être un indice dans le tableau de lancers, je me propose donc de la renommer `rollIndex`.

Ensuite, ces calculs polluent un peu la compréhension du code. Plutôt que d'expliquer que nous comptons des spares ou des coups réguliers dans des commentaires, faisons *du code qui se documente tout seul* en extrayant ces calculs dans des méthodes dont le nom est explicite. Voici ce que ça donne.

```
1 bowling_test.go|30| game.Score() should be 20, got 0.
```

C'est tout de même plus facile à suivre, vous ne trouvez pas?

Essayons un autre test avec les spares, maintenant, en faisant tomber 5 quilles à chaque lancer. Chaque tour va être un *spare*, et le dernier tour sera composé de 3 lancers. Le score final devrait être de 10 * (10 + 5), soit 150.

```
1 bowling_test.go|30| game.Score() should be 20, got 0.
```

Et… ce test passe tout seul, donc on en a fini avec les spares ! 🍊

5. Gérer les "strikes"

Nous entamons la dernière partie du kata. Il faut maintenant que nous soyons capables de prendre en compte les *strikes*. Et pour cela nous allons…?

Écrire un nouveau test, bien sûr! 🍊

Nous allons réaliser un *strike*, puis tomber successivement 4 et 3 quilles au prochain tour, puis complètement rater les 8 tours (16 lancers) suivants. Le score du strike devrait être 17, le score total de la partie devrait donner 24.

```
1 bowling_test.go|30| game.Score() should be 20, got 0.
```

Et ce test… échoue avec un débordement d'indice: `panic: runtime error: index out of range [19] with length 19`.

Ce n'est pas une grande surprise: un strike est un tour qui se joue **en un lancer**. Notre code ne sait pas gérer ce genre de tours. Apprenons-lui:

```
1 bowling_test.go|30| game.Score() should be 20, got 0.
```

Nous sommes **GREEN** à nouveau. Procédons à la refacto:

— Comme pour les *spares*, isolons les calculs dans des méthodes aux noms explicites,
— En Go spécifiquement, cette suite de `if {} else if {} else` peut se récrire avec un `switch` qui sera plus lisible.

```
1 bowling_test.go|30| game.Score() should be 20, got 0.
```

Voilà, un beau code bien propre. Nous pouvons maintenant écrire un dernier test, qui compte ce qui se passe lorsque l'on réalise une **partie parfaite**, c'est-à-dire, 10 *strikes* de suite, puis deux lancers "bonus" parfaits. Le score final d'une partie parfaite est 10 * (10 + 10 + 10) = 300.

```
1 bowling_test.go|30| game.Score() should be 20, got 0.
```

Et… ce test passe avec succès, donc nous n'avons plus rien à faire. Mission accomplie.

Et voilà comment on aboutit à un code, **simple**, **lisible** et **couvert de tests à 100%**.

Ce que je vous recommande *grandement* de faire, c'est d'apprendre à réaliser ce kata dans votre langage de prédilection. Lorsque vous répéterez ce kata, *ne vous contentez pas de faire du "par coeur"*: réfléchissez à chaque étape, essayez peut-être de prendre une autre décision et de dérouler le reste du kata pour voir où cela vous mène.

5. Gérer les "strikes"

Il est recommandé de travailler régulièrement sur un même kata avant d'en apprendre et d'en pratiquer un autre, car ce n'est pas la réalisation de cet exercice qui nous sert à progresser, mais *toutes les questions que l'on se pose* entre deux exécutions du même kata.