Aspects historiques

a. Langages dynamiques

Un chef de projet informatique, quelque soit son domaine d'intervention, doit savoir maîtriser les concepts sous-jacents aux langages de programmation, ceux qui marquent les différences profondes entre les langages actuels, passés et futurs, ainsi que différentes notions relatives au sens et à la correction des programmes. Cette connaissance informatique lui sera clairement nécessaire s'il contribue, ou s'il dirige des développements informatiques, mais elle le lui sera aussi utile lorsqu'il aura à évaluer des éléments logiciels, ou à participer à des prises de décision stratégiques impliquant des choix en matière de logiciel.

Programmer un ordinateur c'est lui donner des instructions que la machine doit exécuter. Ces ordres sont écrits dans un langage informatique particulier, nommé langage de programmation. Comme tout langage informatique, et contrairement à une langue humaine, ce langage est rigide, fixe et normalisé de façon à ce que l'ordinateur puisse contrôler, vérifier et exécuter les phrases de ce langage.

Chaque langage peut-être classé suivant :

- *Son objectif*,
- *Son type de traitement* des programmes,
- *son importance*,
- *et enfin son type de licence ainsi que son cout*.

Plusieurs de ces caractéristiques peuvent correspondre à un ou plusieurs langages dans un bon nombre de version à la fois, sans pour autant être contradictoire.

Le but d'un langage peut être le *calcul* scientifique (Fortran, Apl) ou mathématique (Maple, Mathematica), ce peut être aussi un but de travail de *gestion* (Cobol) ou de *bases de données* (Dbase, Sql, Sasl). L'objectif peut enfin d'être déductif ou logique (Prolog) à moins que ce ne soit un objectif universaliste pour "tout" programmer avec ou sans interface graphique (C, Pascal, Java, Perl).

Le type de traitement des instructions se fait via la compilation ou l'interprétation. La compilation traduit l'ensemble des instructions de départ nommé le "code-source" en un

code machine intermédiaire qui après une édition de liens est un code-assembleur exécutable par la machine. On peut ne fournir que cet "exécutable" et l'utilisateur ne sait alors rien de ce qu'il y a dans le programme sauf ce qu'il utilise et ce qu'il voit dans l'interface utilisateur (menus, fenêtres...) si elle existe. L'*interprétation* consiste à lire les instructions une à une au fur et à mesure de l'exécution. L'analogie classique avec un serveur de restaurant compare la compilation à la prise de la commande pour l'ensemble d'un groupe de personnes alors que l'interprétation consiste à demander à une personne ce qu'elle veut, traiter sa commande puis passer à une autre personne ainsi de suite. De nombreux interpréteurs ont aujourd'hui un mécanisme de *pré-compilation*, notamment pour tester la validité de la syntaxe des programmes. Pour exécuter un programme interprétable, il faut disposer du code-source et d'un "exécutable" nommé "interpréteur du langage". Traditionnellement C, C++, Pascal, Ada, Java, Fortran, Cobol... sont des langages compilés alors que Apl, Awk, Rexx, Perl, Prolog, Javascript, Tcl/Tk, sont des langages interprétés.

Dans un langage de programmation, on peut soit indiquer comment faire avec un langage impératif, soit indiquer ce que l'on veut faire avec un langage déclaratif, le mécanisme de gestion du langage gérant alors comment réaliser ce qu'on veut faire.

Les langages **impératifs** peuvent être divisés en trois catégories :

- les langages *peu structurés* comme Basic, Fortran,
- les langages *structurés par blocs* comme Pascal, C,
- les langages dits *orientés objets* comme SmallTalk, C++, Java, Ada.

Les langages **déclaratifs** ont des natures différentes pour lesquels on distingue :

- les langages *fonctionnels* (Lisp),
- les langages *logiques/à contraintes* (Prolog),
- les langages orientés *bases de données* (Sql).

Certains langages sont **typés explicitement** ce qui signifie que les variables doivent être déclarées selon des moules prédéfinis nommés types. Le typage impose des restrictions car il contraint le programmeur à déclarer beaucoup de choses. En retour la machine peut vérifier que tout se passe comme prévu et que rien ne "déborde". Traditionnellement, les langages de script (Awk, Rexx, Perl) sont non typés explicitement. Par contre les langages compilés (C, C++, Pascal, Ada, Java, Fortran, Cobol) sont pratiquement toujours [fortement] typés explicitement.

Pour les langages impératifs, l'exécution d'un programme est linéaire, c'est à dire qu'elle s'effectue d'une instruction à la suivante, sachant que l'instruction suivante n'est pas forcément la ligne consécutive dans le fichier à cause des mécanismes de "contrôle de flot" que sont les tests, les boucles et les appels de sous-programmes.

Les langages impératifs classiques de la famille dite de « Von Neumann » sont ceux qui ont eu le plus grand succès grâce à la proximité des concepts qu'ils offrent avec l'architecture des machines sur lesquels les programmes s'exécutent. Ils exposent en effet au programmeur la mémoire et les fonctionnalités d'écritures. La mémoire est ici vue comme un état global ou local modifiable. Les programmes sont donc constitués d'instructions qui vont essentiellement modifier la mémoire. Ces modifications de l'état ou de l'environnement du programme sont appelés effets de bord : ils ne sont en effet pas détectables au simple vu du type des fonctions ou procédures. Il s'agit plus d'effets imputables à l'exécution d'un morceau de programme. Ces effets de bord sont appelés « side effects » en anglais.

Les langages orientés objets ont bénéficié d'une popularité récente, relativement aux langages impératifs classiques, même si on peut faire remonter leur origine au langage Simula conçu avant 1970.

Les langages orientés objets offrent une organisation des programmes radicalement différente de la vision classique qui voit le programme comme un calcul opérant avec l'aide de structures de données, et dont la structure vise à obtenir une structuration claire de ce calcul. La vision objet privilégie les structures de donnée objets, qui sont, dans les langages orientés objets, organisés en familles de classes et possédant des attributs et des méthodes. Ainsi, le programme reste bien sûr un calcul, mais l'organisation de ce calcul est bien différente de l'organisation classique. Les objets ont chacun leur état, sorte de mémoire privée modifiable, et le modèle de calcul reste généralement un modèle impératif. Le

langage objet le plus pur est le langage Smalltalk, dans lequel toute donnée est un objet. C++ est probablement le plus utilisé. Les langages majeurs récemment apparus que sont Java et C# sont tous les langages `a objets, ainsi que les langages de script que sont Python ou Ruby. Les langages fonctionnels peuvent eux aussi offrir un sous-système orienté objets comme Common Lisp (CLOS : Common Lisp Object System), mais dans un modèle de calcul qui garde des aspects potentiellement impératifs.

Pour un langage déclaratif, une fois le but à atteindre ou le calcul à effectuer, l'exécution du langage suit un mécanisme d'exploration, de résolution ou d'interrogation qui est en général propre au langage.

Les langages fonctionnels sont essentiellement des langages d'expressions (par opposition aux langages d'instructions) où les calculs sont des évaluations d'appels de fonctions. Certains d'entre eux sont dits « purs », au sens où ils ne disposent pas du tout d'opérations de modification de variables ou de modification de la mémoire. Ces langages privilégient toutefois la notion d'expression, de définition et d'appels de fonction. On notera que, de ce fait, on peut discuter du bien fondé du classement de ces langages fonctionnels dits « impurs » dans la catégorie des langages déclaratifs :

Ce sont en effet des langages o`u l'expression des programmes est plus algorithmique que purement déclarative.

Les langages logiques ou à contraintes sont probablement les meilleurs exemples de langages déclaratifs si l'écriture d'un programme consiste réellement à l'exposition du problème : les faits connus, les schémas de déduction possibles et la question posée dans le cas de la programmation logique, les contraintes à satisfaire et le problème à résoudre dans le cas des langages à contraintes. Le mécanisme d'exécution des programmes peut alors se schématiser comme un mécanisme de résolution qui va chercher à répondre à la question posée dans chacun de ces deux cas. Dans ces langages, on n'écrit pas d'algorithme : c'est le langage qui fournit une sorte d'algorithme universel. Il est toutefois bien utile de comprendre de manière précise le mécanisme de recherche de solution fourni par le langage pour adopter un style de description qui lui soit bien adapté. Le langage logique le plus connu est le langage Prolog, conçu en France, et qui a connu son heure de gloire dans les années 1980 après qu'il ait été mentionné dans un grand projet japonais de conception d'ordinateurs dits « de cinquième génération ». L'heure est plutôt aux langages de

résolution de contraintes, et il se trouve que les langages du style Prolog s'enrichissent assez naturellement de mécanismes de résolution de contraintes.

Le cout du langage à l'achat varie de *zéro* pour des langages libres ou avec une licence *GNU*, comme pour Awk, Rexx/Regina, Perl, Gcc à plusieurs centaines voire plusieurs milliers d'euros pour les logiciels développés par des sociétés privées, comme par exemple Maple, Oracle...

L'apport de la méthodologie objet a contraint certains anciens langages à modifier la syntaxe de base pour conserver des adeptes. Ainsi Pascal, langage non objet de conception est devenu Pascal Objet puis Delphi ; de même Lisp non objet à ses débuts a vu une variante nommée Clos lui conférer le statut de langage à objets. On nomme souvent ces langages *"orientés objets"* pour les distinguer des langages vraiment objets ou encore *"langages objets purs"* dont SmallTalk est le plus pur, suivi par C++ et Java.

S'il est relativement facile d'apprendre un petit langage comme Perl lorsqu'on sait déjà programmer dans un autre langage, il est beaucoup plus difficile d'aborder des langages originaux comme Lisp ou Prolog. La plupart des langages impératifs ont d'ailleurs souvent une syntaxe voisine qui permet d'utiliser un **langage algorithmique** commun que l'on a implémenté pour en français offrir une traduction presque totalement automatique dans ces langages, ce qui ne peut pas être le cas ni pour ni pour Lisp ni pour Prolog.

Dans ce genre de langage algorithmique impératif, il y a peu d'instructions à connaitre, la richesse et la difficulté venant du nombre d'instructions contenues dans l'algorithme ou le programme. On distingue classiquement

- les commentaires,
- les affectations,
- les structures (tests et boucles),
- les appels de "fonctions standards",
- les appels de sous-programme.

Le but des algorithmes est de faciliter la communication entre humains alors que le but des programmes est de commander les machines. Les algorithmes sont donc souvent plus lisibles, moins ambigus, moins contextuels au fur et à mesure du temps qui passe. Si on utilise une même notation dans les langages pour indiquer la fin d'une instruction

composée, c'est parce que cela fait moins de texte à manipuler. La gestion par l'ordinateur en sera plus rapide. Par contre, pour un humain, avoir plus de mots est un gage de meilleure lisibilité.

Les grandes dates sont les suivantes:

- Années 50 : Création des langages de haut niveau (plus proches de l'homme).
- Années 60 : Foisonnement de langages spécialisés. Forth. Simula I. Lisp, Cobol. On essaie sans succès d'imposer des langages généraux: Algol, PL/1.
- Années 70 : Duel entre programmation structurée avec Pascal et l'efficacité du langage C (cela dure encore en 2000). Généralisation du Basic interprété sur les micro-ordinateurs apparus en 1977, jusqu'à la fin des années 80.
- Années 80 : Expérimentation d'autres voies et notamment des objets. ML. Smalltalk.

  Sur les micro-ordinateurs, on utilise maintenant C, Pascal, Basic compilé.
- Années 90 : Généralisation de la programmation objet grâce aux performances des micro-ordinateurs. Java, Perl, Python s'ajoutent aux langages micros.
- Années 2000 : Programmation Internet (et les innovations à venir)

b. Systèmes distribués

Tout commence au début des années 1960. Paul BARAN, travaillant pour une agence gouvernementale, a été commandée par l'US Air Force dans le but d'analyser « comment le gouvernement pourrait maintenir le contrôle sur les armes complexes après une attaque nucléaire ». Ce qu'il a introduit dans le monde est un système d'ordinateurs qui devaient être décentralisés. Ces ordinateurs communiquaient à des tiers en utilisant les réseaux permettant l'envoi, le déroutage des informations. A partir de là a été développé le réseau. En effet, le premier programme d'e-mail a été créé par Ray Tomlinson de BBN. ARPANET. Ce qui a permis le déploiement des protocoles pouvant transférer des données. La communication entre les machines qui tournent désormais sur le même réseau.

En 1973, le développement a commencé sur le nouveau protocole qui a été plus tard appelé TCP / IP. Ce nouveau protocole a permis divers réseaux d'ordinateurs de s'interconnecter et de communiquer les uns avec les autres. Le terme «Internet» a été utilisé pour la première fois par Vint Cerf et Bob Kahn dans le document sur la Transmission Control Protocol.

Définition

Le terme de système distribué a plusieurs définitions, suivant l'auditeur. La plus simple est celle-ci : il s'agit d'un système dont la représentation logique se différencie de son implémentation physique. Cette implémentation physique est constituée par un ensemble d'entités indépendants de calcul tels que les ordinateurs, les PDA,… interconnectés entre eux par un réseau de communication. De l'autre côté, la représentation logique est celle que l'utilisateur final perçoit. Voici quelques exemples de systèmes distribués : le serveur de fichiers, le web, les calculs scientifiques.

Pourquoi les systèmes distribués sont nés ?

La naissance des systèmes distribués dérive de plusieurs besoins qui sont les suivant :

- Aspects économiques (rapport prix/performance).
- Adaptation de la structure d'un système à celle des applications (géographique ou fonctionnelle).
- Besoin d'intégration (applications existantes).
- Besoin de communication et de partage d'information.
- Réalisation de systèmes à haute disponibilité.
- Partage de ressources (programmes, données, services).
- Réalisation de systèmes à grande capacité d'évolution.

Les enjeux

Voici les enjeux principaux des systèmes distribués :

- la maîtrise de la complexité des applications.
- la sûreté des logiciels et des systèmes qui les accueillent.

- la diminution du temps de développement des produits.
- le traitement en temps réel d'un volume toujours croissant d'informations.
- la robustesse des architectures matérielles et logicielles face aux contraintes d'environnement.