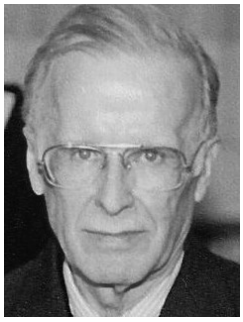
Chapitre 1

Méthodes de programmation

UN SCIENTIFIQUE



Après des études de chimie, puis de médecine, **John Backus** (1924-2007) devient technicien radio. Ce n'est que tardivement qu'il s'intéresse aux mathématiques puis entre à IBM en 1950.
En novembre 1954, il présente le langage Fortran puis participe à l'élaboration du langage de programmation ALGOL. Par la suite, il participe à l'élaboration de langage de programmation purement fonctionnels. Ses travaux sont récompensés par le prix Turing obtenu en 1977.

Un peu d'histoire

Lors du tout début de l'informatique, dans l'immédiat après-guerre, on travaillait directement en langage machine ce qui demandait une connaissance technique très spécifique.
Arrivé à IBM en 1950, John Backus est confronté à la difficulté de noter les nombres. Il élabore la syntaxe et le fonctionnement d'un langage intermédiaire entre la machine et le programmeur dont il allège la tâche en créant le Fortran.
De très nombreux langages suivirent très rapidement comme le COBOL en 1959. Dans les années 1970 apparaît la programmation structurée et les langages de programmation s'adaptent aux différents besoins qui apparaissent. Ils acquièrent alors de nouvelles potentialités grâce à la vitesse toujours plus grande des processeurs. Le développement d'Internet nécessite la création de nouveaux langages spécifiques.

Objectifs

- Comprendre les problématiques du développement logiciel :
  - algorithme, programme ;
  - paradigme de programmation ;
  - méthodologie de développement ;
- Comprendre la compilation et l'exécution d'un programme :
  - langage compilé ou interprété ;
  - messages d'erreur ;
  - tests ;
- Évaluer la performance d'un algorithme :
  - terminaison, correction ;
  - complexité algorithmique ;
- Comprendre la représentation numérique de l'information.

Cours

Développement logiciel

Un logiciel est un programme ou un ensemble de programmes qui peut être exécuté par un ordinateur pour réaliser des tâches automatiquement. Pour cela, un programme déroule une suite d'instructions qui considèrent en entrée des données et produisent en sortie d'autres données. Dans un premier temps, il est nécessaire de bien distinguer les termes *programme*, *algorithme* et *langage de programmation*.

Un **algorithme** est une méthode de résolution d'un problème. Il définit une suite d'instructions permettant d'aboutir à une solution au problème. Chaque instruction y est décrite de manière précise, non ambigüe, avec des données d'entrées et de sortie bien identifiées mais l'algorithme reste à un niveau abstrait, indépendant de son implémentation dans un langage de programmation. La notion d'algorithme pour résoudre des problèmes est d'ailleurs largement antérieure à l'apparition des premiers ordinateurs, le terme "algorithme" venant d'une latinisation du nom d'un mathématicien perse du IX^e siècle, Al-Khwârizmî, qui a recensé et classé des algorithmes de résolution de problèmes mathématiques.

Un **programme** est un ensemble d'instructions exprimées dans un langage interprétable par un ordinateur pour qu'il puisse les réaliser. Il peut contenir l'implémentation d'algorithmes dont la suite d'instructions abstraites est alors traduite dans le langage utilisé par le programme. L'**exécution** du programme consiste à demander à un ordinateur de réaliser la suite d'instructions qui le compose.

Un **langage de programmation** est une notation définissant un ensemble d'éléments et de règles syntaxiques pour écrire le code **source** d'un programme. Un langage de programmation peut être vu comme une étape intermédiaire entre le langage machine, très difficile à comprendre pour un programmeur humain, et une description abstraite d'algorithmes qu'un ordinateur ne peut pas interpréter. Le code source d'un programme va ensuite être traduit en langage machine lors d'une phase de compilation ou d'interprétation (voir plus loin) pour pouvoir être exécuté.

Deux langages de programmation sont utilisés dans cet ouvrage pour les codes sources des programmes présentés et pour les exercices. Il s'agit des langages C et OCaml. Pour la manipulation de bases de données abordée dans le chapitre 9, le langage SQL est utilisé. Les codes sources des programmes du livre sont téléchargeables à l'adresse :

`https://github.com/lvercouter/INFO_MP2I`

Paradigmes de programmation

Il existe de nombreux langages de programmation qui peuvent être classés par la manière dont ils abordent l'écriture du code source d'un programme et par les concepts qu'ils manipulent. Ces familles de langages s'appellent des **paradigmes de programmation**.

Le paradigme de la programmation **impérative** est le plus répandu. Il consiste en l'écriture d'instructions qui s'exécutent en séquence, c'est-à-dire les unes après les autres. L'exécution d'un programme se déroule en exécutant sa première instruction, puis quand son exécution est terminée

exécute la suivante et ainsi de suite jusqu'à la fin du programme. En plus de l'exécution séquentielle, la programmation impérative repose sur trois concepts principaux :

- l'opérateur d'**affectation** utiliser pour modifier le contenu d'une variable ;
- les **instructions conditionnelles** qui évaluent la valeur de vérité d'une condition et en fonction de cette évaluation exécute ou non un bloc d'instructions ;
- les **instructions itératives** qui répètent l'exécution d'un même bloc d'instructions jusqu'à ce qu'une condition soit atteinte.

Une sous-famille de la programmation impérative est le paradigme de la programmation **structurée**. Il reprend les concepts de la programmation impérative en y ajoutant une décomposition du code source en plusieurs sous-programmes. Cette décomposition permet l'écriture de blocs d'instructions plus petits, plus ciblés sur une tâche précise et améliore la lisibilité et la modularité du code. De nombreux langages de programmation appartiennent au paradigme de la programmation structurée, comme le C utilisé pour les exemples de cet ouvrage, le C++, Pascal, Ada, BASIC, Fortran...

Le paradigme de la programmation **fonctionnelle** préconise une approche déclarative dans l'écriture d'un programme. Elle repose sur l'écriture de fonctions mathématiques et aborde la résolution d'un problème comme une composition de fonctions. La programmation fonctionnelle n'admet pas l'opération d'affectation. Le passage de valeurs d'une fonction à une autre n'utilise donc pas de variables mais une imbrication de fonctions pour que le résultat produit par l'évaluation de certaines fonctions soit pris en entrée d'autres fonctions. Parmi les langages de programmation fonctionnelle, on retrouve, entre autres, le Lisp (premier langage de ce paradigme), Scheme, Haskell, Scala ou OCaml (utilisé dans les exemples de cet ouvrage).

Le paradigme de la programmation **orientée-objet** repose sur la décomposition d'un programme en plusieurs entités appelées *objets*. Chaque objet regroupe à la fois des variables qui lui sont locales et des méthodes de traitement. La définition d'objets favorise la modularité dans l'écriture d'un programme par des objets dont on va encourager la cohésion dans leur contenu pour qu'un objet soit focalisé sur un concept ou une entité et définisse les données qui le caractérise ainsi que les méthodes qui s'y applique. Parmi les langages de programmation orientée-objet on retrouve Smalltalk, Ada, C++, Java, Python...

Remarque : Un langage de programmation ne suit pas forcément exclusivement un paradigme de programmation. Certains langages fournissent des instructions appartenant à plusieurs paradigmes de programmation de façon à ce que le programmeur puisse choisir l'approche qui convient le mieux au problème traité.

Méthodologie de développement logiciel

Pour répondre à un besoin donné, il est possible d'écrire un programme ou un logiciel de nombreuses manières différentes mais toutes ne sont pas de même qualité. Ils peuvent différer par leur temps d'exécution, par le fait qu'ils marchent tout le temps ou provoquent parfois des erreurs, par la possibilité de ré-utiliser certaines parties pour d'autres tâches et ne pas avoir à ré-écrire complètement un autre programme... Bertrand Meyer [1] identifie un certain nombre de facteurs de qualité pour évaluer un logiciel. Parmi ceux-ci, les facteurs externes représentent les qualités d'*utilisation* du logiciel. Il s'agit principalement de :

- la **correction** qui est la faculté du logiciel à bien réaliser la tâche pour laquelle il est conçu ;

1. "Conception et programmation orientées objet", B. Meyer, eds. Eyrolles, 5ème tirage, 2011.

- la **robustesse** qui est la faculté du logiciel à réagir de manière appropriée en présence de conditions anormales ;
- l'**extensibilité** qui est la facilité de modification du logiciel pour de nouveaux besoins ;
- la **réutilisabilité** qui est la capacité de certains éléments logiciels à servir à la construction de logiciels différents ;
- la **compatibilité** qui est la facilité de combiner le logiciel avec d'autres ;
- l'**efficacité** qui est la capacité du logiciel à minimiser les ressources (temps, calcul, ...) nécessaire à son exécution ;
- la **portabilité** qui est la capacité du logiciel à être utilisé dans différents environnements matériels ou logiciels ;
- la **facilité d'utilisation** qui est la faculté du logiciel à être pris en main par ses utilisateurs.

Les facteurs *internes* représentent les qualités attendues pour les programmeurs. Certains facteurs comme la réutilisabilité ou la portabilité sont à la fois externes et internes car il est souhaitable pour un programmeur que l'effort produit pour écrire un programme puisse aussi contribuer à d'autres logiciels ou dans d'autres environnements. Deux autres principaux facteurs internes peuvent être évoqués ici :

- la **lisibilité** qui est la facilité de compréhension du code source ;
- la **modularité** qui est la pertinence de la décomposition du code source en entités relativement petites, ré-utilisables et à forte cohésion sémantique.

La prise en compte de tous ses facteurs nécessite une grande rigueur lors de l'écriture des programmes d'un logiciel. Le suivi d'une *méthodologie* de développement logiciel permet de traiter les problèmes séparément, par étapes, et de se focaliser sur la rigueur nécessaire à chacune d'entre elles. Il existe différentes méthodes de développement proposées dans le domaine du *génie logiciel*. L'étude de ces méthodes n'entre pas dans le programme de cet ouvrage mais nous pouvons lister ici les principales étapes que l'on retrouve fréquemment :

- la phase d'**analyse** lors de laquelle les besoins fonctionnels du logiciel sont recensés, avec le client ou l'utilisateur final du logiciel, pour identifier de manière précise et non ambigüe ce que doit réaliser le logiciel ;
- la phase de **conception** lors de laquelle l'architecture globale du logiciel est définie pour identifier différents modules, leurs relations, les structures de données à manipuler, ... Des langages de modélisation, tels que UML ou OMT, sont souvent utilisés lors de la conception globale du logiciel. Une conception détaillée peut aussi intervenir par la définition d'algorithmes ;
- la phase d'**implémentation** lors de laquelle le ou les programmeurs écrivent dans un langage de programmation le code source des programmes ;
- la phase de **tests** lors de laquelle le bon fonctionnement des programmes est testé, notamment sur leurs facteurs de correction et de robustesse ;
- la phase de **recette** lors de laquelle le logiciel est livré pour être utilisé.

Même dans la réalisation de programmes de taille assez modeste, il est important de procéder par ses étapes. Si les besoins sont mal exprimés ou mal compris, la réalisation ne correspondra pas aux vraies attentes. Si la phase de conception est omise, le programmeur se retrouve à définir l'architecture de son programme en même temps qu'il écrit avec de très fortes chances pour que celle-ci ne soit pas adaptée. Ces deux exemples illustrent le risque d'avoir besoin de revenir en arrière, d'avoir écrit des parties inutiles, abandonnées résultant en une perte de temps.

Lisibilité, utilisabilité

Parmi les facteurs internes de qualité d'un logiciel, la lisibilité a une place importante car elle va influer sur d'autres facteurs tels que l'extensibilité ou la réutilisabilité. La lisibilité correspond à la facilité à comprendre un programme. On peut distinguer deux niveaux : la compréhension d'un code source pour des programmeurs sur ce code et la compréhension de ce que fait un code exécutable et comment l'utiliser.

Dans le premier cas, il convient d'adopter de bonnes pratiques, dès l'écriture du code source des premiers programmes, qui facilite leur lisibilité. Une première pratique consiste à insérer des **commentaires** dans le code source. Des symboles, dépendant du langage de programmation, servent à définir des zones de commentaires qui ne seront pas pris en compte lors de la compilation ou de l'interprétation du code source. De cette manière, il est possible d'écrire un texte libre pour expliquer une partie un peu compliquée du code source. Il faut trouver un bon équilibre dans la quantité de commentaires et les consacrer aux parties de code non triviales. Avoir trop de commentaires alourdit inutilement le code source mais trop peu de commentaires peut rendre des parties de code difficiles à comprendre. Un piège à éviter est que les commentaires ne doivent pas paraphraser le code. Ils doivent aider à comprendre la nature des tâches réalisées. L'exemple ci-dessous est un commentaire qui paraphrase l'instruction qui suit (le code est en langage C où une ligne de commentaires est indiquée en commençant par `//`) :

```
// c prend comme valeur 2 * 3.14159 * r
c = 2 * 3.14159 * r;
```

Un meilleur exemple de commentaire pour cette ligne est le suivant qui explique pourquoi cette opération est réalisée :

```
// c est la circonférence du cercle de rayon r
c = 2 * 3.14159 * r;
```

Une deuxième bonne pratique est l'**indentation** du code. L'indentation consiste à insérer des tabulations ou des espaces en début de ligne pour que tout le code ne soit pas aligné à gauche mais que des blocs d'instructions soient décalés. Cela permet de visualiser très facilement la structure du code et la manière dont les blocs d'instructions vont être emboîtés dans d'autres instructions. L'exemple ci-dessous montre une fonction en C calculant le factoriel d'un nombre sans indentation.

```
1 int factoriel(int n) {
2 int r = 1;
3 for (int i = n; i > 0; i--)
4 r = r * i;
5 return r;
6 }
```

Une version indentée du même code est donnée ci-dessous.

```
1 int factoriel(int n) {
2   int r = 1;
3   for (int i = n; i > 0; i--)
```

```
4       r = r * i;
5     return r;
6   }
```

Dans ce second exemple, on voit l'intérêt des tabulations ajoutées en début de ligne pour comprendre en un coup d'oeil comment sont structurées les instructions. Les lignes 2 à 5 sont décalées par rapport aux première et dernière ligne pour montrer qu'elles sont dans la fonction `factoriel`. La ligne 4 est de nouveau décalée pour montrer que cette instruction est dans la boucle `for` de la ligne précédente. Par contre la ligne suivante est au même niveau que la boucle `for` pour montrer qu'elle est en dehors des instructions de cette boucle et s'exécutera après. Vous pourrez revenir sur ces codes après lecture du chapitre 2 pour comprendre leur fonctionnement. Dans la plupart des langages (à l'exception notable de Python), l'indentation n'a aucune influence sur la compilation ou l'interprétation du code source. Son rôle est uniquement de rendre le code source plus lisible. Cela veut dire aussi que rien ne vous empêche de faire une mauvaise indentation qui rendrait le code plus illisible !

Une dernière bonne pratique pour la lisibilité du code source est d'utiliser des **noms explicites** pour les identifiants de variables et de fonctions. Parfois le simple choix d'un nom précis permet tout de suite de comprendre ce que fait une fonction ou la nature d'une valeur stockée dans une variable. Dans l'exemple de code donné précédemment, le fait de nommer la fonction `factoriel` permet de comprendre intuitivement ce qu'elle fait. Si cette fonction avait été simplement appelée `f` cela aurait été plus obscur.

La lisibilité d'un programme s'évalue aussi du point de vue de son utilisation. Dans ce cas, on parle de la **documentation** d'un programme. La documentation ne va pas expliquer les instructions du code source mais plutôt comment utiliser ce qu'il fournit d'un point de vue extérieur. Si le programme est un module destiné à être utilisé par d'autres programmes, la documentation explique le contenu du module et, pour les fonctions, ce qu'elles font, quelles sont leurs entrées et quelles sont leurs sorties. Une documentation est aussi nécessaire pour des programmes exécutables pour expliquer comment ils doivent être installés, configurés puis utilisés.

■ Compilation vs interprétation

Le code source d'un programme ne peut pas être directement exécuté par un ordinateur. Il doit être traduit dans le langage de la machine. Selon le langage de programmation utilisé, cette traduction se fait de deux manières différentes, par compilation ou par interprétation du code source.

Langage compilé

Les langages compilés utilisent un programme appelé **compilateur**. Il prend en entrée un fichier contenant un code source et produit en résultat un fichier appelé *objet* contenant les instructions du programme en langage machine, c'est-à-dire qu'elles peuvent être exécutées par l'ordinateur et le système d'exploitation pour lesquels elles ont été compilées. La création d'un programme exécutable est souvent réalisée dans une deuxième étape appelée **édition de liens** par l'assemblage de plusieurs fichiers objets. La figure 1.1 illustre ce processus de compilation.

Un compilateur est propre à un langage de programmation et à un environnement informatique, matériel et logiciel. Si j'écris un code source en C, je dois utiliser un compilateur C pour

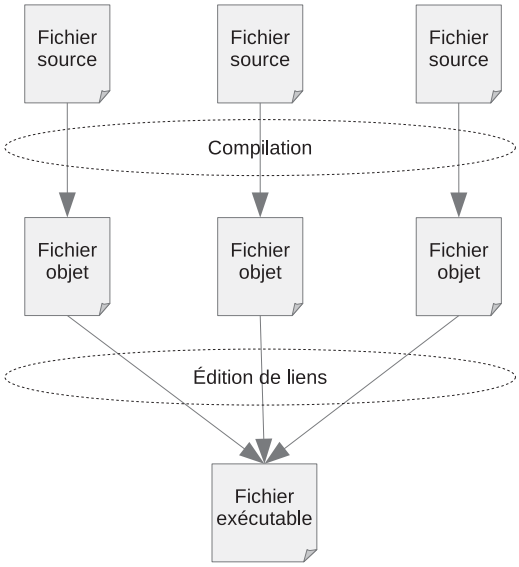


FIGURE 1.1 – Processus de compilation

mon système d'exploitation et pour le type de processeur de ma machine. Le programme exécutable résultant de la compilation pourra être exécuté dans ce même environnement informatique mais il ne sera pas portable sur d'autres environnements. Par exemple, si vous compilez un programme en utilisant le système d'exploitation Linux, il ne pourra pas être exécuté sous Windows (et inversement).

La phase de compilation analyse le code source selon les règles d'écriture du langage de programmation. Si les règles syntaxiques ne sont pas respectées, la compilation va échouer et des messages d'erreurs indiquent les raisons de cet échec. Dans ce cas les fichiers objets et exécutables ne sont naturellement pas créés.

S'il n'y a pas d'erreur à la compilation, les fichiers objets et exécutables sont créés. L'exécution du programme se fait en lançant directement le programme exécutable. La figure 1.2 montre une copie d'écran d'un terminal, sous Linux, dans lequel le fichier source `factoriel.c` est compilé, puis le programme exécutable obtenu exécuté.

Le contenu du répertoire est d'abord listé par la commande `ls` montrant qu'il n'y a que le fichier `factoriel.c`. La compilation est ensuite réalisée avec le compilateur `gcc` (l'utilisation de `gcc` est expliquée dans le chapitre 2 page 27). Elle produit le fichier exécutable `factoriel` qui apparaît maintenant après une nouvelle exécution de la command `ls`. `factoriel` est ensuite exécuté et l'affichage du texte demandant un nombre puis calculant son factoriel sont le résultat de l'exécution des instructions du programme.

Les deux langages utilisés dans cet ouvrage, `C` et `OCaml` sont des langages compilés.