NOHAR

Geste de savoir

« Args » : un kata dense et réaliste

16 juin 2021

Table des matières

Introduction

Il y a un peu plus de deux semaines, je vous présentais un kata célèbre (le "Bowling Game") pour vous faire découvrir cette pratique d'entraînement.

La discussion qui a suivi ce billet a été plutôt riche en enseignements. En particulier, je me suis aperçu que simplement vous montrer l'exécution d'un kata ne suffisait pas vraiment à mettre en valeur l'intérêt de pratiquer les katas.

Ceci est imputable à plusieurs éléments:

— Le TDD est une discipline suffisamment peu (ou mal) connue pour qu'elle ait attiré le plus gros de l'attention,
— Je n'ai pas spécialement rendu apparent le genre de progression que l'on peut suivre d'une exécution à l'autre du kata,
— Je ne me suis pas appesanti sur ce que ce kata m'avait appris,
— Le problème en lui-même était simpliste, ce qui n'aide pas à faire comprendre ce que cette pratique peut apporter à un développeur dans la réalité quotidienne de son métier.

Eh bien soit, permettez-moi de réessayer avec cette fois un problème beaucoup plus péchu (qui prend, lui, plusieurs heures), en essayant cette fois de vous montrer, à chaque décision, le genre

de progression et de leçons que j'ai pu tirer de mes itérations successives, et surtout *leur rapport* avec des expériences réelles vécues à mon travail.

Je vais prendre pour cela un autre exercice imaginé par Uncle Bob Martin que l'on trouve détaillé en Java dans son livre *Clean Code*, et qui fait intervenir beaucoup plus de concepts réalistes que la dernière fois.

Contrairement à la dernière fois, **on s'en fout un peu moins** que mon code pour ce kata soit en Go.

En effet, je vais ici m'attarder sur des détails que j'ai délibérément cherché à travailler dans le contexte de ce langage. Par contre, comme la dernière fois, **cet exercice est transposable**: si les réflexes que je me suis entraîné ici à acquérir sont très go-esques, les situations et problématiques précises dont il est question sont universelles.

1. Reprendre le contrôle sur nos réactions

Avant d'entrer dans le vif du sujet, permettez-moi de vous raconter ce qui m'a amené à travailler sur ce kata en particulier.

1.1. L'élément déclencheur

Récemment, au boulot, j'ai dû implémenter une fonctionnalité *vraiment complexe*. Pour résumer, il s'agissait de permettre à des joueurs d'enregistrer leur adresse mail ainsi que quelques autres infos pour demander une clé d'accès à la version alpha de notre jeu, en interagissant pour ce faire avec un bot sur Discord.

Ce qui rendait ce truc complexe, c'est en particulier les éléments suivants:

— Les utilisateurs interagissent par messages privés avec le bot, et cette interaction est bien sûr "stateful" : le bot doit savoir à tout instant à quelle étape de la discussion il se situe avec chaque interlocuteur. C'est très loin du "commande-réaction" ou "requête-réponse" classique que l'on trouve dans la plupart des bots. Cela implique de modéliser une sorte d'automate.
— Le contenu de la conversation (les messages du bot) doit être facile à paramétrer au gré des *stakeholders*, en particulier ici les personnes responsables du marketing et de la communauté.
— RGPD oblige, les utilisateurs doivent pouvoir à tout moment annuler leur démarche, éditer ou supprimer les données personnelles qu'ils ont envoyées dans notre base. Le "flot d'exécution" de la conversation n'est donc pas linéaire. Au-delà de ça, un utilisateur déjà enregistré n'aura pas le même genre de conversation avec le bot qu'un utilisateur qui désire s'inscrire.
— Le bot doit potentiellement gérer des milliers de conversations simultanées[1] footnote:1 et certains utilisateurs peuvent décider de ne plus lui parler après avoir démarré un formulaire sans jamais le valider, donc le bot doit pouvoir détecter les abandons passifs de ce genre (avec un *timeout*) pour ne pas gaspiller sa RAM.

— Ce bot représente le *premier contact* des utilisateurs avec notre SI: il est hors de question que le programme ait le moindre bug ou comportement surprenant, il y va de la première impression que l'on laisse à nos joueurs et donc de l'image de ma boîte.

Pour finir cette fonctionnalité et l'envoyer en prod, j'ai annoncé une bonne semaine de marge, avec comme objectif personnel de sortir rapidement (en 2–3 jours) un bot que mes collègues pourraient ensuite essayer de casser dans tous les sens.

— Étant donné le besoin de configurabilité de la chose, il a fallu que je crée dès le départ des abstractions qui permettent de créer/modifier cette fonctionnalité de façon déclarative.
— Étant donné l'aspect critique de la qualité du code, j'étais plus résolu que jamais à le développer en TDD, en isolant cette fonctionnalité derrière des abstractions (on parle d'*inversion de dépendances*) pour pouvoir la tester en la couvrant de tests à plusieurs niveaux: des tests unitaires pour les détails (vérifier que je sais correctement vérifier la validité d'une adresse mail, ou reconnaître un pays par son code ou son nom, de façon insensible à la casse…), ainsi que des tests, unitaires également mais à plus haut niveau, qui simulaient des conversations complètes avec le bot, pour reproduire facilement des scénarios "réalistes".

Ce qui s'est passé, c'est qu'à la place des 2–3 jours que je m'étais donné, cela m'a pris **toute la semaine** pour développer cette fonctionnalité. On pourrait se dire «*5 jours au lieu de 2–3, ça va, ça arrive, on a vu pire*». Certes, et d'ailleurs, j'avais eu le nez creux en prenant une semaine de marge. Mais le problème, voyez-vous, c'est que **ce retard n'était pas imputable à une cause extérieure**.

— Je n'ai pas eu de difficulté technique imprévue à gérer ;
— Il ne manquait pas un bout de la fonctionnalité que l'on aurait oublié de spécifier à la base (pas de *scope creep*) ;

Autrement dit, la cause de ce retard, c'était une défaillance **de ma part**: j'ai perdu presque deux jours à redémarrer plusieurs fois ce code. Par corollaire, si je ne veux pas que cela se reproduise et devenir plus efficace pour les prochaines fois, il faut que je corrige ce qui m'a "foutu dedans", c'est-à-dire qu'il faut que je corrige ma propre défaillance.

1.2. Analyse critique du problème

Ce que je décris ici est une difficulté courante lorsque l'on apprend le TDD.

Lorsque l'on *test-drive*, on construit une solution par petites touches incrémentales très rapides. Ce que l'on construit à un instant donné est "solidifié": ça marche et c'est testé, donc ça ne cassera plus. Le truc, c'est que pour ce faire, on ne pense pas n'importe comment: *on s'impose une discipline* pour résoudre notre problème, qui prend la forme de la célèbre boucle "Red - Green - Refactor". Cette démarche permet de jouir d'un lot d'avantages absolument monstrueux que la littérature (notamment Kent Beck et Uncle Bob Martin) a déjà longuement décrit, mais cela présuppose aussi que l'on se pose, dans ce contexte plus que jamais, *les bonnes questions au bon moment*, et que l'on ait dès le début du développement de bons réflexes.

1. [2]footnote:1 Dans la réalité, ça sera probablement beaucoup moins que ça, mais ça ne change rien au problème.

C'est un peu comme aux échecs:

La façon dont on "ouvre" le problème conditionne la facilité avec laquelle on le résout.

En l'occurrence, c'était la première fois que j'essayais de faire du TDD sur une fonctionnalité comportant plusieurs composants (pensez "classes" en POO) avec des abstractions internes qui permettent de composer facilement en permettant de substituer certains objets les uns aux autres (comme des "widgets"), avec une gestion d'erreurs non-triviale. *Ce n'est pas le plus courant* quand on fait du backend. Depuis 2 ans que je fais du Go au travail, le composant le plus péchu qu'il m'avait été donné de développer est un contrôleur Kubernetes, et même ce genre de composant s'articule autour d'une architecture *d'une profondeur bien moindre* que ce que je faisais là.

Ce qui a causé mes redémarrages successifs, c'est notamment que je n'avais pas de réflexes assez solides pour décider de l'ordre dans lequel j'allais écrire mes tests et mon code. Ce n'est pas forcément la seule raison, mais si cette lacune n'avait pas ouvert une brèche, les autres problèmes n'auraient pas eu l'occasion de s'y engouffrer [3]footnote:2. Surtout, **c'est quelque chose que l'on peut maîtriser** avec de l'entraînement !

1.3. L'intérêt de se donner un cadre contrôlé

Dans les commentaires du précédent billet, la réaction la plus sceptique était celle de @Gabbro que je me permets de citer ici:

> Une idée fondamentale du kata, c'est, si j'ai bien compris, de réfléchir à notre pratique et se donner les moyens d'essayer des choses dans un environnement cadré pour voir si c'est efficace. Ce à quoi je réponds: comment peux-tu savoir si ça marche en condition réelle si tu essaies en condition cadrée?

Ma réponse, plutôt laconique, était que cette question posait le problème à l'envers, et que la pratique des katas repose plutôt sur le phénomène inverse:

Comment peut-on croire que ça marche en conditions cadrées si ça ne marche pas en conditions réelles?

J'ai bien senti que cette formulation toute seule était un coup d'épée dans l'eau. Je ne m'étais pas étendu sur le sujet parce que pour ce faire, j'avais besoin d'amener mon argumentaire avec *une problématique réelle*. Ça tombe bien, nous y sommes, et c'est donc maintenant que je vais pouvoir expliciter ce point.

Ici, le développement de ma *vraie* feature, pour mon *vrai* boulot, dans mes *vraies* conditions de travail, a subi un foirage qui a pratiquement doublé le temps de résolution du problème. Quelque chose n'a donc "pas marché" en conditions réelles. J'ai facilement pu déterminer que la cause de ce foirage venait de moi et, en vertu du déterminisme que je viens d'expliquer, en déduire que ça venait de la façon dont j'avais abordé mon problème et réagi à différentes étapes de sa résolution. Pour ce faire, j'ai pris le cadre d'un exercice connu dont les problématiques (les "facteurs de complexité") sont similaires à ma fonctionnalité réelle (gestion d'erreur non-triviale

2. [4]footnote:2 Ça a l'air évident quand je le raconte comme ça, mais sachez que ce qui m'a permis d'arriver à mettre le doigt aussi précisément sur la cause de mon problème, c'est… justement d'avoir eu l'occasion de réfléchir en exécutant le kata dont je parle dans ce billet.

et polymorphisme interne), et je me suis employé à le résoudre dans un contexte technique similaire: en TDD et en Go. En conditions cadrées, donc.

Il s'est alors passé quelque chose de remarquable: au premier essai, **j'ai complètement foiré cet exercice de la même manière que ma fonctionnalité réelle**. J'ai dû recommencer mon code plusieurs fois, j'ai du effacer complètement mes tests et les reprendre dans un ordre différent, j'ai dû passer *plusieurs heures* à faire, défaire et refaire mon "ouverture" avant de trouver comment démarrer correctement cet exercice.

Dès lors, j'ai pu pousser un soupir de soulagement. *J'avais trouvé un exercice qui reproduisait mon problème*. Alors j'ai entrepris de bosser sur ce kata, lentement, en prenant bien soin de noter mentalement et peser chaque décision et ses conséquences. Je l'ai bien sûr répété plusieurs fois, en prenant des décisions différentes, pour trouver quelle approche me poussait à écrire, à chaque instant, **un code dont la lecture m'inciterait plus naturellement à me poser les bonnes questions et prendre les bonnes décisions pour la suite**. Bien évidemment, si vous en êtes à lire ce billet aujourd'hui, c'est parce qu'en travaillant avec ce kata, j'ai réussi à mettre le doigt sur mon erreur et à en tirer de très précieuses leçons pour le résoudre et ne plus tomber dans les mêmes pièges à l'avenir.

Une simple analyse rétrospective de mon problème du boulot m'a donné la certitude que si j'avais compris cette leçon plus tôt, j'aurais *vraiment* tenu mon délai de 2–3 jours.

> Si cela fonctionne aussi bien dans ce cas, c'est dû à la nature du TDD.

En discutant avec diverses personnes aux profils très divers, dont ma femme qui est psychologue, il s'avère que le TDD touche un phénomène qui dépasse largement le cadre de la programmation. Par exemple, il y a un parallèle absolument fascinant entre ce qui se passe lorsque l'on travaille sur un kata en TDD, et les bienfaits avérés de la «méditation pleine conscience»: en ralentissant pour se centrer sur notre propre ressenti, on apprend à identifier nos émotions, et donc à réagir "en connaissance de cause" à une situation donnée car on reconnaît l'état mental dans lequel elle nous place. En particulier, cela nous apprend à ne plus "mal" réagir d'instinct, et à placer notre réaction sur un plan beaucoup plus rationnel: c'est à ce moment-là que l'on peut faire appel à notre expérience et, grosso-modo, nous demander *comment nous voulons réagir*.

Bien sûr, ce genre de pratique n'est pas forcément valable pour tout, pour tous, dans tous les contextes, mais **le TDD s'y prête énormément**, car l'un des bénéfices naturels d'appliquer la discipline du TDD est que l'on doit prendre une micro-décision rationnelle à chaque tour de la boucle de feedback, soit toutes les 30 à 90 secondes. Chacune de ces décisions étant rationnelle, celle-ci est *délibérée* (il est facile de la noter mentalement, de s'en souvenir, et même de se rappeler pourquoi on l'a prise). Chacun de ces tours de boucle est donc un "checkpoint". Et c'est ça qui rend cette discipline aussi efficace: elle nous permet d'avoir le contrôle de ce que l'on fait **à chaque minute** de notre travail. Les katas permettent de nous entraîner à exercer naturellement ce contrôle en sachant ce que nous faisons, et donc maîtriser parfaitement notre façon de bosser.

1.4. Allez, il faut bien que quelqu'un la sorte…

Autrement dit, si vous me permettez cette comparaison *cheesy* qui n'attend que ça de sortir depuis tout à l'heure:

> En programmation, le TDD est une discipline professionnelle tout comme le kung fu est une discipline martiale, et dans les deux cas, les katas sont une forme de méditation qui nous permet d'affiner notre discipline, en apprenant à *mieux se connaître* pour devenir *maître de soi*.

Voilà, c'est dit. C'est un peu débile et je n'aime pas trop ce parallèle à cause de sa consonnance *bullshit* particulière, mais le fait est que ce parallèle est avéré, qu'il est explicable, observable, et même prouvable dans une certaine mesure.

Allez, passons à notre exercice de guerriers du code, maintenant. On va travailler notre technique du Gaufre.

FIGURE 1.1. – Illustration par Maria Letta distribuée sous les termes de la Free Gophers Licence v1.0

2. Énoncé du problème

Le but de cet exercice est d'implémenter un parseur pour lire facilement des arguments en ligne de commande.

L'idée n'est pas tant de faire une CLI complète à la POSIX que de réaliser quelque chose de très simple à utiliser. Typiquement, on veut pouvoir spécifier et parser très facilement ce genre de cas:

```
1 myserver -h 127.0.0.1 -p 8080 -v
```

Où :

— `-h` accepte un argument sous forme d'une chaîne de caractères,
— `-p` accepte un entier,
— `-v` est un simple flag booléen (`false` par défaut, `true` si présent).

On doit pouvoir passer les paramètres de la ligne de commande dans n'importe quel ordre. On doit pouvoir également les omettre: ils sont tous optionnels. La syntaxe de base revient à séparer le flag de sa valeur par une espace: les syntaxes POSIX comme `-h=localhost` ou `-p8080` ne font pas partie du scope de l'exercice.

Selon les énoncés, ou votre goût si ça vous amuse, vous pouvez parfaitement rajouter des types différents (des `double`, des dates, des durées, des fichiers…), mais comme nous allons le voir, une fois que l'on supporte *correctement* ces 2–3 types de base, toute la difficulté de l'exercice est déjà résolue.

L'interface que l'on expose à l'utilisateur de ce composant est laissée libre. Cependant, celle présentée par Uncle Bob dans son livre est particulièrement intéressante à reproduire: il utilise pour ce faire un DSL. Autrement dit, le parseur pour la ligne de commande que je montre juste au-dessus peut être entièrement spécifié avec le schéma suivant `v,p#,h*`, où:

- Les paramètres acceptés sont séparés par des virgules,
- Chaque paramètre ne peut être composé que d'une seule lettre, minuscule ou majuscule,
- Un symbole optionnel donne le type du flag :
  - Par défaut, les paramètres sont des booléens,
  - Un `#` indique un paramètre entier,
  - Un `*` indique une chaîne de caractères.

Le composant doit bien sûr gérer proprement tous les cas d'erreur qui peuvent se présenter. Ce que veut dire "gérer correctement les erreurs" est laissé à votre libre appréciation, selon votre contexte et notamment le langage dans lequel vous implémentez la solution.

Enfin, l'objectif est d'aboutir à un code *bien conçu*, c'est-à-dire qu'il doit être facile d'étendre ce parseur avec de nouvelles fonctionnalités.

3. Un démarrage critique

Je pense que vous l'aurez compris, c'est **ici et maintenant** que tout va se jouer. Au tout début.

… Enfin, dans quelques secondes en fait. Avant, il faut démarrer le projet. 🍊

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

Voilà, c'est **ici et maintenant que tout l'exercice se joue**!

3.1. Le Premier-Test-Qui-Ne-Compile-Pas

Si vous êtes un peu familier du TDD (par exemple si avez lu le précédent billet), vous savez que tout commence par l'écriture d'un tout premier test *qui ne compile pas*. Dans les exercices un peu simples comme celui du Bowling ou une décomposition en facteurs premiers, cette étape est souvent très simple, avec peu, voire aucune possibilité de se planter. Dans cet exercice en revanche, c'est réellement ce que l'on va écrire dans ce test qui conditionne le succès ou l'échec

(ou le *succès-mais-en-galérant-comme-c'est-pas-permis*) de toute l'opération. Enfin, sans vouloir vous mettre la pression, hein! 🍊

Permettez-moi de *beaucoup* m'attarder sur ce passage. Quitte à prendre des raccourcis par la suite quand l'exo sera bien lancé sur ses rails.

3.1.1. Un code facile à tester c'est un code facile à utiliser, mais pas l'inverse

L'enjeu du premier test, c'est de décider l'interface du code que nous allons écrire. Dans le cadre de cet exercice, surtout quand on a déjà utilisé des bibliothèques comme `argparse` en Python, ou encore `flags` ou `cobra` en Go, il est **très facile** d'imaginer quelque chose qui s'utilise en deux temps.

D'abord, on spécifie les paramètres de la ligne de commande, ce qui nous permet de créer un parseur, disons avec une fonction comme:

```
1 $ rm -rf ./cli && mkdir cli && cd cli && go mod init cli && vim
```

Ensuite, on appelle une méthode de notre parseur avec les arguments de la ligne de commande pour récupérer une structure qui nous donne accès aux arguments passés par l'utilisateur du programme:


```
1 $ rm -rf ./cli && mkdir cli && cd cli && go mod init cli && vim
```


C'est tout à fait logique, comme raisonnement!

? Mais est-ce que c'est vraiment le meilleur moyen d'aborder notre problème?

Vous vous en doutez, la réponse est **non**. C'est dans ce piège que je suis tombé la toute première fois. C'est aussi dans ce genre de piège que je suis tombé quand j'ai développé mon bot. Si je devais résumer ce qui rend vraiment cette approche nulle, c'est qu'elle revient à modéliser le "*problem space*" plutôt que le "*solution space*". Mais avant de vous montrer une vraie solution, poursuivons quand même ce raisonnement en avance rapide pour vous montrer où ça va coincer.

Quand on réfléchit de cette façon, on réfléchit à deux fonctions distinctes:

— Le parsing du schéma,
— Le parsing des arguments grâce au parseur décrit par le schéma.

Et naturellement, dès que l'on pense à deux fonctions distinctes, on commence à penser à **plusieurs tests distincts**. On se retrouve donc à écrire, disons, un test `TestNewParser` et un test `TestParse`.

Ce faisant, on va foncer tête baissée dans l'écriture du parseur de schéma avec ses jolis tests. À ce moment-là, un élément devrait nous mettre la puce à l'oreille: on ne peut pas vraiment tester

le schéma si on ne le confronte pas à *ce qu'il va servir à parser*. Mais admettons que cela nous passe au-dessus de la tête. On commence alors à écrire du code qui va tester la seule chose que nous ayons à nous mettre sous la dent : *l'état interne du parseur* produit par notre schéma. Ça aussi, ça pue, mais admettons que l'on ne s'en rende pas compte. On a maintenant un beau "parseur de schéma" totalement couvert de tests. Ce parseur de schéma gère des tas de types différents, des tas de cas différents, et il génère un parseur d'arguments qui… ne marche pas !

Et évidemment qu'il ne marche pas, puisqu'on ne l'a pas encore testé: il nous reste encore tout ce code à taper, ce qui va nous obliger à modifier, revoir, détruire tout ce que l'on a fait auparavant au fur et à mesure que se présentent des cas nouveaux.

… Bref, c'est pourri ! Alors on pousse un gros juron et on recommence tout depuis zéro, puisque rien de ce que l'on n'a écrit pour le moment ne nous permettra d'avancer.

Et cette fois on écrit les deux tests en même temps que le code:

— On ajoute un test pour notre parseur de schéma,
— On écrit le code qui passe ce test,
— On ajoute un test correspondant pour notre parseur de ligne de commande,
— On écrit le code qui passe ce test,
— On recommence avec un test pour le parseur de schéma…

Et on finit alors par écrire des tests redondants, par tomber sur des comportements où on n'est plus trop sûrs dans quel test il faut les spécifier, et tout devient horriblement brouillon, les tests sont incompréhensibles, et on se rend compte qu'ils sont incompréhensibles *parce qu'ils sont distincts*.

Alors on efface tout à nouveau, on pousse un juron encore plus gros, et cette fois on remonte encore plus loin dans notre raisonnement.

Contrairement à ce que l'on pourrait croire intuitivement, notre composant n'a pas deux entrées distinctes. Il n'en a *qu'une seule*, et celle-ci est composée de deux éléments.

Un cas d'utilisation, c'est une paire `(schéma, args)`. C'est le comportement de **l'ensemble du programme** face à cette entrée unique que l'on veut tester.

Autrement dit, ce que l'on va tester, et donc programmer, c'est **une fonction unique**:

```
1 $ rm -rf ./cli && mkdir cli && cd cli && go mod init cli && vim
```

Cette fonction va nous retourner une structure `Flags` qui nous donne un accès en lecture aux arguments passés à la ligne de commande. Et ça sera encore plus simple à utiliser que notre module `argparse` préféré!

Et en bonus, si à l'avenir on veut présenter notre composant à l'utilisateur suivant une interface plus classique, "en deux temps", alors on aura déjà un ensemble de tests cohérents et une fonction utilitaire qui nous permettront de refactoriser ce code et le découper naturellement, sans casser le moindre cas d'utilisation, en un rien de temps.

Et en bonus du bonus, vu que l'on va prendre le temps de refactoriser notre code en moyenne toutes les minutes grâce au TDD, cela reviendra certainement juste à rendre publique une méthode qui existera déjà !

Autrement dit, si vous voulez vous aussi éviter de perdre des heures ou des jours à redémarrer votre code depuis 0 en TDD, la leçon à retenir ici c'est qu'il faut penser à l'interface la plus simple possible **pour tester facilement le code**, ce qui le rendra automatiquement facile à utiliser. Alors que l'inverse n'est pas forcément vrai!

3.1.2. Par quel test commencer ?

Dans de nombreux autres katas, on n'a pas vraiment de cas d'erreurs à gérer. Ici, oui. Alors la question se pose: *Qu'est-ce que je teste en premier: le "happy path" ou les cas d'erreur?* Et si la réponse était évidente, j'ose espérer que je ne me serais pas vautré dessus. Mais ça aussi c'est une erreur que j'ai commise en situation réelle, faute d'expérience et de réflexes clairs.

D'après ma modeste expérience, la meilleure réponse que j'aie à vous proposer, c'est qu'il faut tester en premier les cas qui sont censés "terminer le plus vite", ceux où il manque une donnée à traiter, ceux où l'exécution est court-circuitée… Le plus souvent (mais pas toujours) ce sont des cas d'erreur. Autrement dit, il faut toujours essayer d'écrire le test pour le cas qui nous demandera d'écrire le moins de code possible.

Formulé encore autrement, le prochain test à écrire, c'est **le test qui va minimiser la durée de la boucle Red-Green-Refactor**.

En ce qui nous concerne, le cas le plus simple à tester en premier, c'est l'entrée nulle:

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

Notez que je ne me contente pas de la première ligne-qui-ne-compile pas, mais que j'ai écrit un véritable test qui opère une vérification sur le retour de la fonction: ici, l'erreur que je m'attends à recevoir.

Nous ne sommes pas encore vraiment RED, écrivons d'abord du code pour que le test compile mais échoue.

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

Essayons:

```
cli_test.go|11| Expected invalid schema, got <nil>
```

Ça y est, nous sommes, RED. Passons GREEN:

```
cli_test.go|11| Expected invalid schema, got <nil>
```

Le test passe.

Notons que j'utilise une fonctionnalité assez récente qui date de Go 1.13, et qui consiste à retourner une erreur qui *wrappe* une autre erreur bien définie (avec le placeholder `%w`), et à tester que l'erreur que nous cherchons est bien *wrappée* par celle que nous récupérons. Dans un langage orienté classes avec de l'héritage et des exceptions, cela reviendrait à tester que nous avons bien rattrapé une exception qui hérite d'un type auquel on s'attend.

Il est maintenant temps de refactoriser. Il est évident que des erreurs de type `ErrInvalidSchema` vont être retournées dans plein de cas différents à l'avenir. Il n'est donc pas prématuré de créer une fonction utilitaire qui va nous permettre d'en faciliter la création. De même, un autre réflexe que j'ai adopté en Go est d'écrire directement le commentaire de doc de tous les éléments publics que je viens de créer pour que `golint` soit content.

```
cli_test.go|11| Expected invalid schema, got <nil>
```

3.2. Un pattern bien commode pour tester du Go

Il existe un *pattern* d'écriture de tests sur lequel je suis tombé l'autre jour au détour d'un article, et que j'ai spécifiquement voulu essayer sur ce kata. Ce pattern, on le doit à l'incontournable Brad Fitzpatrick (vous connaissez memcached? OpenID? C'est lui !) et on peut notamment le retrouver dans les tests du package `net/http/httptest` de la bibliothèque standard de Go.

Concrètement, nous avons déjà suffisamment défini notre API pour connaître la structure qui sera commune à *tous nos tests*. En effet, nous avons une fonction principale dont les entrées et les sorties sont bien définies. Nous pouvons donc nous dire que chaque test sera composé:

— D'un nom qui décrit ce que l'on teste,
— D'un `schema`,
— D'une liste d'arguments,
— Et de vérifications courantes sur les valeurs de retour de `ParseArgs`, que nous pouvons abstraire dans des fonctions.

Dans ces conditions, j'ai récrit notre premier test comme ceci:

```
cli_test.go|11| Expected invalid schema, got <nil>
```

Je ne vais pas détailler tout ce code (qui est quelque peu alambiqué). Si vous ne faites pas de Go, remarquez simplement que notre test est défini de façon on ne peut plus claire par l'ensemble de lignes que j'ai surlignées.

Dorénavant, pour ajouter des tests, nous allons le plus souvent avoir besoin d'ajouter des nouvelles entrées similaires à celle-ci dans notre slice `testCases`, et, beaucoup plus ponctuellement, des fonctions qui sont soit des `checkFunc` (c'est-à-dire des fonctions qui prennent le retour de `ParseArgs` en entrée et crachent optionnellement une erreur), soit des fonctions qui créent des `checkFunc` (comme `isError`).

Quand un test échouera, cela nous affichera un message parfaitement clair, comme le suivant:

```
Case: Empty schema
Schema: ""
Args: []
Message: Expected error invalid schema, got <nil>
```

Ça y est, nous sommes lancés!

4. Avance rapide !

Bon, ce billet commence à être bigrement long, alors on va mettre un petit coup d'avance rapide pour arriver au prochain point réellement intéressant!

Pour ce faire, je vais simplement expliquer l'ordre dans lequel j'ajoute mes tests dans les cas triviaux.

4.1. Spécifier le format des flags

Pour choisir le prochain test, on le choix entre les chemins "court-circuités" suivants:

— D'autres cas d'erreur quand le schéma comporte des flags mal formés,
— Un happy path.
— Un cas d'erreur sur le format des arguments.

À vrai dire, vu la tête actuelle de notre code, le premier chemin (continuer à spécifier des cas de `SchemaError`) me ferait écrire des tests qui vont tous automatiquement passer sans que je n'aie rien à changer dans le code "de production", et je me retrouverai à un moment à devoir implémenter un code qui fait passer tous ces tests à la fois, ce qui ruine tout l'intérêt du TDD. Il faut donc que j'écrive au moins un "happy path" avant de m'y lancer.

```
Case: Empty schema
Schema: ""
Args: []
Message: Expected error invalid schema, got <nil>
```

Ici, `noError` vérifie simplement que `ParseArgs` retourne une erreur nulle.

Pour faire passer ce test, il suffit d'ajouter un simple `if`. Maintenant, je peux continuer à spécifier ce qui est acceptable dans le schéma. Par contre, je vais considérer que le but du jeu est au départ de gérer **uniquement les booléens**, c'est-à-dire les flags non-typés. Je ne toucherai donc pas aux annotations de type pour l'instant.

Ajoutons un schéma comportant plusieurs champs, dont un vide:

```
Case: Empty schema
Schema: ""
Args: []
Message: Expected error invalid schema, got <nil>
```

Cela nous pousse à séparer le schéma sur les virgules et itérer sur les champs pour vérifier que chaque champ n'est pas vide. Remarquez que puisque `argList` ne nous intéresse pas ici, je peux me dispenser de le spécifier (il vaudra `nil` par défaut).

Passons maintenant à un champ qui a un nom invalide.

```
Case: Empty schema
Schema: ""
Args: []
Message: Expected error invalid schema, got <nil>
```

Vérifions quand même que l'on accepte les majuscules et les minuscules, et même plusieurs flags distincts à la fois:

```
Case: Empty schema
Schema: ""
Args: []
Message: Expected error invalid schema, got <nil>
```

Maintenant, prenons un dernier cas un peu vicieux avant de passer aux arguments de la ligne de commande:

```
Case: Empty schema
Schema: ""
Args: []
Message: Expected error invalid schema, got <nil>
```

Pour faire passer ce test, nous allons devoir ajouter un état à notre type `Flags`: une hashmap qui associe à chaque nom de flag un booléen (puisque nous ne gérons que les booléens pour l'instant), qui vaudra `false` par défaut. En fait, je vais aller encore plus loin, en décidant que le type `Flags` n'est d'autre qu'un alias pour un `map[string]bool`.

On remarque alors que la fonction `ParseArgs` ment: pour l'instant, elle ne parse que le schéma. Nous devrions donc isoler tout son contenu dans une méthode `parseSchema` du type `Flags` avant de passer à la suite.

Voici où nous en sommes:

```
Case: Empty schema
Schema: ""
Args: []
Message: Expected error invalid schema, got <nil>
```

4.2. Gérer erreurs sur la ligne de commande

Allez, entamons une seconde avance-rapide, jusqu'à ce que nous soyons en mesure de gérer correctement tous les flags booléens.

Comme pour le parsing du schéma, commençons par les cas d'erreur. Par exemple le cas où l'on passe un argument qui ne commence pas par `-`, ou celui où l'argument est évidemment trop long, ou encore celui où le flag n'existe pas.

```
Case: Empty schema
Schema: ""
Args: []
Message: Expected error invalid schema, got <nil>
```

Cela va me demander de créer la fonction utilitaire `argv` (qui n'est rien d'autre qu'un `split()`), ainsi que l'erreur `ErrInvalidArgument` (et bien sûr une fonction `invalidArgument()` pour la wrapper).

4.3. Un cas nominal, enfin !

Et maintenant, implémentons un vrai happy path où l'utilisateur entre *enfin* un flag booléen qui existe.

Pour cela, nous allons avoir besoin de définir l'interface pour récupérer les arguments passés par l'utilisateur. Nous allons donc devoir définir une nouvelle méthode publique de notre type `Flag`:

```
Case: Empty schema
Schema: ""
Args: []
Message: Expected error invalid schema, got <nil>
```

Cette méthode nous va nous retourner la valeur booléenne du flag `flagName`, ou bien une erreur non-nulle si *aucun flag booléen n'existe avec ce nom*. Cette dernière précision est importante. C'est aujourd'hui la première fois que je décide de l'implémenter comme cela, c'est-à-dire avec une gestion d'erreurs *très stricte*.

L'autre alternative, en Go, serait de retourner la valeur "zéro" du type que nous demandons, mais «qui peut le plus peut le moins». Autrement dit, si nous savons gérer ce cas, alors rien ne nous empêcherait de rajouter plus tard des méthodes supplémentaires pour récupérer les flags avec renvoi d'une valeur par défaut si celui-ci n'existe pas ou n'a pas le bon type.

Pour ce faire, je vais créer trois nouvelles fonctions utilitaires pour décrire les tests:

— `hasBool(flagName string, want bool) error` permet de vérifier qu'un flag booléen a bien la valeur attendue,
— `hasNoBool(flagName) error` permet de vérifier que `flags.GetBool(flagName)` retourne bien une erreur.
— `expect(checks ...checkFunc) checkFunc` permet de chaîner les fonctions de vérification sur un cas de test.

L'erreur qui sera renvoyée en cas de requête d'un flag qui n'existe pas sera `ErrNoSuchFlag`.

Cela va me permettre d'écrire le test suivant:

```
Case: Empty schema
Schema: ""
Args: []
Message: Expected error invalid schema, got <nil>
```

Bon, j'admets volontiers que ce cas teste beaucoup de choses en même temps. D'un autre côté, c'est ni plus ni moins un "getter" que l'on teste ici, donc il ne faudrait pas non plus pousser le zèle à l'excès!

Voici le code (sans les helpers triviaux) qui permet de faire passer tous les tests jusqu'à maintenant.

```
Case: Empty schema
Schema: ""
Args: []
Message: Expected error invalid schema, got <nil>
```

Nous avons maintenant fait plus des deux tiers de l'exercice. Il ne nous reste plus qu'un petit passage délicat à passer avant le sprint final. 🍊

5. Gérer des types de flags arbitraires

Maintenant que nous savons parfaitement gérer les booléens, il est temps de passer à un nouveau type de flags. Mais plutôt que d'écrire tout de suite un test, faisons une pause pour réfléchir à notre code et anticiper les changements imminents.

Remarquez les quatre lignes que j'ai surlignées dans le précédent code. Ce sont les quatre endroits où notre code est spécifique aux flags booléens. Ce sont les seuls, certes, mais **il y en a quatre**.

Là où je voudrais en venir, c'est que si nous fonçons de façon bête et méchante dans l'implémentation des autres types, nous risquons de devoir modifier notre code *à quatre endroits distincts* du code pour chaque nouveau type que nous allons rajouter. C'est plutôt inélégant, et même si nous n'allons jamais pouvoir respecter tout à fait strictement le *open-closed principle* avec ce code, il serait préférable de découpler la variété des types du code qui s'occupe du parsing et de la gestion des erreurs.

5.1. Casser le couplage

Pour cela, nous allons définir une abstraction et utiliser du polymorphisme, et il n'y a pas 36 façons de faire cela en Go : plutôt que de rendre notre code dépendant d'un type concret (`bool`), nous allons masquer cette dépendance **derrière une interface**.

Profitons donc du fait que nous passons actuellement tous les tests pour refactoriser ce code.

— Là où nous stockons des booléens dans notre hashmap, nous allons plutôt nous référer à un type `abstractFlag`, et nous lui déléguerons le travail dans la méthode `GetBool` plutôt que de le renvoyer immédiatement,
— Là où nous stockons `false` comme valeur par défaut d'un flag existant, nous allons isoler le symbole du type (potentiellement vide) du flag, et passer celui-ci à un genre de *factory* qui instanciera la bonne implémentation d'`abstractFlag`,
— Là où nous stockons `true` quand nous croisons un flag que l'on connaît dans la méthode qui parse les arguments, nous allons appeler la méthode `Parse` de notre `abstractFlag`, en anticipant que certains flags vont prélever deux éléments (le flag et sa valeur), et que cette opération peut échouer en cas d'erreur de conversion.

Cela nous donne l'abstraction suivante:

```
Case: Empty schema
Schema: ""
Args: []
Message: Expected error invalid schema, got <nil>
```

En principe, on ne devrait jamais nommer une interface `AbstractQuelquechose` en Go.

Une interface n'est rien qu'un type, et cela devrait être transparent pour l'utilisateur. Néanmoins, ici, cette interface est *privée*: le fait de préciser `abstract` dans le nom sert simplement à rendre explicite mon intention, et ne sera jamais visible par les utilisateurs de ce code.

Maintenant que ceci est posé, implémentons cette interface pour le type `bool`:

```
Case: Empty schema
Schema: ""
Args: []
Message: Expected error invalid schema, got <nil>
```

Et découplons maintenant le reste du code:

```
Case: Empty schema
Schema: ""
Args: []
Message: Expected error invalid schema, got <nil>
```

Les tests passent toujours. Mission accomplie.

5.2. Ajouter le type string

Entre `string` et `int`, ce qui nous demandera le moins de travail sera `string` (puisqu'il n'y a aucune conversion à effectuer lorsque l'on stockera la valeur). Cependant avant d'ajouter ce type, écrivons un test qui nous manque:

```
Case: Empty schema
Schema: ""
Args: []
Message: Expected error invalid schema, got <nil>
```

En toute rigueur, on aurait dû écrire ce test avant de rajouter le cas `default` dans la fonction `newFlag`. C'est même ce que j'ai fait dans la réalité. Vu que la suite est triviale, permettez-moi de faire une avance rapide jusqu'à la fin.

Ajoutons le type `string`.

```
Case: Empty schema
Schema: ""
Args: []
Message: Expected error invalid schema, got <nil>
```

Je ne pense pas avoir besoin de vous détailler ce que font les fonctions utilitaires `hasString` et `hasNoString` puisque c'est la même chose que pour les booléens.

Remarquons qu'un nouveau cas d'erreur est possible ici, lorsque l'on oublie de spécifier une valeur:

```
Case: Empty schema
Schema: ""
Args: []
Message: Expected error invalid schema, got <nil>
```

Enfin, ajoutons les flags entiers, en remarquant que nous avons cette fois à gérer le cas supplémentaire d'un échec de la conversion en `int`:

```
Case: Empty schema
Schema: ""
Args: []
Message: Expected error invalid schema, got <nil>
```

Voici les implémentations de `abstractFlag` correspondantes:

```
Case: Empty schema
Schema: ""
Args: []
Message: Expected error invalid schema, got <nil>
```

Remarquez que je me sers ici d'un type `defaultFlag` que j'embarque dans toutes les implémentations concrètes afin de ne pas avoir à répéter mon code.

Enfin, un petit coup de coverage me montre justement que ces 3 méthodes de `defaultFlag` ne sont justement jamais appelée durant les tests, ce à quoi il est très facile de remédier avec ce dernier cas:

```
Case: Empty schema
Schema: ""
Args: []
Message: Expected error invalid schema, got <nil>
```

Nous en avons maintenant terminé avec ce kata !

Conclusion

Et voilà le travail! Ce kata est long. **Très** long.

Cependant, comme vous le voyez, il n'y a pas vraiment besoin d'aller jusqu'au bout pour en découvrir les aspects les plus importants: une fois que l'on a trouvé l'abstraction qui va bien, la fin se fait un peu en mode "pilote automatique", contrairement au démarrage qui peut très vite partir en vrille.

À vrai dire, cette implémentation ne me satisfait pas encore tout à fait: en effet, en cas d'erreur de type (si on demande `GetBool("i")` alors que le flag `-i` est un entier), l'erreur ne nous précise pas le nom du flag incriminé. Cela dit, ce ne serait pas bien compliqué à changer…

En tout cas, j'espère que vous comprenez mieux maintenant l'intérêt de ce style d'exercice, en particulier dans le contexte du *Test-Driven Development*. 🍍

Liste des abréviations

DSL Domain Specific Language. 7

TDD Test-Driven Development. 1