Chapitre 1-9
Algorithmique - Présentation de la méthode

1. Objectifs du chapitre

- Décomposer hiérarchiquement un ensemble de données.
- À partir des décompositions des données écrites (données de sortie) et des données lues (données d'entrée) par le programme, établir l'organigramme structuré de la logique de l'application.
- Écrire le programme à partir de l'organigramme.

2. Présentation de la méthode

2.1 Pourquoi une méthode de programmation ?

Dans un projet informatique, la première chose qu'exprime en général le futur utilisateur, c'est le résultat qu'il souhaite. "Je veux un programme pour gérer mon stock, je souhaite un site web pour présenter ma société...". Il faut alors regarder en quoi consiste le stock, quels sont les renseignements que le site web doit présenter ? Etc.

Enfin, ayant établi avec précision la liste des données disponibles, et l'ensemble des résultats à obtenir, le développeur programme la logique qui permet d'arriver au résultat voulu. On dit qu'il conçoit l'**algorithme** de la solution.

Remarque

Une recette de cuisine est un algorithme. Quel est le plat que je veux réaliser ? Ai-je les ingrédients pour le réaliser ? Enfin, comment m'y prendre ? Quelle est la recette ?

L'art de l'informaticien est de découvrir la recette, l'algorithme à programmer.

Dans les chapitres précédents, nous avons réalisé des programmes sans utiliser de méthode. Un raisonnement logique, traduit convenablement en instructions structurées du langage (`if`, `while`...) suffit souvent pour résoudre les problèmes.

Quand la complexité augmente, que l'ensemble des données à traiter est important, l'algorithme devient moins évident à trouver.

– Le premier intérêt d'une méthode est d'aider l'informaticien à concevoir l'algorithme.
– Le deuxième avantage, c'est qu'une méthode propose des étapes pour arriver au résultat.

Remarque

Si deux personnes suivent les mêmes étapes pour trouver un algorithme, elles arriveront à des programmes semblables. Au sein d'une entreprise, ***une méthode facilite la communication et la maintenance des programmes****.*

2.2 Quelle méthode, pour quels types d'applications ?

La méthode et les notations proposées dans les chapitres concernant l'algorithmique, sont inspirées des travaux de Jean Dominique Warnier, auteur de nombreux travaux sur les systèmes d'information. Plus précisément, les notations (accolades par exemple) sont tirées de sa méthode de Logique de Construction de Programmes.

– La méthode consiste à :
  – étudier les données de sortie (impressions, mises à jour), en examinant la façon dont elles sont organisées et structurées ;
  – étudier les données d'entrée (fichiers), en examinant la façon dont elles sont organisées et structurées ;
  – déduire de ces deux études l'algorithme à programmer.
– Les applications que nous allons réaliser en suivant cette méthode sont connues sous le nom de traitements "**batch**". C'est une partie de l'informatique parfois qualifiée "d'ancienne", mais qui reste d'actualité.
  – Une banque doit mettre à jour les comptes de ses clients en fonction de leurs dépenses et de leurs recettes. Une commune doit éditer son budget. Une entreprise de vente par Internet doit mettre à jour ses stocks. Les quantités d'informations manipulées sont parfois considérables.

– Les informations sont stockées dans des fichiers (ou des bases de données), préparés en vue du traitement.
– Une fois le traitement de ces données lancé, il n'y a plus d'intervention humaine avant l'obtention du résultat. **C'est un traitement batch**. Cette absence de dialogue avec l'utilisateur s'oppose aux traitements conversationnels, où l'utilisateur dialogue avec le programme par l'intermédiaire d'écrans de saisie et d'affichage.

Remarque

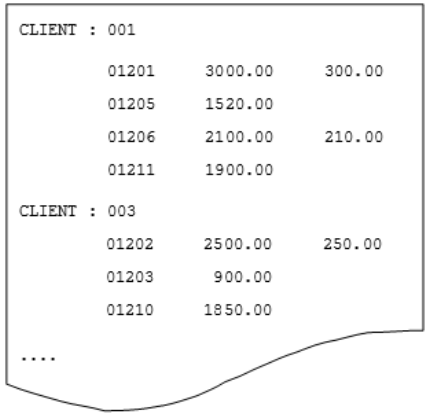
Le langage C est bien adapté aux applications batch. Pour le conversationnel, il est possible d'utiliser des bibliothèques graphiques spécialisées, ou d'utiliser un des nombreux enfants du C : C++, PHP, C#, Java.

2.3 La démarche hiérarchique

Il s'agit de trouver la logique avec laquelle un ensemble de données est organisé, en allant du général au particulier.

3. Exemple : édition de factures

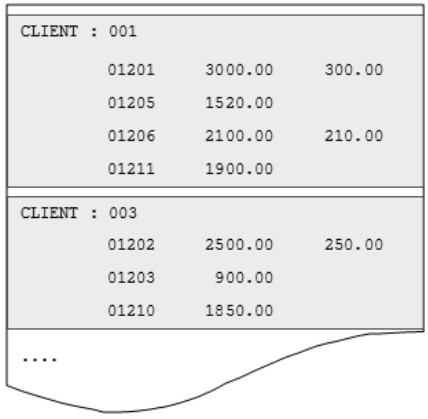
3.1 État de sortie à obtenir



Pour chaque client, on imprime des lignes "facture". Chacune d'entre elles contient un numéro de facture, le montant de la facture, et une remise de 10 % sur le montant de la facture si celui-ci dépasse 2.000 €.

3.2 Décomposition de l'état de sortie

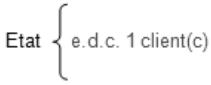
Quand on examine l'état, on constate qu'il est organisé "client par client" :



Chaque sous-ensemble grisé est un ensemble de données concernant un client.

L'ensemble de l'état peut être décrit comme **un ensemble de données concernant un client autant de fois qu'il y a de clients**.

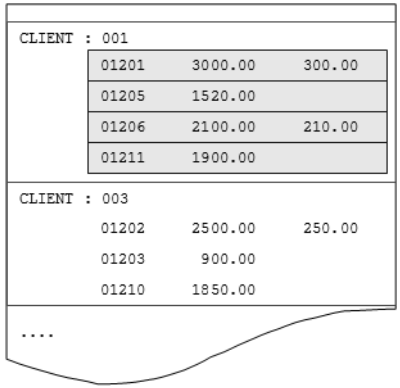
On peut le noter ainsi :



`e.d.c.` : abréviation pour « ensemble de données concernant ».

`(c)` : cardinalité répétitive pour : autant de fois qu'il y a de clients.

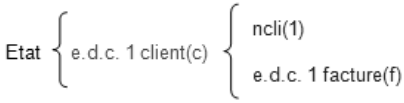
On s'intéresse maintenant à l'organisation des données d'un client, par exemple à celles du `client 001` :



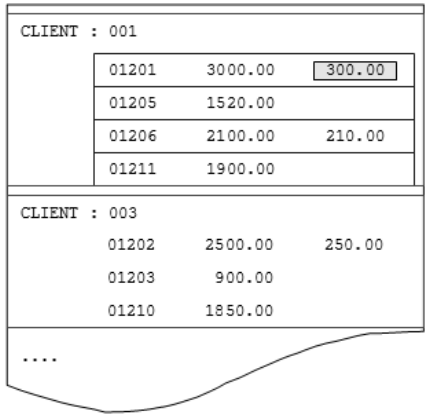
Chaque sous-ensemble grisé est un ensemble de données concernant une facture.

L'ensemble des données d'un client peut être décrit comme : un numéro de client (`ncli`) présent 1 fois, puis un **ensemble de données concernant une facture autant de fois qu'il y a de factures**.

On peut le noter ainsi :



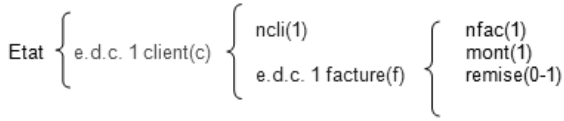
On s'intéresse maintenant à l'organisation des données d'une facture :



Le sous-ensemble grisé correspond à une remise.

L'ensemble des données d'une facture peut être décrit comme : un numéro de facture (`nfac`) présent 1 fois, un montant de facture (`mont`) présent une fois et **une remise présente 0 ou 1 fois**.

Décomposition complète :



3.3 Le fichier des factures

Les informations nécessaires à l'édition des factures sont lues dans un fichier d'entrée dont les enregistrements ont la structure suivante :

N° CLIENT	N° FACTURE	MONTANT

Ce fichier séquentiel est trié sur le N° CLIENT. Il ne comporte qu'un enregistrement par facture.

Structure correspondante :

```
typedef struct
{
    char  ncli[4];
    char  nfac[6];
    float mont;
} FACTURE;
```

On considère que le fichier des factures n'est pas vide.