Anthony **Cyrille**

Redécouvrir la

PROGRAMMATION **ORIENTÉ OBJET**

ancyracademy

Redécouvrir la POO

Un livre qui vous fait voir la programmation orienté-objet sous un autre oeil.

Anthony Cyrille

Ce livre est disponible sur https://leanpub.com/redecouvrir-la-poo

Cette version a été publiée le 2025-09-09

Ceci est un livre Leanpub. Leanpub permet aux auteurs et éditeurs d'utiliser le processus de Lean Publishing. Le Lean Publishing consiste à publier un livre numérique en cours d'élaboration en utilisant des outils légers et de nombreuses itérations pour obtenir les retours des lecteurs, pivoter jusqu'à obtenir le bon livre et développer sa diffusion une fois celui-ci achevé.

Table des matières

Redécouvrir la POO

Introduction

Depuis quand existent les programmeurs ?

Environ… 70 ans ? Un peu plus ?

Une chose est sûre : comparée aux autres industries scientifiques, celle de l'ingénierie logicielle est très jeune. En fait, le terme même "d'ingénierie logicielle" n'est apparue qu'en 1968 lors de la fameuse conférence de l'OTAN de cette année.

La programmation orienté-objet est encore plus jeune que ça.

Elle n'a que 50 ans.

Des centaines, des milliers de livres ont été écrit sur la programmation orienté objet. Et pourtant, l'essentiel de ce qu'était l'orienté-objet s'est perdu avec le temps, si bien qu'aujourd'hui, très peu de développeurs savent réellement ce qu'est la POO.

C'est dommage car ils passent à côté de toute la richesse que peut leur apporter une approche objet.

A travers ce petit livre, j'espère vous faire redécouvrir la POO et vous donner un goût nouveau pour les objets.

L'essence de la POO

L'essence de la POO se manifeste à travers deux concepts extrêmement importants : **l'abstraction et le polymorphisme**.

Abstraction

Imaginez. Vous êtes fin cuisinier, et votre meilleur ami vous demande de l'aider à cuisiner un boeuf bourguinon pour le repas de ce soir. Il tient absolument à ce que le plat soit parfait.

La première étape est de découper le boeuf. Vous lui dites alors :

Découpe le boeuf en cubes

Il vous regarde, interloqué et vous dit.

Je n'ai jamais cuisiné de ma vie. Comment dois-je faire ?

Vous souriez gentiment avant de lui dire.

Prend ton morceau de boeuf et découpe le dans le sens de la longueur, puis découpe chaque morceau en cubes.

Visiblement gêné, il vous dit.

Je n'ai jamais utilisé de couteau…

Patient que vous êtes, vous lui donner des indications plus détaillées.

Place ton pouce contre le manche du couteau et ton index sur le dessus. Penche ton poignet et place la lame au dessus du morceau de boeuf, puis descend ton poignet pour enfoncer la lame dans le morceau et le découper.

Vous l'aidez à placer le couteau correctement dans sa main et à faire sa première découpe. Vous comprenez qu'il aura besoin de beaucoup d'aide pour venir à bout de la recette, mais il y arrivera.

Maintenant, imaginez la même situation avec un cuisinier aguerri. Cette fois-ci, nul besoin de détailler la découpe, vous lui dites simplement de découper le boeuf en cube et d'hacher les oignons, et il s'en sort très bien.

Pour finir, imaginez que vous vous rendiez dans un restaurant 5 étoiles à Paris. Le serveur vous apporte la carte, mais vous savez déjà très bien ce que vous voulez.

Un boeuf bourguignon.

Vous n'avez pas besoin de lui expliquer comment faire. Vous lui dites simplement ce que vous voulez obtenir.

Repensez maintenant à ces trois interactions. **Laquelle est la plus agréable ?**

Dans la première, vous êtes plongé dans les détails les plus profonds de la découpe car c'est le niveau de détail dont votre ami a besoin. C'est pénible, et ça vous demande beaucoup de concentration car vous devez penser à tous ces petits détails et à anticiper toutes les catastrophes possibles. Pour peu qu'il se coupe le doigt, ce sera votre faute !

Dans la seconde, vous êtes un niveau au dessus. Vous dites simplement à votre collègue cuisinier quelles actions faire dans les grandes lignes sachant pertinemment qu'il ne se coupera pas le doigt en prenant mal le couteau. Vous déléguez ces détails et restez concentré sur la recette.

Dans la dernière, vous n'avez même pas besoin de savoir comment faire un boeuf bourguignon. Vous demandez seulement à en avoir un.

Ces 3 situations reflètent **3 niveaux d'abstraction différents**. Avec chaque niveau d'abstraction, on s'éloigne des détails et on raisonne à un niveau beaucoup plus haut. Et même mieux, on se détache du besoin de contrôler les moindres faits et gestes de notre interlocuteur. Il est assez grand pour ne pas se déchiqueter le bras.

Mais en plus, à mesure que le niveau d'abstraction augmente, le langage s'enrichit et permet de dire beaucoup plus de choses en beaucoup moins de mots. Le simple fait de dire "découpe le boeuf en morceaux" est en fait un raccourci qui dissimule la façon dont le boeuf est découpé. Mieux encore, **elle laisse la liberté au cuisinier de procéder comme il l'entend** (à la main, à la machine, et même à la cuillère s'il est vaillant) du moment qu'à la fin,**on obtient des morceaux de boeuf**.

Le fait d'élever le niveau d'abstraction laisse **davantage de liberté aux autres participants du système d'évoluer, et ce sans impacter la recette**. Peu importe la façon dont le boeuf est découpé, la recette reste la même.

Enfin, élever ce niveau d'abstraction **réduit la charge cognitive qui pèse sur le donneur d'ordre**, car cette charge n'est plus centralisée dans sa tête : elle est répartie entre les différents participants. Il peut se concentrer sur la recette, qui est "la tâche" avec un grand T, tandis que chaque autre participant peut s'occuper d'une partie de la tâche sans avoir à prendre en compte l'ensemble. Quand on travaille en groupe, certains ont pour tâche d'avoir une vue d'ensemble sans s'importuner des détails, d'autres d'être concentré sur les détails sans avoir besoin de connaître la vue d'ensemble.

Cette métaphore de la recette parait anodine, **mais elle est centrale à la compréhension du véritable paradigme objet**.

Polymorphisme

Imaginez que vous ayez besoin d'aller à l'autre bout de la ville pour vous rendre chez le docteur, sauf que.. **vous n'êtes pas véhiculé**.

Du coup, vous imaginez quelques solutions possibles :

- **Appeler votre collègue Tony**. Il est sympa et a toujours un truc intéressant à raconter, mais il est pas fichu de se taire plus de 30 secondes.

- **Vous pouvez prendre le bus**. Mais aucun bus ne va directement à destination, donc il faudra changer de bus et attendre à chaque arrêt de bus.
- **Vous pouvez commander un taxi**. Il vous emmenera directement à destination sans trop jacasser, mais il risque de coûter beaucoup plus cher.

Quel est **le point commun** à ces trois situations ? **Le fait d'arriver à l'autre bout de la ville**.

Quel est **le point divergent** entre chacune de ces situations ? **La façon dont vous y arrivez et avantages/inconvénients de chacun**.

L'essence du polymorphisme, c'est exactement ça : **avoir plusieurs moyens d'aboutir à un résultat**. Du moins, c'est le début de l'idée. Le polymorphisme est encore plus puissant que ça.

Imaginez maintenant que vous soyez, en ce moment même, en train de planifier votre journée. Aujourd'hui, vous allez passer une visite médicale. Vous avez donc besoin de préparer vos papiers, choisir une tenue adaptée, puis vous rendre chez les docteur. Après ça, vous irez probablement boire un café, faire du lèche vitrine puis rentrer chez vous.

A cet instant précis, vous planifiez votre journée dans les grandes lignes. *Vous n'avez pas besoin encore de réfléchir à la façon dont vous allez vous rendre chez le médecin.* Vous gardez simplement en tête l'idée générale, l'abstraction "se rendre chez le médecin". Le détail de la façon dont vous vous y rendrez peut attendre un peu plus tard.

A cet instant là, vous ignorez "qui" va vous emmener chez le médecin : Tony, le chauffeur de bus, ou le chauffeur de Taxi ?

En vérité, ça n'a pas d'importance pour l'instant. **Vous pouvez choisir ce "qui" plus tard**, voir déléguer la tâche de déterminer ce "qui". Vous pouvez par exemple demander à ChatGPT de faire une simulation pour choisir le "qui" qui sera le moins cher, ou celui qui vous sera le plus rapide.

C'est ça, la seconde essence du polymorphisme : **ignorer le "qui"**. Vous n'avez pas besoin de savoir qui vous rend le service **du moment qu'il vous rende le service de la façon dont vous l'attendez**. Et ce service, c'est précisément votre abstraction. Ici, votre abstraction est "vous rendre chez le docteur". Et plusieurs personnes peuvent vous y aider.

En appliquant le polymorphisme, vous gagnez un degré de liberté. Si vous vous rendez chez le médecin chaque semaine, vous pouvez varier la façon dont vous vous y rendez selon le besoin. Seulement, votre planification ne changera pas : vous aurez toujours une tâche "se rendre chez le médecin". **L'abstraction reste fixe, mais vous pouvez varier la façon dont elle est exécutée**.

La connaissance

Avez-vous remarqué que les deux notions que l'on vient de voir ont un énorme point commun ? Ces deux principes visent à **réduire le niveau de connaissance en un point de votre programme**.

L'abstraction contribue à réduire la connaissance en cachant les détails de "**comment**" quelque chose est fait et vous aide à vous concentrer sur le "quoi". Un chef d'entreprise n'a pas besoin de savoir comment chaque produit de son usine est construit du moment qu'il est conforme aux exigences.

Le polymorphisme, lui, réduit la connaissance en cachant les détails de "**qui**" fait cette chose. Vous n'avez pas besoin de savoir quelles personnes sont impliquées dans la construction du produit, du moment qu'il est conforme aux exigence. En d'autres termes, le polymorphisme nous dit que le plus important est d'avoir un objet qui soit capable de répondre à nos attentes indépendamment de son identité.

Quelque part, c'est effrayant. Ne pas savoir est très perturbant, surtout lorsqu'on est responsable d'une tâche. Vous avez certainement connu un chef ou un manager ingérant, systématiquement sur votre dos par peur que vous fassiez mal votre travail. Et vous savez à quel point c'est désagréable.

Pourtant, c'est de cette façon dont codent la plupart des développeurs. **Ils ont besoin de savoir ce qu'il se passe**, sans quoi ils sont perdus et stressés. Le code doit leur apparaître devant leur yeux. Le code, rien que le code, et seulement le code.

Mais peut-être que parfois, **l'ignorance est une vertu**.

Reprenons l'exemple du chef d'entreprise. Il a fixé ses exigences et a des moyens de vérifier la conformité des produits créés. Mais derrière, **il laisse carte blanche à l'organisation de son entreprise pour effectuer le travail**. D'ailleurs, il n'a même plus besoin de se soucier de la production. Il peut se concentrer sur ce qu'il fait de mieux : cocher des cases sur Excel.

L'entreprise peut passer de 10 à 50 employés, certains peuvent venir et s'en aller, le processus de fabrication peut se complexifier et s'industrialiser… **autant de choses qui n'auront pas grand impact sur la façon de travailler du chef d'entreprise**.

Et c'est précisément ce que déverouille cette absence de connaissance : **l'indépendance**. Différentes parties du systèmes peuvent évoluer à loisir sans impacter les autres parties du système. Les architectes logiciel l'ont si bien compris que de cet état d'esprit sont nés **les microservices et les architectures distribuées**.

Mais tout ça vient de l'idée originelle des objets : **l'absence de connaissance**. Elle se résume en une phrase.

> Je n'ai pas besoin de savoir qui tu es, ni de savoir comment tu vas faire ce que je te demande. J'ai simplement besoin que tu fasses ce que je te demande.

Avec ça, vous êtes en mesure de comprendre une seconde vérité très profonde au sujet des objets : **leur identité a très peu d'importance**. Un objet n'est qu'une entité qui joue un rôle, et **c'est ce rôle qui est le centre d'attention de la véritable POO**. Ou, plus précisément, le rôle et la communication qui se fait entre les différents rôles de notre système.

Dans le cadre de la visite chez le docteur, plusieurs personnes peuvent nous emmener à son cabinet. Chaque personne joue le rôle de "chauffeur vers le cabinet", mais chacune de ces personnes pourrait jouer bien d'autres rôles auprès d'autres personnes. L'identité de votre chauffeur n'a aucune importance. Ce qui compte, c'est le rôle de chauffeur en lui-même et le fait qu'il puisse vous emmener à votre destination.

Pour résumer, l'essence de la véritable POO sont **les rôles et les messages**. Et, pour ce faire, elle exploite deux principes fondamentaux en ingénierie logicielle : **l'abstraction et le polymorphisme**.

La métaphore peut aller loin, mais rien ne vaut un peu de code.

La POO à l'oeuvre

Votre supérieur a besoin de vous pour écrire un script vraiment banal : lire un fichier et en afficher le texte.

Rien de bien compliqué ! Il suffit de quelques lignes de JavaScript.

```
import fs from 'fs';

async function main(
  filePath : string,
) {
  return fs.readFileSync(filePath, "utf-8");
}
```

Vous livrez le code, il fonctionne parfaitement bien.

Le lendemain, il revient vous voir avec d'autres instructions.

> Ce serait bien de pouvoir également sortir le texte dans un JSON pour pouvoir le manipuler avec d'autres programmes et l'envoyer sur le web.

Tellement simple ! Vous implémentez la solution en 5 minutes.

```
import fs from 'fs';

async function main(
  filePath : string,
  asJson : boolean
) {
  let text = fs.readFileSync(filePath, "utf-8");
  if (asJson) {
    return JSON.stringify({ text });
  } else {
    return text;
  }
}
```

Incroyable, une bonne journée de travail s'achève.

Le jour suivant, votre supérieur a de nouveaux besoins.

> Ce serait bien de pouvoir également lire des fichiers directement depuis Github. On aura besoin de lire certains fichiers de code.

Vous réflechissez un peu et arrivez à la solution suivante.

```
import fs from 'fs';

async function main(
  textPath : string | null,
  githubRepository : string | null,
  githubBranch : string | null,
  githubPath : string | null,
  asJson : boolean
) {
  let text = "";
  if (textPath !== null) {
    text = fs.readFileSync(textPath, "utf-8");
  } else {
    const fullPath = `https ://raw.githubusercontent.com/${githubRepository}/${githubBranch}/${githubPath}`;
    text = await fetch(fullPath).then((e) => e.text());
  }

  if (asJson) {
    return JSON.stringify({ text });
  } else {
    return text;
  }
}
```

Un étrange sentiment de malaise commence à vous habiter, et vous sentez que ce programme n'a pas fini d'évoluer. Et votre intuition ne vous a pas trompé, car votre supérieur a de nouvelles features à ajouter à ce programme.

> En fait, j'aimerais pouvoir sélectionner certaines lignes plutôt que tout le fichier. Et puis, j'aimerais avoir la possibilité de supprimer l'indentation et d'avoir tout le texte bien aligné à gauche. Tu peux me faire ça pour hier ?

La mise à jour prend un peu plus de temps, mais vous y arrivez.

```
async function main(
  textPath : string | null,
  githubRepository : string | null,
  githubBranch : string | null,
  githubPath : string | null,
  lines : string | null,
  removeIndentation : boolean,
  asJson : boolean,
) {
  let text = "" ;
  if (textPath !== null) {
    text = fs.readFileSync(textPath, "utf-8") ;
  } else {
    const fullPath = `https ://raw.githubusercontent.com/${githubRepository}/${githubBranch}/${githubPath}` ;
    text = await fetch(fullPath).then((e) => e.text()) ;
  }

  let textLines = text.split("\n") ;

  if (lines) {
    const parts = lines.split(",") ;
    const selectedLines : string[] = [] ;
    for (const part of parts) {
      if (part.includes("-")) {
        const [start, end] = part.split("-").map(Number) ;
        for (let i = start ; i <= end ; i++) {
          if (i - 1 < textLines.length) {
            selectedLines.push(textLines[i - 1]) ;
          }
        }
      } else {
        const lineNum = Number(part) ;
        if (lineNum - 1 < textLines.length) {
          selectedLines.push(textLines[lineNum - 1]) ;
        }
      }
```

```
    }

    textLines = selectedLines;
  }

  if (removeIndentation) {
    textLines = textLines.map((line) => line.trimStart());
  }

  if (asJson) {
    return JSON.stringify({
      text: textLines.join("\n"),
    });
  } else {
    return textLines.join("\n");
  }
}
```

Vous regardez votre création avec un certain dégoût. Il faut absolument que nous nettoyez ça. Et il y a deux solutions qui se présentent à vous :

- Une approche procédurale et structurée
- Une approche orienté objet

L'approche procédurale et structurée

Que fait-on quand une fonction devient trop large ? **On la découpe en fonctions plus petites !**

Cette approche est souvent appelée "**Divide & Conquer**" et est un élément clé de la pensée dite **structurée**. La programmation structurée est un paradigme né des recherches de **Edsger Djikstra** qui cherchait à rendre les programmes **informatiques prouvable mathématiquement**. Or, lorsque le flux d'exécution du programme n'est pas prévisible, l'analyse devient impossible. C'est ce qui a donné au fameux article intitulé "**GOTO considered harmful**".

L'approche structurée consiste donc à décomposer ses applications avec 3 "building blocks" :

- La **séquence d'opérations**, une instruction après l'autre
- La **sélection** avec les conditions
- **L'itération** avec les boucles ou la récursivité

Aujourd'hui, l'essentiel de nos programmes sont composés de ces trois composants. Mais ces composants ne nous disent pas grand chose sur **la façon d'organiser le code et le programme**. Savoir tenir un stylo ne suffit pas à devenir un écrivain.

La méthode structurée, appelée "**Structured Analysis and Design**" (SAD, SADT ou SA/SD), consiste à décomposer un programme large en sous-fonctions. Lorsqu'une procédure devient trop large, on en extrait des sous-procédures pour la rendre plus simple à comprendre.

J'emploie procédure et fonction de manière interchangeable. Dans notre propos, il s'agit exactement de la même chose, mais dans la litterature informatique, les deux ont des définitions légèrement différentes.

Dans notre cas, on pourrait simplement décomposer notre fonctions de la façon suivante.

```
function readFromFile(textPath : string) {
  return fs.readFileSync(textPath, "utf-8");
}

async function readFromGithub(githubRepository : string, githubBranch : string,
↪  githubPath : string) {
  const fullPath = `https ://raw.githubusercontent.com/${githubRepository}/${
  ↪  githubBranch}/${githubPath}`;
  return await fetch(fullPath).then((e) => e.text());
}

function readSelectedLines(lines : string[], spec : string) {
  const parts = spec.split(",");
  const selectedLines : string[] = [];

  for (const part of parts) {
    if (part.includes("-")) {
      const [start, end] = part.split("-").map(Number);
      for (let i = start; i <= end; i++) {
        if (i - 1 < lines.length) {
          selectedLines.push(lines[i - 1]);
        }
      }
    } else {
      const lineNum = Number(part);
      if (lineNum - 1 < lines.length) {
        selectedLines.push(lines[lineNum - 1]);
      }
    }
  }

```

```
30    return selectedLines;
31  }
32
33  function doRemoveIndentation(lines : string[]) {
34    return lines.map(line => line.trimStart());
35  }
36
37  function convertToJson(text : string) {
38    return JSON.stringify({
39      text,
40    });
41  }
42
43  async function main(
44    textPath : string | null,
45    githubRepository : string | null,
46    githubBranch : string | null,
47    githubPath : string | null,
48    lines : string | null,
49    removeIndentation : boolean,
50    asJson : boolean,
51  ) {
52    let text = "";
53    if (textPath !== null) {
54      text = readFromFile(textPath);
55    } else {
56      text = await readFromGithub(githubRepository, githubBranch, githubPath);
57    }
58
59    let textLines = text.split("\n");
60
61    if (lines) {
62      textLines = readSelectedLines(textLines, lines);
63    }
64
65    if (removeIndentation) {
66      textLines = doRemoveIndentation();
67    }
68
69    if (asJson) {
70      return convertToJson(textLines.join("\n"));
71    } else {
72      return textLines.join("\n");
73    }
74  }
```

En analysant objectivement cette implémentation, elle offre quand même plusieurs avantages comparé à la précédente :

- Le niveau d'abstraction est plus élevé, des méthodes comme `readFromFile` et

`readFromGithub` sont plus simples à comprendre
- Il y a moins de ligne de code dans la fonction

C'est un très bon premier pas. Mais il y a deux aspects de cette méthode qui, s'ils ne sont pas adressés, vont rendre le programme beaucoup plus difficile à comprendre et à maintenir dans le temps.

La complexité cyclomatique

D'abord, le premier point : a quel point cette fonction est complexe ?

Une des mesures de la complexité d'un programme et sa **complexité cyclomatique**. En d'autres termes, c'est le nombre de chemins que l'on peut suivre lors de l'exécution de la fonction.

Si notre découpage n'a pas du tout fait évoluer cette métrique, elle permet au moins de la mesurer beaucoup plus facilement.

Prenons un exemple simple.

```
async function main(
  textPath : string | null,
  githubRepository : string | null,
  githubBranch : string | null,
  githubPath : string | null,
) {
  let text = "" ;
  if (textPath  !== null) {
    text = readFromFile(textPath) ;
  } else {
    text = await readFromGithub(githubRepository, githubBranch, githubPath) ;
  }
}
```

Combien de chemin d'exécution peut-on emprunter ?

- Le premier, c'est lorsque `textPath !== null`
- Le second et dernier, à l'intérieur du `else`

Soit deux chemins.

Rajoutons un peu de code.

```
async function main(
  textPath : string | null,
  githubRepository : string | null,
  githubBranch : string | null,
  githubPath : string | null,
  lines : string | null,
  removeIndentation : boolean,
  asJson : boolean,
) {
  let text = "" ;
  if (textPath  !== null) {
    text = readFromFile(textPath) ;
  } else {
    text = await readFromGithub(githubRepository, githubBranch, githubPath) ;
  }

  let textLines = text.split("\n") ;

  if (lines) {
    textLines = readSelectedLines(textLines, lines) ;
  }
}
```

Cette fois-ci, on a toujours les deux chemin possibles du départ, mais pour chaque chemin, on a deux possibilités :

- Soit `lines` n'est pas égal à `null` et on peut entrer dans la condition
- Soit `lines` est `null` et la condition est évitée

De 2 chemins, on est passé à 4.

Avec le programme complet, on obtient une **complexité cyclomatique de 16**.

Que nous dit cette complexité ? Que pour comprendre pleinement le programme, il me faudra potentiellement comprendre les 16 chemin d'exécution possible de cette fonction. Sur des programmes plus complexes avec des conditions imbriqués, c'est encore pire.

Et il n'existe malheureusement aucun outil dans la boite de la programmation structurée pour dénouer ce problème. Décomposer en sous-fonctions aura même tendance à obscurcir le programme sans réduire la complexité de l'ensemble des fonctions.

Et ce n'est pas le seul problème.

L'extensibilité

Avec votre expérience du développement logiciel, vous savez déjà que votre supérieur va revenir avec tout un tas d'autres features, par exemple :

- Charger un fichier texte depuis un serveur FTP
- Ou le charger depuis un serveur de stockage comme S3 ou Google Drive
- De sortir le résultat au format XML
- Ou en HTML
- De pouvoir lire un fichier depuis une archive

Notre belle fonction qui contenait 5 lignes de code aura tôt fait d'en contenir 1 000 et de prendre en complexité à mesure que le temps passe et que les features s'empilent.

Des bugs obscurs et très difficile à comprendre et à supprimer vont surgir de la masse de code qui, peu à peu, s'effondre sous son propre poids.

Vous avez déjà travaillé sur une classe ou une fonction similaire, et vous savez à quel point c'est déprimant.

L'approche orienté objet

Il y a, dans le livre "Object Thinking" de David West, une citation absolument mémorable.

> Le premier pas à faire pour passer d'un développeur procédural à un développeur objet, c'est la lobotomie.

Ce qu'il veut dire, c'est que penser en objet revient souvent à penser d'une façon très différente, presque contre-intuitive. Il s'agit de ne plus avoir une vue centralisée et impérative, mais de voir notre programme de façon distribuée et déclarative. Notre programme est une société dans laquelle vivent des objets, et la communication entre objets est égalitaire : un objet n'a pas d'ascendant sur un autre.

En d'autres termes, **la responsabilité de mener à bien une tâche est distribuée entre les objets**, et il n'y a pas de connaissance centralisée qui permette de savoir qui fait quoi à un instant T.

Bien que ça puisse paraître inquiétant, ça apporte un nombre considérable d'avantages… pour peu qu'on s'y habitue.

Reprenons notre programme et intéressons nous plus précisément aux premier bloc conditionnel.

```
async function main(
  textPath : string | null,
  githubRepository : string | null,
  githubBranch : string | null,
  githubPath : string | null,
) {
  let text = "" ;
  if (textPath !== null) {
    text = fs.readFileSync(textPath, "utf-8") ;
  } else {
    const fullPath = `https ://raw.githubusercontent.com/${githubRepository}/${githubBranch}/${githubPath}` ;
    text = await fetch(fullPath).then((e) => e.text()) ;
  }
}
```

L'approche procédural serait de la décomposer en deux fonctions : `readFromText` et `readFromGithub`. Le niveau d'abstraction s'est élevé, **mais il reste encore trop haut**.

Alors réflechissons un peu : quels autres cas de lectures peuvent surgir ici ? Comme on l'a vu plus haut, on pourrait nous demander de lire un fichier depuis S3, Dropbox, Google Drive, un NAS, un serveur FTP…

Toutes ces "options" possibles sont des **variations**. Oui, mais des variations de quoi ? Qu'est-ce qui *unit* ces variations ?

Le fait de lire du texte, depuis une source.

Cette analyse que l'on vient de faire est **vitale** en orienté-objet, si bien qu'elle devient rapidement une seconde nature pour le véritable développeur OO. On parle de **Commonality / Variablity Analysis**.

Il s'agit d'analyser quelles sont les variations possibles et de trouver le dénominateur commun. Ou parfois l'inverse, de trouver le dénominateur commun puis de lister les abstractions.

Le dénominateur commun, c'est justement notre **abstraction**. Vous vous souvenez, on en avait parlé un peu plus tôt : c'est ce qui permet à notre programme d'être plus simple à comprendre et de raisonner avec des concepts plus puissants.

Et les variations correspondent au **polymorphisme**. Toutes les variations sont une des options possibles de cette abstraction.

Dans l'exemple du rendez-vous chez le médecin, la notion de "Chauffeur" correspond à notre **Commonality** tandis que les différents chauffeurs possibles (l'ami, le bus ou le taxi) sont des **Variability**.

Avec ces armes en main, on peut tenter une nouvelle approche et refactoriser notre programme.

```
interface Source {
  read() : Promise<string> ;
}

class TextFileSource implements Source {
  constructor(private readonly textPath : string) {}

  async read() {
    return fs.promises.readFileSync(this.textPath, "utf-8") ;
  }
}

class GithubSource implements Source {
  constructor(
    private readonly repository : string,
    private readonly branch : string,
    private readonly path : string
  ) {}

  async read() {
    const fullPath = `https ://raw.githubusercontent.com/${this.repository}/${this.
    ↪  branch}/${this.path}` ;
    return fetch(fullPath).then((e) => e.text()) ;
  }
}

async function main(
  textPath : string | null,
  githubRepository : string | null,
  githubBranch : string | null,
  githubPath : string | null,
) {
  let source : Source ;
  if (textPath !== null) {
    source = new TextFileSource(textPath) ;
  } else {
    source = new GithubSource(githubRepository, githubBranch, githubPath)
  }

  let text = await source.read() ;
}
```

On a créé une interface qui correspond à notre abstraction : une source depuis laquelle lire du texte.

Puis on a créé deux classes qui correspondent à nos variations : `TextFileSource` et `GithubSource`. Les noms sont explicites.

Enfin, on a instantié ces sources à l'intérieur de notre fonction `main`.

Seulement, a t'on vraiment avancé ? La complexité cyclomatique est restée la même, on a rajouté beaucoup plus de code et enfin, on a absolument rien gagné en terme d'extensibilité !

Effectivement, le CVA ne suffit pas à s'approcher de l'objet. Mais c'est un excellent premier pas. Il nous manque une deuxième clé qui, elle aussi, est vitale au paradigme objet.

Open / Closed Principle

Et s'il était possible de pouvoir étendre les fonctionnalités d'un système (à savoir une classe, un module ou une application entière) sans devoir toucher au code de ce système ?

Si vous utilisez un IDE, vous avez déjà profité de cette extensibilité. Votre IDE est un noyau riche qui peut-être davantage enrichi grâce aux **plugins**. Il n'y a pas besoin de modifier le code de votre IDE pour pouvoir supporter un nouveau langage ou automatiser le lancement de tests par exemple. On peut modifier et étendre le programme de l'extérieur.

C'est justement ça, l'Open / Closed Principle. L'IDE est fermé à la modification mais ouvert à l'extension.

Que fait notre programme, dans les grandes lignes ?

- Il lit du texte depuis une source de texte
- Le formatte et sélectionne uniquement les lignes désirées
- Puis le sort dans un format configurable, tel que JSON, Text ou XML

On peut constater que peu importe la source employée, **notre algorithme reste inchangé**. Notre algorithme ne dépend pas d'une source spécifique mais d'une source quelconque. Et si on essayait de trouver le moyen de refléter ça dans le code ?

Pour ça, il faut employer un autre principe fondamental de la programmation orienté-objet : **l'inversion de dépendances**.

Il s'agit simplement de rendre cette dépendance injectable. Et par dépendance, j'entend l'abstraction, la notion de Source.

```
async function main(
  source : Source
) {
  let text = await source.read();
}
```

D'un coup de baguette magique, on a :

- **Simplifié la signature de la fonction**, car là où elle avait beaucoup de paramètres (dont cerains optionnels), elle n'en a plus qu'un
- **Simplifié le code**, car la sélection a été sortie de l'algorithme, ce qui rend le code plus simple à comprendre et à testé
- **Divisé la complexité cyclomatique par 2**
- **Elevé le niveau d'abstraction de la fonction**, car on ignore tout des variations de source possible puisque ça n'a aucun impact sur l'algorithme

Beaucoup de bienfaits gagnés grâce à une approche orienté-objet.

Mais il y a une critique pertinente à faire.

> *Oui mais il faudra quand même faire une sélection quelque part dans le code pour utiliser la bonne variation. Donc au final, tu n'as fait que déplacer le problème !*

En un sens oui, mais ça dépend de la façon dont la fonction est utilisée. Si l'appel de la fonction est codée manuellement, alors la selection se fait **au moment de la compilation**, auquel cas la condition a tout bonnement disparue.

```
const result = await main(
  new TextFileSource("my-text.txt")
)
```

Par contre, dans le cas où la sélection se fait à l'exécution (parce qu'elle dépend d'une configuration en base de données ou qu'elle dépend d'une entrée utilisateur) alors on a tout de même gagné un bénéfice : **on a remonté au plus haut cette sélection avant qu'elle ne se propage dans le coeur du programme**.

Certains parlent de "lift conditionals up".

En fait, il est courant d'avoir un type d'objet bien spécifique dont l'intérêt est de contenir toutes ces conditions : on les appelles des **Abstract Factory**.

Factories are where conditionals go to die. Sandi Metz

Ce sont des objets qui vivent généralement **en périphérie du programme**, là où tout est incertain. Ils reçoivent de l'information venant de l'extérieur (requête HTTP, commande CLI, message d'un broker) et convertissent cette information en objets prêts à l'emploi. C'est ici que survient la décision de créer l'objet.

C'est d'ailleurs une règle fondamentale de l'orienté-objet : **l'objet qui instancie un objet n'est pas celui qui l'utilise**.

Et c'est comme ça qu'on programme en orienté-objet : on sélectionne les **candidats** (autre mot pour "variation") qui vont jouer des **rôles** (autre mot pour "abstraction") dans une **pièce** (autre mot pour "programme"). Une fois les candidats sélectionnés, il n'y a plus qu'à jouer la pièce.

D'ailleurs, on peut considérer qu'un serveur web est composé de mini-programmes et que chaque combo URL+Method correspond à un mini-programme. C'est d'ailleurs le principe derrière les CGI, mais.. je m'égare.

* * *

Reprenons notre programme et essayons d'appliquer notre outil d'analyse afin de le rendre plus extensible.

Commençons par chercher une abstraction. La plus simple se trouve à la fin du programme.

```
1  if (asJson) {
2    return JSON.stringify({
3      text : textLines.join("\n"),
4    });
5  } else {
6    return textLines.join("\n");
7  }
```

Quelle sont les variabilités ? Le fait de sortir la valeur en JSON ou au format texte.

Quelle est donc l'abstraction ? Le fait de sortir la valeur.

On peut donc imaginer créer l'abstraction de cette façon :

```ts
export interface OutputFormat {
  toText(text : string) : string ;
}

export class TextOutputFormat implements OutputFormat {
  toText(text : string) : string {
    return text ;
  }
}

export class JsonOutputFormat implements OutputFormat {
  toText(text : string) : string {
    return JSON.stringify({ text }) ;
  }
}
```

Et voilà qu'on a créé un second point de flexion (on parle également de **Seam**). On peut désormais utiliser n'importe quel format de sortie. Il suffit juste d'en créer une classe et de l'injecter.

* * *

Voyons maintenant l'abstraction suivante, qui est un peu plus subtile : la sélection de lignes.

```ts
if (lines) {
  const parts = lines.split(",") ;
  const selectedLines : string[] = [] ;
  for (const part of parts) {
    if (part.includes("-")) {
      const [start, end] = part.split("-").map(Number) ;
      for (let i = start ; i <= end ; i++) {
        if (i - 1 < textLines.length) {
          selectedLines.push(textLines[i - 1]) ;
        }
      }
    } else {
      const lineNum = Number(part) ;
      if (lineNum - 1 < textLines.length) {
        selectedLines.push(textLines[lineNum - 1]) ;
      }
    }
  }

  textLines = selectedLines ;
}
```

Quel sont les variations ici ? A priori, il n'y en a qu'une : celle de sélectionner des lignes spécifiques.

Sauf qu'en vérité, il y en a deux. L'autre est implicite, mais bien existante. Pour la voir, posez-vous la question : quelle est l'autre cas de figure ?

Si on ne sélectionne pas de lignes spécifiques, **on sélectionne toutes les lignes**.

Et oui, la seconde variation cachée et implicite est celle qui consiste à ne pas filtrer les lignes mais à toutes les prendre !

On peut maintenant créer une abstraction qui consiste à effectuer une sélection sur les lignes.

```
export interface LineSelector {
  selectLines(lines : string[]) : string[] ;
}

export class SpecificLinesSelector implements LineSelector {
  constructor(private readonly lineSpec : string) {}

  selectLines(lines : string[]) : string[] {
    const parts = this.lineSpec.split(",") ;
    const selectedLines : string[] = [] ;
    for (const part of parts) {
      if (part.includes("-")) {
        const [start, end] = part.split("-").map(Number) ;
        for (let i = start ; i <= end ; i++) {
          if (i - 1 < lines.length) {
            selectedLines.push(lines[i - 1]) ;
          }
        }
      } else {
        const lineNum = Number(part) ;
        if (lineNum - 1 < lines.length) {
          selectedLines.push(lines[lineNum - 1]) ;
        }
      }
    }
    return selectedLines ;
  }
}


export class AllLinesSelector implements LineSelector {
  selectLines(lines : string[]) : string[] {
    return lines ;
  }
}
```

Maintenant, on peut varier la façon dont les lignes sont sélectionnés. Super !

Ce type d'objet, `AllLinesSelector`, est également appelé un Null Object. Il s'agit d'un d'objet sans comportement spécifique, ou avec un comportement par défaut. On l'appelle ainsi car il permet de ne pas avoir à traiter les `null` dans notre programme. Toute une classe d'erreur qui disparait juste en codant proprement en objet !

Non seulement on se débarasse d'une condition, mais en plus, on créé un langage explicite et plus clair à comprendre.

* * *

On finit par avoir le code suivant.

```
if (removeIndentation) {
  textLines = textLines.map((line) => line.trimStart());
}
```

Vous avez probablement compris le principe maintenant. On a deux variations possibles :

- Celle de supprimer l'indentation
- Celle de la conserver

L'abstraction est donc le formattage de l'indentation.

```
export interface Formatter {
  format(lines : string[]) : string[];
}

export class IndentationFormatter implements Formatter {
  format(lines : string[]) : string[] {
    return lines.map((line) => line.trimStart());
  }
}

export class NoFormatting implements Formatter {
  format(lines : string[]) : string[] {
    return lines;
  }
}
```

Après l'application de ces abstractions, on obtient le programme suivant.

```
async function main(
  source : Source,
  lineSelector : LineSelector,
  formatter : Formatter,
  outputFormat : OutputFormat,
) {
  const sourceText = await source.readText() ;

  const lines = sourceText.split("\n") ;
  const selectedLines = lineSelector.selectLines(lines) ;
  const formattedLines = formatter.format(selectedLines) ;
  const finalText = formattedLines.join("\n") ;

  return outputFormat.toText(finalText) ;
}
```

Qui s'utilise de la façon suivante :

```
const runText = () =>
  main(
    new FilesystemSource(path.resolve("samples", "mobydick.txt")),
    new AllLinesSelector(),
    new NoFormatting(),
    new TextOutputFormat(),
  ) ;

const runGithub = () =>
  main(
    new GithubSource("facebook/react", "main", ".eslintrc.js"),
    new SpecificLinesSelector("10-12, 15, 17-20"),
    new IndentationFormatter(),
    new JsonOutputFormat(),
  ) ;
```

Voyez comment le code est extrêmement clair et documente le comportement attendu par la fonction. Un bon code OO est très déclaratif et ressemble beaucoup à de la programmation fonctionnel. En fait, si on s'en tient à la définition de l'orienté-objet qu'on a donné plus haut, vous pouvez parfaitement faire de l'OO… en fonctionnel !

```
const main = (
  readLines : SourceReader,
  selectLines : LineSelector,
  format : Formatter,
  output : OutputFormatter,
) =>
  readLines().then((text) =>
    pipe(splitLines, selectLines, format, joinLines, output)(text),
  );

const runText = () =>
  main(
    createFsReader(path.resolve("samples", "mobydick.txt")),
    allLinesSelector,
    noopFormatter,
    textOutputFormatter,
  );

const runGithub = () =>
  main(
    createGithubReader("facebook/react", "main", ".eslintrc.js"),
    specificLinesSelector("10-12, 15, 17-20"),
    indentRemovalFormatter,
    jsonOutputFormatter,
  );
```

L'emploi ou non de classes n'est **qu'un détail d'implémentation du paradigme objet**. Le plus important, c'est **la capacité à pouvoir créer des abstractions et de les rendre injectable**. Peu importe que vous utilisez des classes ou des fonctions.

Si vous voulez refaire l'exercice, vous pouvez partir de ce template. Vous trouverez des branches avec la solution class-based et function-based.

Conclusion

On a vu ensemble comment transformer un programme purement procédural vers une approche orienté-objet. Mais… **devriez-vous vraiment employer cette approche de façon systématique ?**

Je reconnais que j'ai un peu exagéré dans cet exemple, notamment avec la partie "suppression de l'indentation". Il y a très peu de chances qu'une 3e variation apparaisse un jour, donc on se retrouve qu'avec deux options : garder l'indentation ou non.

Approcher cet aspect de la fonction en OO n'était donc pas nécessaire. On aurait parfaitement pu garder un booléen ici, souffrir d'une légère augmentation de la complexité de la fonction, mais s'économiser une dizaine de lignes. Le fait de créer des interfaces et des classes représente elle aussi **une forme de complexité qu'il faut pouvoir justifier**.

L'exemple de la sélection de lignes est plus facile à justifier car il peut y avoir plusieurs algorithmes de sélections. Je peux vouloir sélectionner uniquement les lignes pairs, ou les 5 premières lignes de chaque chapitre. Le taux de variation augmente un peu plus, et le choix d'insérer ici une abstraction polymorphique me semble justifié.

Mais si vous écriviez, à la place, un programme qui ne connaitra jamais de variation ? Par exemple, dans le cas d'un script à lancer une seule fois, ou à n'utiliser que dans un cas spécifique. Dans ce contexte, quel intérêt peut avoir une approche objet ?

Je pense que vous commencez à comprendre ce que j'essaye de vous dire. L'approche orienté-objet se résume à **rendre notre code résilient face aux changements** en **anticipant les changements les plus probables**. C'est une approche de structuration de code qui vous permet de contrôler la complexité de votre code.

Prenons un exemple concret : **une application de gestion de fichiers**.

Nos clients avaient besoin de gérer leur espace de stockage via une interface simple : déposer des fichiers, créer des dossiers, déplacer des fichiers, etc. En bref, un genre de Google Drive.

Le petit atout, c'était qu'ils **utilisaient leur propre serveur de stockage**. Nous, on ne proposait qu'une interface graphique. Mais nos clients utilisaient différents services : S3, Google Drive, Dropbox, etc.

Il fallait donc créer un programme extensible qui permet de gérer un nouveau service très facilement.

On a donc procédé à une analyse telle que vous l'avez vu dans ce petit livre : déterminer les abstractions, le comportement attendu de chaque service, etc. Rien de différent de ce qu'on a vu plus haut, seulement dans un contexte beaucoup plus complexe.

L'approche orienté-objet nous a permis de scaler la gestion des services. Ajouter un nouveau service n'impliquait pas de modifier du code existant, mais de créer un nouveau package et de rajouter du code à l'intérieur. Idem pour la UI. La détection du package était automatique. **Cette facilité et cette sensation de flexibilité, tout développeur devrait la connaître**.

J'espère que ce modeste petit livre vous aura donné envie d'approfondir l'approche orienté-objet et toute la richesse que ce paradigme peut vous apporter.

Aller plus loin

Si voulez explorer plus en profondeur le design orienté objet et son jumeau, le Domain-Driven Design, vous pouvez me joindre sur LinkedIn ou sur ma plateforme AncyrAcademy. J'enseigne régulièrement ces sujets lors de Workshops, aussi bien en entreprise que pour les particuliers. J'accompagne les entreprises dans leur intégration du Domain-Driven Design en réalisant des audit de code, du pair-programming et des formations sur-mesures.

Envoyez moi un e-mail à contact@ancyracademy.fr.