Ksass'Peuk
mehdidou99

Zeste de savoir

Causerie++ - Episode 4

4 mai 2019

Table des matières

1. Introduction 1
2. C++, OO et Alan Kay 1
   - 2.1. L'OO selon Alan Kay 2
   - 2.2. Analyse de la définition 3
   - 2.3. La leçon à retenir 5
   - 2.4. Relativisons 6
   - 2.5. Bonus 7
3. Conclusion 7

1. Introduction

Salut les agrumes ! Bienvenue dans ce nouvel épisode de Causerie++. Si vous avez raté l'épisode précédent, le voici .

Aujourd'hui, on ne va parler des Concepts comme je l'avais promis[1]. En effet, un sujet très intéressant m'est tombé dessus, et je n'ai pas eu le temps de rédiger les deux. On va donc discuter d'Orienté Objet, parce que j'ai eu l'occasion de pas mal y réfléchir ces deux dernières semaines.

Désolé pour le style un peu décousu de cet épisode, j'ai eu une grosse semaine et j'ai dû le finir à la va-vite. J'espère que vous le trouverez quand-même intéressant !

— Merci à Arius, Situphen et surtout Ksass'Peuk pour m'avoir aidé à traduire Alan Kay.
— Merci à Ksass'Peuk pour sa collaboration sur cet épisode !

2. C++, OO et Alan Kay

A la base, j'avais prévu de parler de l'enseignement du C++ à la suite d'une suggestion de @tleb. Mais pour trouver quelques lectures avant de me lancer sur la partie Orienté Objet de mon cours C++, je suis allé voir cette page , car je me souvenais que @lmghs y avait laissé quelques liens :

1. Je décale ça au prochain épisode du coup.

> Pour Kay, je te passe en vrac tout ce que j'ai :
> — Dr. Alan Kay on the Meaning of "Object-Oriented Programming", 2003
> — The Deep Insights of Alan Kay
> — Retracing Original Object-Oriented Programming
>
> *lmghs*

> Une interview plus récente de Kay : http ://www.drdobbs.com/architecture-and-design/interview-with-alan-kay/240003442
>
> *lmghs*

Et ils se sont avérés tellement intéressants que j'ai changé le sujet de cette section. Pour la discussion sur l'enseignement, ça attendra !

Trêve de bavardages, entrons dans le vif du sujet : **si on lit ces contenus en gardant l'esprit ouvert, cela apporte un point de vue très intéressant sur l'OO**. En effet, le concept de système orienté objet tel qu'Alan Kay le conçoit est assez différent de ce qu'on appelle maintenant POO dans les langages *mainstream*.

2.1. L'OO selon Alan Kay

Je ne voulais pas trop citer Alan Kay pour éviter que cet épisode ne se transforme en un énième article sur ses phrases cultes, mais je vais quand-même donner telle quelle une courte définition traduisant ce qu'il entendait par POO :

> OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things.
>
> *Alan Kay*

Pour les anglophobes, je tente une traduction, mais dans la suite on utilisera la version originale :

> Pour moi, la POO se résume uniquement à l'échange de messages ; la rétention, la protection et la dissimulation locales des états ; et une résolution d'appels au plus tard[2].
>
> *Alan Kay mal traduit*

Il cite Internet comme exemple, et le décrit comme l'unique système existant vraiment orienté objet à ces yeux. Et pour cause, Internet est admirablement bien décrit par cette définition, les objets étant des ordinateurs ou assimilés, et les messages étant... les messages. Le *late-binding* est parfaitement respecté puisque l'on peut changer/ajouter n'importe quel objet au système dynamiquement, sans avoir besoin de le relancer[3].

Au début, je me suis dit que c'était une façon de voir les choses qui n'avait aucun rapport avec l'OO comme on l'envisage aujourd'hui, en particulier en C++. D'autant que le style très borné/autoritaire d'Alan Kay n'aide pas[4].

Mais après réflexion, j'ai eu un déclic : le *class oriented programming* - au sens de « OO actuel » - partage en fait de nombreuses idées avec l'OO d'Alan Kay. Ce n'est pas des voies qui divergent complètement, bien au contraire.

2.2. Analyse de la définition

Pour appuyer mon ressenti, décortiquons la définition :

> only messaging

Cette idée traduit, à mon sens, un principe d'abstraction : du moment qu'un objet à qui un message est envoyé peut le comprendre et y réagir correctement, on n'a pas besoin de savoir comment il marche ou ce qu'il est, mais seulement comment communiquer avec lui. En fait, c'est juste une version *hardcore* du DIP, ou *Dependency Inversion Principle* (Principe d'Inversion des dépendances) :

> Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau. Les deux doivent dépendre d'abstractions. Les abstractions ne doivent pas dépendre des détails. Les détails doivent dépendre des abstractions.
>
> *Wikipédia*

Aux yeux d'une entité interagissant avec un objet « façon Kay », cet objet n'est qu'une abstraction, une entité complète et fermée capable de recevoir/réagir/répondre à des messages. Bref, c'est un mini-ordinateur dans un mini-internet, et c'est une application ultra-orthodoxe du DIP.

En C++, on peut essayer de respecter le DIP aussi bien par une approche dynamique (avec des interfaces au sens *class oriented programming*, i.e avec du sous-typage) que statique (avec des templates).

Je laisse telle quelle une note de Ksass'Peuk sur ce point :

note KP

Et pour ça, les langages objets à prototypes sont bien mieux équipés puisqu'ils forcent purement et simplement ce comportement. Dans un langage classe oriented, on est forcé de passer par d'autres machination pour s'en sortir. Mais dans l'idée, une manière simple de régler le problème est ne jamais dépendre de quoi que ce soit d'autres que des interfaces. En revanche, c'est complètement incompatible avec la notion de valeur. Et dans un langage qui serait purement objet, mais l'addition de deux entiers devrait imposer l'échange d'un nombre complètement dingue de messages parce qu'aucun des deux objets ne pourrait avoir accès à la structure de l'autre, donc sa signification.

Merci à lui pour ce complément intéressant !

> local retention and protection and hiding of state-process

A mon sens, cette idée se retrouve lorsque l'on dit qu'il faut penser en termes de services et non plus de données. En C++, cela se fait à travers l'encapsulation, dont les enjeux sont bien traduits par ce bout de phrase :

— *local retention* : Ici, *local* veut dire « à l'échelle de l'objet » ; les données « d'état » sont donc stockées dans l'objet dont elles décrivent l'état, logique. Les deux autres points cachent plus de sens.
— *local protection* : Cela me fait directement penser à la protection des invariants de l'objet, encore un aspect dont la traduction en C++ est l'encapsulation.
— *local hiding* : A mon sens, cette troisième « variation sur thème » est plus orientée du côté de l'abstraction, et est en ce sens la plus proche de l'aspect *messaging* dont j'ai parlé juste avant. C'est ce point dont la déclinaison renvoie le plus directement à « penser en termes de services ».

> extreme late-binding of all things

C'est la partie dont le C++ peut le moins s'approcher. En effet, la solution pour pouvoir tout résoudre au plus tard est un système de typage dynamique. En C++, on est limité par le typage statique sur ce point.

Les fonctions virtuelles sont un moyen d'obtenir une forme extrêmement édulcorée de *late-binding*, mais il est très limité et n'atteint pas du tout l'universalité mentionnée par Kay.

La partie « dynamique » de ce propos est donc hors de portée, modulo ce que l'on vient de dire.

Ksass'Peuk explique dans la note qui suit comment émuler un tel dynamisme ; sa méthode revient en fait à simuler un typage dynamique, seule solution connue à ce jour pour atteindre cet objectif :

note KP

On peut se dépêtrer d'un aspect en généralisant encore ce que j'ai dit plutôt à propos des interfaces. Si l'on veut pouvoir changer en permanence les comportements dynamiquement. Il nous suffit ne dépendre que d'interfaces mais en plus ces interfaces ne sont composées que de std : :function. De cette manière, tout est dynamique.
Pour aller encore plus loin, et permettre l'ajout de méthodes ou encore le changement de nombre de paramètres il faut ruser un peu mais c'est aussi faisable. Il suffit de rendre tous ces éléments nommés et d'avoir non plus une liste de std : :function mais une liste associative "nom de méthode" -> std : :function qui prennent en paramètre une liste d'éléments non typés (mais avec suffisamment d'info pour reconstruire les types). En somme, on se retrouve à implémenter une simulation d'un typage dynamique.

Par contre, j'ai l'impression que le *late-binding* traduit aussi l'idée que les objets sont interchangeables et que remplacer un objet par un autre devrait être possible du moment qu'ils peuvent recevoir/réagir/répondre aux mêmes messages. Du point de vue de la conception, il y a aussi un peu de « l'abstraction c'est bien » là-dedans.

Ksass'Peuk explique qu'on peut même aller plus loin :

note KP

Ils n'ont même pas besoin de savoir répondre au même messages : regarde sur internet si un serveur d'une machine tombe, on aura juste un message d'erreur, ça continue à marcher mais en scénario d'erreur.

Donc pour ce point, en C++, on peut très difficilement atteindre le côté pratique de la chose, mais l'idée qu'il y a derrière en termes de conception est souhaitable.

Ksass'Peuk relativise cependant ce dernier point :

note KP

Oui et non. Un tel système devient de manière intrinsèque très difficilement prévisible et par conséquent il devient très complexe de vérifier son bon fonctionnement. Tu pourras assez facilement vérifier des propriétés de sûreté (et encore c'est relatif). Mais pour ce qui est d'assurer la sécurité ou que le système globale remplit une certaine propriété fonctionnelle, tu es marron puisque tu ne peux quasiment rien tirer de tes objets qui peuvent à tout moment changer.
Typiquement Alan Kay se tamponne du LSP comme de son dernier slip. Alors c'est résilient, c'est sûr, mais pour déduire quoi que ce soit de ta substituabilité, bon courage. Dans un tel monde, si tu veux pouvoir extraire de telle propriétés, il faut que chaque objet te renseigne sur ses contrats lorsque tu communiques avec eux, et tu dois tirer parti de ces infos de manière dynamique. Laisse tomber l'enfer.

2.3. La leçon à retenir

L'intérêt de tout ça, c'est que l'OO façon Kay, *ça marche*. Internet en est une preuve. Et puisqu'on en parle, je me permets une deuxième citation d'Alan Kay, parce que celle-là est vraiment bien :

> The Internet was done so well that most people think of it as a natural resource like the Pacific Ocean, rather than something that was man-made. When was the last time a technology with a scale like that was so error-free ? The Web, in comparison, is a joke. The Web was done by amateurs.
>
> *Alan Kay*

Une belle phrase, des grandes vérités générales, et un tacle à la carotide les deux pieds décollés (et totalement gratuit, comme le fait remarquer Ksass'Peuk) sur les développeurs ; du Alan Kay dans le texte, en somme.

note KP

C'est d'autant plus un tacle injuste en fait. La première raison c'est que les gens qui ont fait le web c'est franchement pas des amateurs (du tout), et que les liens que représente effectivement internet ne sont constitués d'aucune intelligence (dit autrement : l'intelligence

n'est pas dans les messages transmis, elle est locale). Et la deuxième raison qui est de loin la plus importante et qu'internet repose sur une stack logicielle qui est tout sauf OO et en l'occurrence, c'est elle qui fournit l'intelligence actuellement et que ça c'est clairement le travail de dizaines (pour pas dire centaines) de milliers d'expert à travers le monde.

Et bien qu'il soit difficile (et pas forcément souhaitable) d'appliquer à la lettre ces idées dans un code C++, on peut s'inspirer de l'esprit de tout ça et essayer de concevoir nos programmes en ayant ces enjeux en tête ; cela ne pourra qu'améliorer la qualité de notre code.

Cela rassure aussi quant à la légitimité d'idées comme l'importance de l'abstraction ou le DIP[5] par exemple, et nous en donne une origine. Cela montre que ces principes permettent de s'approcher d'une conception qui marche superbement bien, et donc qu'ils ne sont pas absurdes.

Pour ma part, ces lectures m'ont beaucoup marquées, et m'ont permis de prendre beaucoup de recul sur l'OO.

Encore une note de Ksass'Peuk pour ceux qui veulent aller plus loin :

note KP

La définition que donne Alan Kay de l'OO est trop générique et en plus il ne détaille pas grand chose. C'est pour ça à mon sens qu'il vaut mieux préférer celle énoncée par William Cook. Il a même fait un post de blog qui allège le discours de la publi originale : http ://wcook.blogspot.com/2012/07/proposal-for-simplified-modern.html

2.4. Relativisons

Bien sûr, les systèmes OO d'Alan Kay ne sont pas la panacée, même si lui a l'air de le penser. Je vais encore laisser une note de Ksass'Peuk qui explique certains problèmes :

note KP

Internet **n'**en est **pas** une preuve. Internet est une preuve qu'un système peut marcher par échange de message à très grande échelle. Mais il faut pas oublier qu'internet, c'est aussi :

— coûteux en énergie
— douteux en terme de sécurité

Et que les objets qui le composent ne sont pas tous OO et certainement pas jusqu'à la base. Bref : ils ne sont pas eux mêmes composés d'objets. Du coup est ce qu'il faudrait refaire nos systèmes localement sur une approche objet ? Je doute qu'on puisse l'affirmer si facilement, il y a un moment où ça devient inefficace. Pour reprendre l'exemple plus tôt :

```
1 c = a.add(b)
```

Comment est ce que `a` doit implémenter l'opération d'addition ? Il n'a pas le droit d'avoir d'autres informations à propos de `b` que l'interface des entiers. Quelle interface dois-je

> définir pour les entiers ? Comment définir cette interface pour avoir un calcul de ma somme raisonnablement rapide et qui ne contraignent pas ma liberté de représentation interne ? On a au moins de une borne inf mais ce n'est pas la borne inf maximale, il faudrait remonter pour savoir ce qu'elle est, et pour l'instant je ne pense pas que qui que ce soit, surtout pas Kay n'a la réponse à cette question. Imagine si dans un système de bas niveau, chaque composant devait demander à son voisin s'il peut dialoguer avec lui à chaque opération (ben oui puisqu'il peut changer d'avis). Ce serait une perte phénoménale de performances et d'énergie. Qui voudrait d'un tel système ?

Et je suis tout à fait d'accord avec lui. On ne veut clairement pas d'un tel système purement OO d'Alan Kay. Mais réfléchir à tout ça fut pour moi une expérience très enrichissante, et c'est ça que je veux transmettre.

2.5. Bonus

Voici une petite conférence bien sympathique sur le sujet :

> Une vidéo de Jim O Coplien (accu 2016) où il parle d'OO, et au passage de Kay : https ://www.youtube.com/watch ?v=lQQ_CahFVzw
>
> *lmghs*

C'est à nouveau @lmghs qui a donné ce lien, merci à lui.

3. Conclusion

C'est tout pour cet épisode, j'attends vos commentaires avec impatience ! Surtout en ce qui concerne la section sur l'OO.

A lundi prochain ! <3

2. Au point où l'on peut demander la résolution d'un appel sur un objet dont on ne sait même pas encore s'il existe et s'il est capable de dialoguer avec nous.
3. Sinon ça serait légèrement contraignant.
4. N'y voyez aucun manque de respect envers Kay, son génie ou son travail. Mais faut avouer qu'il a quand-même des avis très tranchés sur à peu près tout ce dont il parle.
5. Et même, plus indirectement, de certains autres principes SOLID comme l'OCP, bien que je n'en aie pas parlé.