

Beste de savoir

TPP — Chapitre II

20 mars 2019

Table des matières

1.	Introduction	1
2.	Au sommaire	1
3.	Les méfaits de la duplication	2
4.	Orthogonalité	3
5.	Réversibilité	4
6.	Tracer bullets	4
7.	Prototypes et post-notes	5
8.	Langage dédié	6
9.	Estimations	7

1. Introduction



Un mot d'abord

Lire fait partie de ce qu'on appelle la veille technologique. C'est pour cela que j'essaye, régulièrement, d'avancer dans plusieurs livres réputés du domaine informatique. Aujourd'hui, c'est le chapitre II « **Une approche pragmatique** » du livre *The Pragmatic Programmer* que je vais résumer à ma façon, en me basant sur la compréhension que j'en ai. En espérant que ça vous plaise, bonne lecture.

Le chapitre commence en nous expliquant qu'il existe plusieurs astuces et trucs, applicables à différents stades de développement, principes universels mais peu connus du grand commun des développeur car n'étant pas écrits noirs sur blanc, tel quel. Le but du chapitre est donc d'en parler.

2. Au sommaire

- **The evils of duplication.** Ici, on aborde des problèmes causés par la duplication de la connaissance en de multiples endroits dans le code et des différentes
- **Orthogonality.** Quand un morceau de connaissance est divisé entre plusieurs morceaux de code, cela peut amener à des problèmes réduisant la qualité et l'évolutivité du code.
- **Reversibility.** Les environnements dans lesquels les projets évoluent changent et il est important de bien isoler ces derniers.
- **Tracer Bullets.** Rassembler au même endroit les besoins, les tests de design et le code d'implémentation, dans la suite des points précédents.
- **Prototypes and Post-it.** Suite direct du point précédent, comment utiliser le prototype pour tester des architectures, des algorithmes, des interfaces et des idées.

3. Les méfaits de la duplication

- **Domain Languages.** Ou comment programmer avec le vocabulaire métier, du domaine d'application.
- **Estimating.** Une partie de notre travail consiste à évaluer le temps que prendra une tâche et de le communiquer. Quels conseils peuvent-ils nous aider ?

3. Les méfaits de la duplication

La connaissance, dans un projet informatique, n'est pas stable, mais passe au contraire son temps à évoluer. C'est ce qui fait que notre métier consiste beaucoup en maintenance, réorganisation, réécriture, refactoring. La maintenance n'est absolument pas une étape qui démarre quand le projet est en production, mais dès la phase de codage, dès qu'une évolution ou correction de la compréhension qu'on avait est nécessaire.

Pour éviter la duplication de cette connaissance dans divers endroits, les auteurs donnent un principe simple (l'astuce 11), qui est connu sous le nom de **DRY** [↗](#) .

Chaque morceau de connaissance doit avoir une représentation, au sein d'un système, unique, non-ambigüe et faisant autorité.

Astuce p.27

Un morceau dupliqué doit être changé et corrigé à de multiples endroits, ce qui signifie que, tôt ou tard, l'un de ces endroits sera oublié et nous aurons plusieurs versions contradictoires d'un même bout d'information.

Si la duplication existe, elle peut avoir plusieurs causes.

- **Duplication imposée.** Il semble qu'il n'y ait pas le choix et que l'environnement nous impose de le faire. On donne l'exemple d'une structure de donnée qui doit être dupliquée parce qu'un back et un front ne sont pas dans les mêmes langages.
- **Duplication par inadvertance.** Le développeur ne se rend pas compte qu'il duplique.
- **Duplication par impatience.** Ça semble plus facile, le développeur est feignant, il prend la solution de court-terme. Exemple avec le passage à l'an 2000 et les problèmes résultant, parfois, de développeurs trop fainéants pour prévoir des dates à quatre chiffres et non deux.
- **Duplication inter-développeur.** On parle ici de duplication soit au sein de la même équipe, soit entre plusieurs équipes. Les auteurs parlent d'un ensemble de programmes pour l'État américain qui avaient tous des mécanismes de validation des numéros de sécurité sociale différents.

Pour le dernier point, des bibliothèques communes et facilement accessibles peuvent être d'une grande aide, d'où le conseil donné en fin de la section.

Make it easy to reuse.

Astuce 12

4. Orthogonalité


En mathématiques, deux droites se coupant à angle droit sont dites **orthogonales**. Se déplacer sur l'une des lignes n'a aucun effet sur l'autre. En informatique, pareillement, deux modules sont dits orthogonaux si des modifications sur l'un n'affectent pas l'autre. S'ils ne le sont pas, même une modification minimale peut, de façon apparemment illogique, casser quelque chose dans un module qui ne semble rien avoir à faire avec le premier.

Eliminate effects between unrelated things.

Astuce 13

On y gagne plusieurs choses à se lancer dans cette quête.

- **Productivité augmentée.** Le temps de développement et de test s'en trouve réduit. Les composants développés sont plus facilement réutilisables et modifiables.
- **Risques réduits.** Des morceaux de code bien séparés ont moins de chances d'être contaminés. Le système dans son ensemble est moins fragile et mieux testé. Cela aide aussi à réduire le *vendor lock-in* et la dépendance aux bibliothèques tierces.

La [loi de Déméter](#)  devrait ainsi nous guider dans l'écriture du code, en ne dévoilant au reste du monde que ce qu'un module est sensé dévoiler et pas plus. On évitera, dans le même genre, les variables globales. On se méfiera aussi des fonctions qui partagent trop de code en commun, typiquement celles qui sont identiques mais où seul l'algorithme central varie.

Cependant, ce principe s'applique-t-il à d'autres choses que le code lui-même ? Les auteurs donnent plusieurs points.

D'abord, dans la **gestion d'une équipe**. On connaît tous le syndrome de la réunionite, avec ces réunions à rallonge qui bien souvent ne nous concernent que peu voire pas du tout. Souvent, cela vient du fait qu'on ne sait pas qui peut être affecté par telle ou telle chose. Gérer et modifier ce genre d'équipes relève de la gestion de projet, mais le test suivant est révélateur : pour une modification à faire, combien de personnes vont être impactées ? Plus le nombre grandit, moins l'équipe est orthogonale.

Ensuite, le **design**. Si l'on change drastiquement les besoins derrière une fonction en particulier, combien de modules seront affectés ? Rendre une fonction d'entrée sensible à la casse ne devrait pas modifier d'un cheveu le comportement du module de paiement, tout comme un bouton de GUI ne devrait jamais influencer le module d'accès à la base de données.

Continuons avec les **toolkits** et les **bibliothèques**. Changer une bibliothèque externe impose-t-il des changements qui n'ont pas lieu d'être ? Si l'on devait changer le fournisseur de base de données, seul la couche d'accès devrait bouger. Si la moitié des contrôleurs doivent au passage être réécrits, ce n'est pas orthogonal.

i

Note personnelle

Je me rappelle, dans mon premier travail, qu'ils étaient liés à Oracle de manière quasi-définitive car une bonne partie de la logique métier était écrite sous forme de procédures PL/SQL. Rien de mieux pour s'assurer l'enfermement logiciel.

5. Réversibilité

Abordons le dernier point qui est **le test**. Chaque module devrait avoir ses propres tests et ne pas dépendre d'autres tests pour ça. Si une large partie du programme est nécessaire pour un simple test, alors c'est une preuve qu'il y a de trop forts couplages.

i

Note personnelle

On pourrait aussi y rajouter des données communes et utilisables par tous. Rien de tel que des données de dev et d'UAT différentes pour susciter des remontés de bugs et des incompréhensions.

5. Réversibilité

Si aujourd'hui $x = 2$, demain il peut valoir 5 puis -3 après-demain. En informatique, rien ne dure toujours, il ne faut donc pas trop s'appuyer sur des faits, car ils peuvent changer. Cependant, certaines décisions, comme le choix de la base de données ou du fournisseur de cloud sont lourdes de conséquences et peuvent ne pas être évidentes à changer après coup. Chaque décision prise entraîne en effet une réduction du champ de possibilité et une version de moins en moins flexible du futur.

Il faut donc prendre en compte la notion de **réversibilité**. Dans un code conçu avec cette idée en tête, passer d'une base de donnée A vers une base de donnée B, ou d'une architecture client-lourd à serveur / client ne devrait pas être trop compliqué ni trop long.

i

Note personnelle

Typiquement, ce n'était pas le cas d'un des contrôleurs de l'application mobile, lors de mon premier travail. Ici était mélangé, dans la même classe, commandes SQL stockées dans des `string`, code métier, manipulation de l'interface. Un beau bazar !

There are no final decisions.

Astuce 14

De la même manière, ce principe s'applique en architecture et en déploiement. Le livre parle de [CORBA](#) pour permettre de réécrire un programme sans modifier un autre qui communique avec lui. On pourrait citer aujourd'hui RPC ou une API REST. De même, si le programme est prévu pour Windows, il faut déterminer quelle version et s'il est possible que cela évolue.

6. Tracer bullets

Comment toucher une cible dans le noir ? On peut faire des calculs complexes pour extrapoler sa position. On peut aussi se crispier sur la gâchette et tirer au hasard un peu partout. Ou bien utiliser des traceurs, qui laissent une trainée visible du fusil à la cible.

7. Prototypes et post-notes

Un projet informatique revient souvent à vouloir toucher une cible dans le noir. Et l'une des façons de s'y prendre consiste à sur-spécifier le projet, ce qui est lourd et consommateur de temps. L'autre approche, c'est d'utiliser des traceurs.

Use tracer bullets to find the target.

Astuce 15

Mais qu'est-ce que c'est ? **Un code destiné à la production mais pas encore fonctionnel.** Il est conçu avec les mêmes contraintes qu'un projet classique (gestion des erreurs, logging, etc), simplement qu'il lui manque encore des parties pour être 100% fonctionnel d'un point de vue métier. Il n'est pas destiné à être jeté au bout de quelques jours / semaines de développement pour repartir sur une base propre.

i

Note personnelle

Typiquement, on est ici à 100% dans **la démarche agile**. L'approche considérée comme classique à la fin des années 1990 était plutôt la division en modules et fonctions, assemblés à la toute fin pour, seulement à ce moment, avoir un livrable à présenter au client.

- Les utilisateurs ont quelque chose à voir rapidement.
- Les développeurs ont un cadre dans lequel travailler.
- L'intégration est régulière et donc plus rapide et facile.
- Le progrès est plus facilement mesurable et donc cela nourrit la motivation.

Si on rate la cible, pas grave, on a un *feedback* rapide et on peut vite changer la trajectoire et ajuster le coup.

Il ne faut cependant pas confondre le concept précédent avec le prototypage. La distinction est sans doute plus facile à voir et comprendre en 2019 qu'en 1999, mais rappelons-la quand même. Le prototype est là pour **tester un aspect spécifique du système final**. C'est du code jetable, qui n'a aucune vocation à être pérennisé.

Le traceur lui est comme **un squelette sur lequel bâtir**. Cela peut consister en un algorithme naïf dans un premier temps, mais qui rend le programme fonctionnel. On prendra ensuite le temps de repasser dessus pour l'améliorer.

7. Prototypes et post-notes

Le prototypage nous permet de tester plusieurs combinaisons, de détecter les risques, de réduire les coûts et de prendre des mesures préventives. Il n'est pas obligé d'être sous forme de programme. Ainsi, un *workflow* peut être prototypé avec des post-it ou sur un tableau blanc.

Un prototype peut **ignorer certains détails**. Typiquement, si l'on veut tester plusieurs variantes d'interface, on va se passer des règles business derrière et coder en dur quelques valeurs. À l'inverse, des prototypes visant à comparer des algorithmes peuvent se passer d'interface. Et tout est bon à prototyper si cela nous aide à mieux cerner les besoins et risques.

- Architecture.
- Nouvelles fonctionnalités pour un système existant.

8. Langage dédié

- La structure ou le contenu de données externes.
- Des outils tiers.
- Des problèmes de performance.
- Du design d'UI.

Prototype to learn.

Astuce 16

On peut, par sa nature, se permettre plusieurs infractions.

- **Exactitude.** N'importe quelle donnée, aussi stupide soit-elle, peut faire l'affaire.
- **Complétude.** Pas grave si certaines fonctionnalités sont absentes, comme les entrées, un certain menu, etc.
- **Résistance.** Pas de temps à perdre avec la gestion des erreurs. Le programme peut lamentablement se vautrer, ce n'est pas un soucis.
- **Style.** Comme le code est jetable, pas la peine de le documenter ou de coder dans un style propre et clair. On peut ici se permettre de la duplication ou autre.

i

Note personnelle

Il faut évidemment faire très attention à bien préciser que le prototype n'a pas pour destination la production mais bien la poubelle. Sinon c'est un coup à générer de la dette technique en grande quantité.

8. Langage dédié

Il est important d'être en phase avec le métier. Cela passe notamment par un **vocabulaire commun**. Mais on peut pousser plus loin en définissant un langage spécifique au domaine. Il n'a pas besoin d'être exécutable. Il suffit simplement que les deux parties se mettent d'accord. Bien entendu, rien n'empêche d'implémenter ce langage pour en faire un vrai langage exécutable. Ainsi, les spécifications deviennent le code.

Program close to the problem domain.

Astuce 17

Un *Domain Specific Language* [↗](#) peut être **interne** s'il s'agit d'un langage qui sera traduit dans un autre (Python, Scala, C++ ou autre), ou bien **externe** si c'est pour produire un programme externe, indépendant du programme qui l'a généré.

i

Note personnelle

Ici, on touche typiquement au **Behavior Driven Development** [↗](#). L'outil Cucumber [↗](#) est typiquement un DSL interne qui traduit un langage spécifique vers du Ruby.



- 1 Scenario: Eric wants to withdraw money from his bank account at an ATM Given Eric has a valid Credit or Debit card
- 2 And his account balance is \$100
- 3 When he inserts his card
- 4 And withdraws \$45
- 5 Then the ATM should return \$45
- 6 And his account balance is \$55

C'est un moyen simple pour avoir un vocabulaire commun avec le métier et ici, de générer des tests.

9. Estimations

Estimer est un exercice délicat puisqu'il nous manque parfois des informations pour donner une réponse exacte. Pourtant, c'est une compétence nécessaire.

Estimate to avoid surprises.

Astuce 18

Si grand-mère demande quand on compte arriver, c'est pour savoir si elle prépare plutôt le déjeuner ou plutôt le dîner. Par contre, quelqu'un coincé sous l'eau en train de se noyer réclame une réponse dans l'ordre de la seconde. Cela souligne **l'importance des unités**.

Si tu réponds que le projet est prêt dans 130 jours, on s'attend à ce que ça soit le cas et les retards ne seront pas tolérés. À l'opposé, si on estime le délais à environ 6 mois, alors on s'attendra à un livrable d'ici 5 à 7 mois. L'unité à donner va donc varier en fonction des cas.

Durée	Estimation à donner en
1-15 jours	Jours
3-8 semaines	Semaines
8-30 semaines	Mois
+30 semaines	Bien réfléchir avant de répondre

Il s'agit cependant d'un exercice délicat et d'une compétence qui s'acquiert surtout par l'expérience. En attendant, on peut s'aider en demandant à des personnes qui sont déjà passées par là, qui ont déjà eu ce genre d'estimations à faire.

On peut notamment utiliser les chiffrages des itérations précédentes pour s'aider à mieux chiffrer l'itération actuelle. C'est d'ailleurs le conseil qui conclut le chapitre.

Iterate the schedule with the code.

Astuce 19