Programmes

- Objet : Programmes
- Suivi :
  - Création par smolski le 03-05-2013
- Commentaires sur le forum : Lien vers le forum concernant ce tuto[1)]

Généralités

1. Un ordinateur ne comprend que des 0 et des 1 (du binaire).
2. Un développeur (la personne qui va créer les programmes) doit faire en sorte de lui donner les instructions nécessaires pour que son programme réalise les tâches souhaitées.

Ecrire tout en langage binaire serait fastidieux (voire impossible). C'est pour cela qu'il existe des langages de programmation beaucoup plus simples à appréhender.

Ceux-ci possèdent des syntaxes prédéfinies et normalisées permettant par exemple de spécifier que l'on souhaite qu'un message soit affiché à l'écran ou qu'un traitement soit réalisé 3 fois de suite.
Cela est réalisé à l'aide de mots-clés, instructions et symboles spécifiques au langage considéré.
Connaître ces derniers et les mécanismes sous-jacents constitue l'apprentissage d'un *langage de programmation*.

Ecrire un programme sera en fait l'édition d'un simple fichier texte facilement lisible par un programmeur appelé fichier source ou, par raccourci, tout simplement source.

En réalité, si le programme est important, il y aura *plusieurs fichiers* correspondant à des sous-parties de celui-ci. Et plusieurs personnes travailleront dessus.
Il y a plusieurs étapes pour cette conception pouvant être réalisées par des équipes différentes.

Déroulement d'un développement

Les étapes données ici concernent plutôt ce que l'on rencontre dans les entreprises réalisant des développements.
Toutefois de grands projets libres peuvent suivre tout ou partie de ce déroulement. Et rien n'empêche un développeur isolé de s'en inspirer.

Tout d'abord les besoins sont identifiés.

1. Il s'agit d'établir ce que le programme doit faire et quel sera son comportement face aux sollicitations.
2. Ceci peut être fait par une personne qui se positionne comme client du produit fini.
3. Ce travail amène l'élaboration du cahier des charges.

Si le programme est destiné à avoir une interface graphique, on peut aussi établir un prototype de celle-ci.

Un simple croquis présentant le positionnement des éléments de l'interface peut suffire.

Commence alors une analyse du projet.
Il s'agit d'établir une modélisation du programme.
Cela est éventuellement réalisé par des spécialistes du domaine d'application.
Un logiciel devant faire de nombreux calculs par exemple sera étudié par des mathématiciens afin d'établir les formules nécessaires.

Cette modélisation amène aussi des découpages en sous-programmes. Des *normes de modélisations* comme l'**UML** (Site officiel UML) permettent de réaliser des graphiques représentant aussi bien les cas d'utilisation que les parties devant être programmés.
De tels *schémas*, définis par des conventions de notation, facilitent la communication entre les équipes, techniques ou non.

Peut alors commencer l'écriture de code elle-même.
Elle se réalise par étapes successives.
A chaque fois qu'une partie plus ou moins indépendante est terminée, on procède à des tests unitaires, réalisés le plus souvent par les programmeurs eux-mêmes.

Lorsque les différents composants sont intégrés ensemble, des tests de fonctionnalité globale sont réalisés tout d'abord par les développeurs, mais surtout ensuite par le client qui a demandé le projet.
On compare alors le résultat avec ce qui avait été demandé dans le cahier des charges.

C'est ce que l'on appelle la r*ecette du projet*.

Un programme n'étant pas figé une fois pour toutes, il peut y avoir besoin d'y ajouter des fonctionnalités par la suite.
Lorsque celles-ci ont été intégrées, on procède à des *tests de non-régression*. Cela afin de vérifier que l'existant n'est pas modifié de manière indésirable par les nouveaux ajouts.

Pour finir sur cette brève présentation du développement d'un logiciel, il faut parler du développement coopératif.

Etant donné que plusieurs développeurs travaillent simultanément sur les mêmes fichiers sources, il existe des systèmes pour coordonner cela.

Le plus connu et utilisé est probablement CVS (Site officiel CVS).
Il permet de gérer des versions et de verrouiller des fichiers pour qu'une seule personne à la fois puisse le modifier.
On peut avec un tel outil revenir facilement à une ancienne version en cas de problème ou fusionner le travail de plusieurs programmeurs.

Compilation

Une fois le programme écrit, il faut faire en sorte qu'il puisse être compris par la machine. Cela se réalise grâce à l'opération de compilation.
Elle se fait au moyen d'un compilateur qui est lui-même un programme.
Il va transformer le fichier source (qui n'est que du texte) en un programme compréhensible par l'ordinateur (en binaire).

En réalité, il y a une étape intermédiaire. Le fabricant du microprocesseur le dote d'un jeu d'instructions baptisées *instructions assembleur*.

Dans un premier temps, le source est donc transcrit en assembleur qui constitue aussi un langage de programmation.

On peut développer directement dans ce langage. Mais cela devient de plus en plus rare sauf pour des parties très proches du système, comme le coeur d'un système d'exploitation ou des parties de pilotes de périphériques.

Et ensuite sera créé l'exécutable.
Généralement ce sera un fichier .exe sous Windows et un binaire ELF pour Linux.

Ces fichiers exécutables comprennent en plus des instructions destinées au processeur, des informations pour le système d'exploitation.
Celles-ci indiquent de quelle manière il doit être executé et aussi des informations permettant la recherche de problème dans le logiciel (ceci est appelé le débuggage ou déverminage).

C'est tout cela (en plus de l'utilisation des librairies) qui fait qu'un programme destiné à un système ne fonctionnera que sur celui-ci une fois compilé.

Quelques exemples de langages compilés: **C**, **C++**, **Pascal**, ...

Langages interprétés (scripts)

Il existe des langages de programmation qui n'ont pas besoin de compilation.
Ce sont les langages interprétés appelés aussi langages de script.
Le programme est là encore écrit dans un fichier texte, mais il peut alors être directement exécuté.

Ce type de programme nécessite un interpréteur.
Il joue un peu le même rôle que le compilateur, à savoir transformer des instructions texte en langage machine, si ce n'est qu'il le fait au fur et à mesure.
A chaque nouvelle ligne du fichier, les commandes adéquates sont envoyées au processeur.
Cela a pour conséquence d'avoir des temps d'exécution plus long, la conversion prenant du temps.

Mais en contrepartie, ces programmes sont plus faciles à modifier et peuvent immédiatement être testés sans devoir attendre le temps de la compilation.

Un autre avantage est une plus grande portabilité.
Si un système dispose d'un interpréteur pour un langage donné, on peut alors y utiliser un script, même écrit sur un autre système avec un processeur différent (et donc un langage assembleur différent).

Pour lancer un tel programme, on appelle en fait l'interpréteur en lui passant en paramètre le fichier texte qu'il doit exécuter.

Sous GNU/Linux, ce peut aussi être réalisé à l'aide d'une syntaxe particulière. La première ligne du fichier doit être de la forme :

```
#!/chemin/vers/l/interpreteur
```

Le programme peut alors être lancé comme n'importe quel exécutable.

Les shell-scripts sont des exemples de ces programmes.
PHP, utilisé notamment pour ce site, est également un tel langage.
On peut aussi citer entre autres Perl, Python et Tcl.

Librairies

Lorsqu'un nouveau programme est développé, celui-ci aura besoin de réaliser des tâches ou des calculs qui auront déjà été faits par d'autres auparavant.
Par exemple plusieurs programmes ont besoin de calculer le cosinus d'un angle ou d'afficher un bouton.

Pour cela, il existe les *librairies*.
Leur rôle est de *regrouper des ensembles de fonctionnalités* pouvant ensuite être réutilisées.
On peut avoir, pour reprendre les exemples précédents, une librairie mathématique ou une de composants graphiques.

On distingue 2 types de librairies:

1. Les statiques et
2. les dynamiques.

Sous Windows, les premières portent l'extension .lib et les secondes .dll alors que pour Linux ce sont respectivement .a et .so qui sont utilisées.

Les *librairies statiques* sont incluses dans le programme lui-même.
On obtient un seul gros fichier exécutable dans lequel les fonctions utilisées sont copiées.

A l'opposé, les *librairies dynamiques* sont présentes dans des fichiers séparés et ne seront chargées que lorsqu'elles sont nécessaires.

Si plusieurs programmes utilisent simultanément les mêmes librairies partagées, elles ne seront mises en mémoire qu'une seule fois.
C'est notamment le cas des librairies mises à disposition par le système d'exploitation ou l'interface graphique qui sont souvent sollicitées.

Ces dernières ont donc des avantages. Mais malheureusement, il faut qu'elles soient présentent sur le système pour que le programme fonctionne.
Il est possible de la distribuer avec le logiciel pour en être sûr, mais souvent elle devra être présente sur la machine.

Si ce n'est pas le cas, l'exécution conduira à une erreur du style “Librairie non

trouvée”.

Nota :

Avec une *librairie statique*, l'utilisateur ne sait même pas en fait que la librairie est utilisée car elle se trouve dans le programme au même titre que ce qui a été développé explicitement pour celui-ci.

Source

- SAIT - Cours Programmation

1)

N'hésitez pas à y faire part de vos remarques, succès, améliorations ou échecs !