CONDUITE D'UN PROJET ET QUALITÉ

(suite et fin)

Alain VAN SANTE

Une bonne démarche de production s'appuie sur les trois opérations suivantes :

- L'abstraction pour favoriser le choix des modules
- La décomposition pour diviser le logiciel en ces différents composants
- L'intégration pour regrouper les différents modules

L'ABSTRACTION

L'abstration est l'opération qui consiste à simplifier l'analyse en séparant les éléments pertinents de ceux qui ne le sont pas.

Dans la vie courante, nous utilisons naturellement cette technique qui peut se résumer à la phrase suivante : **Aller à l'essentiel**.

Nous manipulons régulièrement des abstractions :

- Une carte routière représente une abstraction du réseau routier, qui présente l'essentiel et élimine le superflu.
- La notion de fichier dans un langage de programmation est une abstraction d'une organisation de stockage complexe, manipulée grâce à des primitives simples.
- Un modèle conceptuel des données représente une abstraction du système d'information d'une entreprise.

En programmation, la technique d'abstraction est fortement facilitée par l'usage des langages de haut niveau qui permettent l'extension de leurs fonctionnalités par la déclaration de procédures et de fonctions, voire d'objets et de méthodes.

L'abstraction repose sur l'usage et la construction de structures de données et de primitives de manipulation de ces structures.

Elle peut prendre deux formes :

L'ABSTRACTION PAR PARAMÉTRISATION OU GÉNÉRALISATION

C'est l'opération qui consiste à représenter un ensemble infini de calculs par un texte unique, abstraction de chacun d'entre eux, grâce à l'usage des paramètres.

Elle permet la réutilisabilité d'un module sur des objets différents.

Exemple : Factorielle (N) donne la factorielle de N. Elle peut être utilisée pour tout N entier positif.

Nous devons alors décrire le mode de calcul de la factorielle d'un entier quelconque et non le calcul pour un entier précis.

L'ABSTRACTION PAR SPÉCIFICATION

C'est l'opération qui consiste à représenter un module par ses objectifs (le quoi), en décrivant :

- la précondition : propriétés qui doivent être vraies à l'entrée du module
- la postcondition : propriétés supposées vraies à la fin de chaque exécution du module avec une précondition satisfaite

Exemple : Factorielle(N)

Avant : N est un entier positif ou nul

Après : Factorielle retourne :

N! = N*(N-1)*...*1 pour N >0, 1 pour N = 0

Ces deux opérations sont le plus souvent étroitement liées.

Nous allons maintenant les détailler à travers les deux types d'abstractions fondamentaux :

- l'abstraction procédurale
- l'abstraction de données

L'ABSTRACTION PROCÉDURALE

C'est le premier type d'abstraction que réalise tout programmeur débutant lorsqu'il crée ses premières procédures ou fonctions.

C'est l'opération qui consiste à étendre la machine virtuelle (représentée par l'ordinateur et le langage de programmation) par de nouvelles primitives.

L'abstraction par paramétrisation généralise le concept d'action nommée en conduisant à la création de ce que nous appelons des actions paramétrées. Nous nous intéressons au nombre et au type de chaque argument, sans nous préoccuper de son identité, remplacée par une variable muette : l'argument formel.

L'avantage d'une telle démarche réside dans la réutilisabilité des procédures et fonctions ainsi définies (si toutefois ces actions ou fonctions sont suffisamment génériques et bien documentées).

L'abstraction par spécification consiste à insister sur le comportement du logiciel intéressant l'utilisateur, en faisant abstraction des détails d'implantation. On se préoccupe du quoi et non du comment.

Il importe de fournir une description claire et non ambiguë des spécifications. Le mieux est encore d'utiliser le langage naturel, associé éventuellement à des représentations schématiques en cas d'ambiguïté.

Pour chaque procédure ou fonction, il est essentiel de préciser :

- L'en-tête : nom de la fonction ou de la procédure
  Rôle
  Nombre, ordre, nom et type de ses données et de ses résultats
- La sémantique
  - Conditions d'utilisation (précondition)
  - Arguments modifiés et comportement (postcondition)

Les principales qualités d'une abstraction procédurale sont :

La localité

L'implantation d'une abstraction peut être lue ou construite sans qu'il soit nécessaire d'examiner les implantations d'autres abstractions.

On peut donc utiliser une abstraction sans connaître les détails de son implantation. On peut même l'utiliser dans la démarche de conception avant qu'elle soit réellement implantée.

Exemple : on peut utiliser une procédure de tri sans connaître l'algorithme utilisé.

La modifiabilité

Il est possible de modifier l'implantation d'une abstraction procédurale sans modifier les programmes qui l'utilisent.

Exemple : il est possible de changer un algorithme de tri utilisé par une procédure sans modifier les programmes qui l'appellent.

Pour atteindre cet objectif qualitatif, certaines règles doivent être respectées.

La minimalité

La description du comportement d'une abstraction procédurale ne doit pas contenir d'éléments superflus. Elle ne doit pas préciser la méthode d'implantation (sauf en cas de contraintes de performance).

La généralité

Elle doit faire un usage maximal des paramètres.

La simplicité

Le rôle de l'abstraction doit être clairement défini et facilement explicable. Il doit être indépendant du contexte d'utilisation (il doit toujours pouvoir être résumé à un nom significatif).

L'utilité

L'abstraction procédurale doit être véritablement utile. Il ne faut pas chercher à multiplier les procédures et fonctions.

La sûreté

Une procédure doit en théorie réagir à toutes les situations (pas de précondition), ce qui lui permet d'être réutilisable dans n'importe quel contexte. En pratique on préférera souvent des procédures partielles plus

restrictives (présence d'une précondition) lorsque leur usage est plus limité. Tous les cas d'exceptions doivent pouvoir être pris en compte.

L'ABSTRACTION DE DONNÉES

L'abstraction de données consiste à créer de nouveaux types de données abstraits (noté TDA ou TAD) constitués d'un ensemble d'objets et d'une famille de primitives de gestion associées.

Nous utilisons de telles abstractions en créant une pile ou une file d'attente, en nous référant à leur usage et non à leur implémentation.

L'objectif est toujours de se rapprocher des préoccupations de l'utilisateur en travaillant sur des machines fictives permettant de résoudre plus facilement le problème (comme de voir une phrase comme une suite de mots avant de la voir comme une suite de caractères).

La paramétrisation permet encore une fois la réutilisabilité de la structure par généralisation.

La spécification permet de décrire la structure à travers ses primitives de gestion, en faisant abstraction de la représentation interne.

Une abstraction de données doit donc contenir :

- une description générale du nouveau type
- la spécification de chaque primitive

On doit retrouver les opérations de création, suppression, modification et consultation.

En imposant que tous les programmes ne fassent accès à la structure qu'à travers les primitives (en nombre suffisant), on aboutit aux même qualités que les abstractions procédurales : LOCALITÉ et MODIFIABILITÉ.

On aboutit ainsi à l'effet de « boîte noire » qui permet à un programmeur d'utiliser une structure complexe sans en connaître l'implantation.

Un cas particulier de l'abstraction de données concerne les primitives de parcours d'une structure. Les langages de programmation disposent de structures comme les fichiers qui peuvent être parcourues à l'aide de primitives prédéfinies (ouvrir, lire, fermer, fin).

Nous savons que tout traitement itératif peut s'analyser comme le parcours d'une suite d'éléments. Inversement toute structure de données doit pouvoir être parcourue à l'aide d'une répétitive, à condition de disposer des primitives suivantes :

Ouvrir	: prépare le parcours séquentiel de la structure
Suivant	: fournit l'élément suivant dans la structure (le premier après Ouvrir)
Fin	: retourne vrai si l'élément suivant n'a pu être atteint
Dernier	: retourne vrai si l'élément atteint est le dernier
Fermer	: termine le parcours de la structure

La présence des deux indicateurs (Fin et Dernier) ou d'un seul d'entre eux dépend généralement des spécifications.

Cette démarche est très utile lorsque les structures à manipuler sont complexes ou lorsqu'elles ne sont pas directement compatibles avec les résultats à obtenir [1].

LA DÉCOMPOSITION

La décomposition du problème s'appuie toujours sur une démarche d'analyse descendante. Elle repose sur le concept de machine abstraite dont nous avons déjà parlé.

Il s'agit de partir des spécifications du problème et de rechercher la ou les machines abstraites (abstraction de données) qui permettront de simplifier au mieux le problème et les principaux traitements agissant sur ces données (abstractions procédurales).

Chacune de ces abstractions est ensuite étudiée indépendamment et éventuellement décomposée selon le même schéma.

Cette méthode repose sur deux exigences :

- Disposer à tout moment d'une documentation claire et précise fournissant les spécifications de toutes les abstractions isolées.
- Disposer d'une indication concernant la dépendance des abstractions entre elles. On utilise de façon classique un graphe de dépendance dans lequel les nœuds représentent les abstractions

[1] Sur ce thème, voir les articles consacrés à la méthode JSP (Trace n°1 et de 6 à 11).

(données ou procédures) et les arcs orientés la relation de dépendance « utilise ».

Un type peut utiliser un autre type ou une procédure.

Une procédure peut utiliser un type ou une autre procédure.

L'analyse d'un tel graphe doit permettre de construire une hiérarchie des abstractions et de déterminer dans quel ordre il est souhaitable de les analyser.

Lors de l'implantation, on pourra choisir :

Une démarche ascendante. Dans ce cas, on construit d'abord les éléments de plus bas niveau. Il faut alors généralement écrire des programmes supplémentaires pour les tester : on les appelle des **LANCEURS**.

Exemple : dans le programme d'impression des références croisées d'un programme source dBASE, on construira d'abord les abstractions d'arbres binaires et de files d'attente nécessaires à la mémorisation de la liste ordonnée des mots utilisateurs (dictionnaire), qu'on testera à l'aide d'**un programme écrit spécialement à cet usage**.

Une démarche descendante. On construit d'abord les éléments de plus haut niveau. Il faut alors écrire des procédures simulant les éléments de plus bas niveau : on les appelle des **BOUCHONS**.

Exemple : dans le programme d'impression des références croisées d'un programme dBASE, on pourrait s'intéresser d'abord à la partie concernant la constitution de l'arbre des mots utilisateurs, en utilisant l'abstraction procédurale permettant d'extraire les mots pertinents du programme traité. Pour tester cette partie, il faudrait alors simuler la procédure permettant l'extraction d'un mot du programme, en constituant par exemple un fichier texte contenant un mot pertinent par ligne et en écrivant le **programme qui le traitera séquentiellement**.

UNE IMPLANTATION dBASE

La qualité et la facilité de l'implantation nécessite le choix d'un langage approprié.

Il est cependant très important de comprendre qu'un langage n'est pas tout et qu'on peut faire d'excellents programmes et d'utiles biblio

thèques de modules réutilisables avec des langages inadaptés (ou considérés comme tels), qu'on peut aussi "saloper" son travail dans le meilleur des langages (*que je ne citerai pas, car je m'en sens bien incapable*).

Bien entendu, un langage permettant d'implanter simplement tous les concepts vus précédemment et proposant un environnement de développement apte à gérer les versions et la réutilisabilité sera toujours souhaitable. Mais on peut aussi le faire avec tout langage.

Ainsi, le langage dBASE ne permet pas une traduction immédiate de tous les concepts présentés.

On dispose bien sûr des procédures et fonctions, du passage d'arguments par valeur et par variable (bien que le type de passage soit implicite). Il est cependant impossible de passer un vecteur en paramètre et le passage d'un élément d'un vecteur ne s'effectue que par valeur.

De même, l'absence de typage explicite représente un inconvénient important, car il est la source de beaucoup d'erreurs :

- Changement du type à la suite d'une erreur d'affectation.
- Déclaration implicite d'une nouvelle variable à la suite d'une erreur dans l'orthographe d'un nom de variable existant.

Ces défauts doivent être compensés par une documentation plus grande des programmes (indication en commentaires des types de chaque variable déclarée).

Il présente également des avantages, puisqu'un tableau peut contenir des éléments de type différents. Ceci permet de pallier l'absence d'une structure de type enregistrement (RECORD en PASCAL). On pourra donc définir une structure enregistrement à l'aide d'un vecteur dont chaque élément représentera un champ, cette solution étant cependant limitée par le manque de lisibilité (un champ étant repéré par un numéro, plutôt que par un nom significatif) et par le problème du passage des paramètres.

dBASE dispose par ailleurs d'un traitement d'exceptions relativement puissant (instruction ON ERROR) qui permet de prendre en compte toutes les erreurs systèmes. Il est possible de tester l'erreur qui a provoqué le débranchement (fonction ERROR()), la ligne sur laquelle l'erreur s'est produite (fonction LINENO() ou encore le nom de la procédure ou fonction dans laquelle elle s'est produite (fonction PROGRAM()).

Il est alors possible de traiter l'erreur (toute instruction dBASE), d'exécuter une nouvelle fois l'action qui a provoqué l'erreur (RETRY), de

passer à l'instruction suivante en ignorant l'erreur (RETURN) ou d'arrêter le programme (CANCEL).

Mais on peut citer trois limitations importantes :

A - dBASE ne permet pas à une fonction de retourner des données structurées, mais c'est le cas d'en bien d'autres langages.

B - dBASE ne permet pas de limiter l'accès à une structure à travers une série de procédures ou fonctions (concept d'encapsulation dans les langages orientés objets).

Il faut s'obliger à construire autant de fonctions de consultation que d'attributs de la structure. On peut cacher cette structure et toutes ses primitives dans un fichier de procédures (cela devient cependant impossible lorsqu'on utilise plusieurs structures complexes, puisqu'on ne peut ouvrir qu'un seul fichier de procédure à la fois).

C - dBASE ne permet pas d'utiliser une même procédure ou fonction pour des données de types différents (concept de polymorphisme dans les langages orientés objets). Ainsi, il n'est pas possible de prévoir une procédure de tri pour un vecteur d'éléments quelconques.

On peut tourner la difficulté de deux façons :

- Construire des bibliothèques de procédures et fonctions avec des types 'muets'. Il suffit alors au dernier moment de faire une substitution sous éditeur du type muet par le type à manipuler, puis de recompiler.
- Utiliser les possibilités de typage faible de variables. Cette solution est plus lourde et nécessite de passer en paramètre une indication concernant le type à manipuler (sa longueur ou un type énuméré), ce qui la limite généralement aux types prédéfinis (fonctions ISNUMBER(), ISALPHA(), etc.).

Une dernière restriction concerne l'absence de gestion de pointeurs (j'en vois déjà qui font des bonds !). On peut cependant utiliser le concept de déclaration dynamique des variables (possibilités de créer et de supprimer des variables durant l'exécution d'un programme) et celui de « macro » (variable contenant le nom d'une variable) pour résoudre ce problème.

CONCLUSION

Il n'est pas possible de terminer cette rapide présentation sans quelques conseils importants :

- Evitez les erreurs en réfléchissant avant d'agir.
- Utilisez toutes les ressources disponibles pour retrouver vos erreurs, sans en abuser : l'usage d'un COMPILATEUR RAPIDE et l'emploi d'un DEBOGUEUR INTERACTIF ne dispensent pas de réfléchir sur la logique du programme.
- Programmez défensivement : partez du principe que vous allez faire des erreurs et documentez soigneusement vos programmes pour pouvoir les relire facilement, même après des mois d'interruption.
- Tenez à jour une liste des variables globales que vous utilisez, ainsi qu'une liste des sous-programmes. Utilisez toujours des noms significatifs.
- Travaillez sur listing plutôt que sur écran. Vous éviterez la fatigue et économiserez votre temps et celui de la machine.
- Ne vous lancez pas dans des modifications sans avoir longuement réfléchi. Une correction trop rapide peut générer plus d'erreurs qu'elle n'en corrige.
- Ne vous lancez que dans une modification à la fois.
- Notez soigneusement toutes les modifications que vous effectuez sur un programme. Utilisez des marqueurs de couleurs. Si vous travaillez en équipe, prenez vos responsabilités en associant votre nom à la correction.
- Conservez soigneusement les versions successives du logiciel, en les datant de façon précise (date et heure). Il est quelquefois utile de faire marche arrière.
- Travaillez toujours sur une seule version à la fois.
- Pensez à sauvegarder régulièrement vos programmes.
- Méditez la loi de MURPHY :

En cas de risque de catastrophe, celle-ci se produit toujours au plus mauvais moment.

Qui n'a pas passé une partie de la nuit à retaper tout ou partie d'un programme ou d'une documentation perdue à la suite d'une mauvaise manipulation ou d'un incident matériel ?

- Appliquez ce précepte :

Un bon informaticien est celui qui :

- dit ce qu'il va faire,

- fait ce qu'il a dit,

- dit ce qu'il a fait.

Alain VAN SANTE
Lycée Gaston Berger - Lille