Chapitre 1

Génie Logiciel

1.1 Le quoi et le comment

Comme nous l'avons mentionné dans l'introduction, pour des raisons de portabilité, d'efficacité, et plus généralement de culture, le programmeur doit souvent utiliser un langage de programmation généraliste qui convient rarement aux spécificités de son application. Très philosophiquement, mais très schématiquement, il y a deux manières d'aborder le problème de la programmation, en privilégiant *« le quoi ? »* ou le *« le comment ? »* .

- *« le quoi ? »* conduit à s'intéresser au processus de modélisation qui donnera naissance au programme. C'est là le vrai travail de l'ingénieur et c'est cette question qui a donné naissance au *génie logiciel*. Nous parlerons alors – en tout cas ici – d'*abstraction de données*, de *programmation par objets*, ou encore de *modèles de programmation et d'exécution* et pourquoi pas, mais cela dépasse quelque peu le cadre de cet ouvrage, d'*applications distribuées* et de *composants logiciels* ;
- *« le comment ? »* nous amène à privilégier la réalisation et le codage de l'application. Nous nous intéresserons alors au langage pour lui-même, à sa syntaxe, à la sémantique de ses constructions, à son efficacité. C'est le but des parties suivantes de ce livre où nous présentons le langage C++, néanmoins nous essaierons de garder toujours les aspects méthodologiques – *le quoi ?* – en toile de fond. Un autre aspect technique de la programmation, qui sort aussi du cadre de cet ouvrage, est l'utilisation des outils qui permettent d'optimiser l'application en tirant parti des caractéristiques architecturales des calculateurs : *calcul vectoriel* ou *superscalaire* et les différents niveaux de *parallélisme*.

Le but de cette première partie est de donner quelques idées sur *« le quoi ? »* , utilisables avec profit pour le développement de logiciels dans des langages à objets, typiquement en C++ mais pas uniquement. Nous commencerons par quelques notions

et techniques de génie logiciel en incluant les idées de base de la programmation par objets : *spécification*, *conception descendante* ou *ascendante* et les différentes formes de *généricité*. Nous présenterons ensuite les mécanismes fondamentaux que nous allons retrouver dans les langages de programmation orientés objet.

1.2 Cycle de vie

Nous appellerons *cycle de vie* les différentes phases de la réalisation puis de l'exploitation d'un logiciel. Cette notion n'existe pas uniquement dans le monde des objets, néanmoins le cycle de vie prend tout son sens en support de la méthodologie objet et nous allons en détailler les différentes étapes.

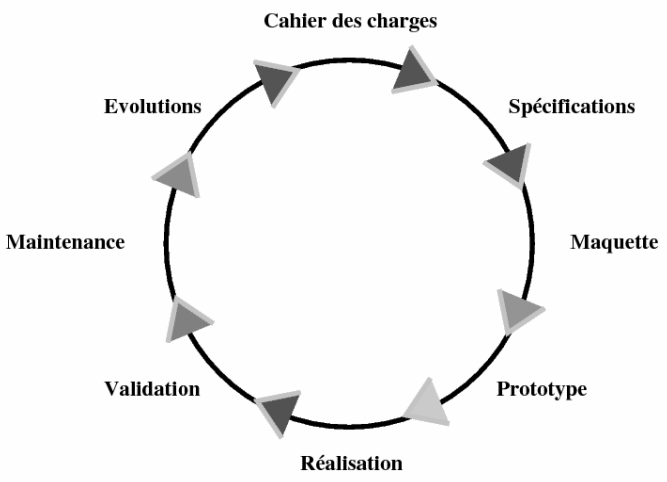

FIGURE 1.1: Le cycle de vie du logiciel

Cahier des charges Il faut dresser au préalable un cahier des charges qui doit décrire précisément les fonctionnalités du logiciel, les entités manipulées par les utilisateurs et les travaux qu'ils pourront effectuer. L'art d'établir un cahier des charges n'entre pas dans le cadre de cet ouvrage, néanmoins il semble important d'attirer l'attention sur le fait que le cahier des charges doit décrire ce que fait le logiciel et non comment il doit être réalisé. Le plus souvent, les rédacteurs ont *une idée de la façon dont cela va être fait* et, s'en inspirant, détournent quelque peu l'esprit du projet. De plus il est rare que le cahier des charges « prévoit tout » et il est préférable au niveau de la conception d'essayer d'anticiper si possible les évolutions du logiciel. Enfin ce cahier des charges est bien une sorte de contrat entre un client – le futur utilisateur – et

un prestataire – le développeur. Il ne faut donc pas hésiter à demander des précisions, voire même à demander au « client » de revoir sa copie.

Spécification À partir du cahier des charges, la spécification consiste à modéliser toutes les entités manipulées par l'application, leurs relations mutuelles, les opérations dans lesquelles elles interviennent et les transformations qu'elles subissent. Ce qui sera accessible aux utilisateurs (issu du cahier des charges) n'est, on s'en doute, que la partie émergée de cet iceberg.

Maquette La maquette est une ébauche de réalisation qui représente l'architecture du produit final et permet de tester la cohérence des spécifications avant de passer aux phases de codage proprement dites. Un exemple trivial serait la conception d'une interface graphique. Dans une maquette, l'utilisateur aurait accès aux menus et options sans que les fonctionnalités soient codées « par derrière », mais il pourra vérifier qu'*a priori* le système répondra bien à ses besoins. Plus précisément, la question que l'on se pose à cette étape est la suivante : *les propriétés des entités manipulées par le logiciel -et spécifiées à l'étape précédente- permettent-elles vraiment de décrire en termes algorithmiques les fonctionnalités du cahier des charges ?* Si la réponse est oui notre système est correctement spécifié.

Prototype À ce stade, nous pouvons commencer à coder. Mais avant de se lancer à corps perdu, il est bon de penser prototypage. En effet, il est souvent prudent de vérifier le bien fondé de certains choix de réalisation, découpage fonctionnel, implantation des données, *etc.* pour s'assurer que l'application pourra fonctionner correctement. Il faut aussi et surtout tester l'adéquation avec l'architecture matérielle qui sera utilisée : *les données tiendront-elles dans la mémoire de la machine ? le délai de restitution des résultat sera-t-il acceptable ?* En résumé, le prototype est une partie du produit final, il permet de valider des choix de réalisation avant de bâtir tout l'édifice.

Réalisation C'est seulement maintenant que nous pouvons aborder la réalisation proprement dite. Si le codage de l'application représente bien la plus grande partie du travail à réaliser, nous voyons bien ce n'est pas la première priorité. La conception « incrémentale » des programmes est, nous le comprendrons rapidement, difficilement compatible avec l'approche objet mais souvent, hélas, largement pratiquée par les programmeurs `Fortran` ou `C`. Ainsi « faire grossir » un petit prototype pour qu'il devienne un code est une démarche incompatible avec des objectifs de qualité.

Validation Avant de laisser les utilisateurs s'emparer du logiciel, il faut le valider c'est à dire leur donner quelques garanties. *L'adéquation avec le cahier des charges*

initial est-elle parfaite ? le logiciel est-il fiable ? donne-t-il des résultats corrects ? peut-on préciser son domaine de validité ? c'est à dire détecter des données d'entrée formellement correctes mais logiquement non valides. Pour cela et selon le degré d'exigence du cahier des charges, nous devrons fournir des jeux de tests ou un dossier de certification. Des procédures d'assurance qualité peuvent aussi être mises en œuvre contractuellement ou non tout au long du cycle de vie. Enfin, à ce stade, il faut finaliser les *documentations* – l'une interne, l'autre destinée aux utilisateurs – dont l'établissement et l'enrichissement est aussi une tâche « transversale » à toutes les étapes du cycle de vie.

Maintenance Tout cela fonctionne et a été mis sur le marché ; nous entrons maintenant dans la phase d'*exploitation* du logiciel. Il y a des utilisateurs et nous serons confrontés à la maintenance qui inclut :

- les optimisations, souvent nécessaires malgré les prototypes ;
- les problèmes variés liés à l'hétérogénéité des environnements informatiques : utilisation d'autres compilateurs, portage de l'application sur d'autres systèmes d'exploitation, d'autres architectures matérielles ;
- et d'une façon générale la corrections des erreurs, le suivi des mises à jour, la validation de la compatibilité des versions successives, *etc.*

Évolutions Enfin, il est peu probable que le logiciel reste figé. Il faut dès le début se préparer à y intégrer par la suite des évolutions hélas rarement prévisibles. Par exemple, des « oublis » du cahier des charges, des extensions, de nouvelles fonctionnalités. Là, la documentation interne prend toute son importance.

Il s'agit bien d'un cycle, les évolutions nous conduirons à revoir le cahier des charges initial, puis sans doute les spécifications. Il faudra de nouveau maquetter et prototyper, valider de nouveaux choix de réalisation, effectuer les maintenances, *etc.*

1.3 Spécifier

Pour concevoir un logiciel, il faut au préalable choisir une méthode pour le diviser en un certain nombre de parties « homogènes » : nous parlerons souvent de *modules* et... de *modularité*. Il est facile d'imaginer que la méthode choisie sera souvent induite par les outils utilisés, par exemple – justement – par le langage de programmation, et que cela conditionnera tout le cycle de vie du logiciel tel qu'il est présenté ci-dessus. C'est pourquoi nous opposerons souvent dans ce qui suit l'approche « procédurale »– comprendre induite par l'utilisation de langages procéduraux – à l'approche orientée objet, préalable à l'utilisation des langages à objets.

1.3.1 Approche procédurale

La démarche traditionnelle, liée donc aux langages procéduraux, consiste à décomposer le traitement à effectuer en *tâches élémentaires*. Cette approche se perpétue, notamment dans le milieu du calcul scientifique, par la large diffusion du langage `Fortran` (mais aussi du langage `C`) où ces tâches sont implantées comme des sous-programmes. Dans ce cadre, ce qui influe de façon déterminante sur la conception des logiciels c'est la séparation *a priori* entre les données et les traitements. Dans l'approche procédurale le logiciel comprend deux parties finalement disjointes et de nature très différentes :

– une partie *statique*, la description de l'implantation des données du problème ;
– une partie *dynamique*, les procédures qui manipulent ces données.

Ainsi, tout concepteur de code « procédural » commence-t-il généralement par décrire des *structures de données* dont la remise en cause risque de faire écrouler tout l'édifice, si elles ne restent à jamais figées... La figure 1.2 donne une vision pessimiste mais réaliste d'un tel logiciel : tout repose sur les données dont rien ne garantit la cohérence car les fonctions sont conçues indépendamment les unes des autres.

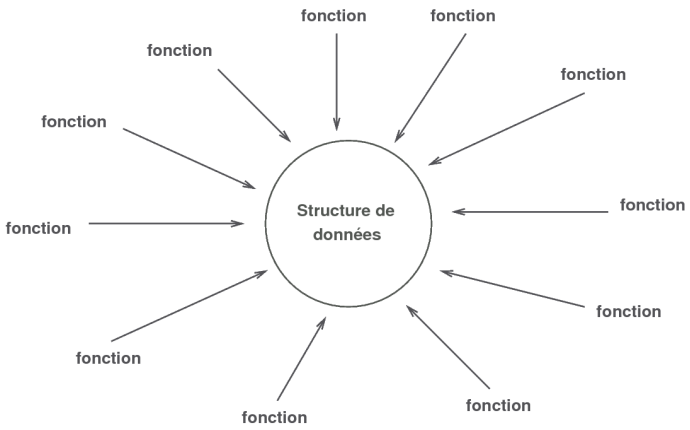

FIGURE 1.2: Un logiciel « procédural »

L'expérience des langages procéduraux montre, malheureusement, que l'on a du mal, avec cette méthode, à concevoir, valider, exploiter et maintenir des applications de grande taille. Pour ces raisons, un langage comme `Fortran` ou `C` n'est jamais employé seul :

– en développement, on utilise souvent des extensions au langage ou des systèmes de directives gérées par des *précompilateurs* ainsi que de nombreuses bibliothèques qui pallient les insuffisances du langage ;

– en validation, il est nécessaire de disposer d'*analyseurs statiques* qui travaillent sur le code source, par exemple détection des « branches mortes » qui ne seront jamais exécutées et d' *analyseurs dynamiques* pour valider le comportement à l'exécution de codes « instrumentés » pour le suivi, par exemple, des allocations mémoire. La gestion des erreurs, n'est pas intégrée aux langages et se révèle très complexe à mettre en œuvre.

L'utilisation du langage `C` ou des versions réputées « évoluées » de `Fortran` complique encore le problème. Par exemple, l'utilisation de *pointeurs* rend toute analyse statique approfondie impossible car les adresses en mémoire ne sont pas déterminées à la compilation et la preuve de l'utilisation correcte de la mémoire au fil des allocations et désallocations successives ne peut se faire qu'en utilisant des logiciels spécialisés qui travaillent sur des codes instrumentés.

À travers tous ces outils et toutes ces pratiques on recherche, bien sûr, l'amélioration de la qualité des logiciels, sans pour autant abandonner les bons côtés – l'efficacité, la normalisation – des langages de programmation les plus répandus, donc les plus anciens. Mais il y a d'autres raisons de ne pas changer de langage et donc de méthodologie. Les codes « patrimoine » représentent plusieurs années de travail d'équipes parfois nombreuses et ne peuvent être abandonnés du jour au lendemain. Le changement de méthode de conception implique aussi un changement d'organisation dans le travail d'une équipe : redéfinition des rôles et des responsabilités dans le processus de développement. Faire le « pari de l'objet » pour un développement *ex nihilo* est évidemment le bon choix, on s'en convaincra en lisant ce qui suit ! En revanche, changer de méthode nécessite une étude approfondie du retour sur investissement à long terme – réflexion rarement menée à terme – et implique de bloquer les développements jusqu'à ce que le nouveau code soit « au niveau » de l'ancien.

1.3.2 Des types abstraits à l'objet

L'approche *types abstraits de données* propose lors de la conception d'un logiciel de ne plus s'intéresser en priorité aux tâches ou aux fonctionnalités du logiciel mais à un critère *dual* du précédent, les *entités* qui seront manipulées par l'application.

En effet, si dans les langages traditionnels, nous disposons d'un nombre figé et très limité de types de données, `entier, réel, booléen, caractère`, ..., les langages orientés objet vont fournir des mécanismes pour définir, manipuler et combiner de nouveaux types de données à travers une construction généralement appelée *classe*. Ainsi, pour construire une « interface » utilisateur sur un écran, nous pourrons spécifier puis programmer les types `écran`, `fenêtre`, `menu`, ... et si nous travaillons avec des figures géométriques, les types `point`, `arête`, `polyèdre`, ...

Mais alors comment faire pour définir un nouveau type, la première question à se poser est très simple, c'est *qu'est ce que nous voulons faire avec ?* C'est à dire réfléchir aux informations auxquelles nous voudrons accéder et aux opérations spé-

cifiques que nous voudrons lui appliquer. Le point clef ici est que la description du type abstrait doit se faire indépendamment de tout problème de représentation de l'information associée ou de codage dans un langage de programmation donné. Ce qui est important, c'est ce que nous pouvons faire avec les variables de ce nouveau type, encore abstrait, et non pas leur structure. C'est bien pour cela que nous parlerons d'*abstraction de données*.

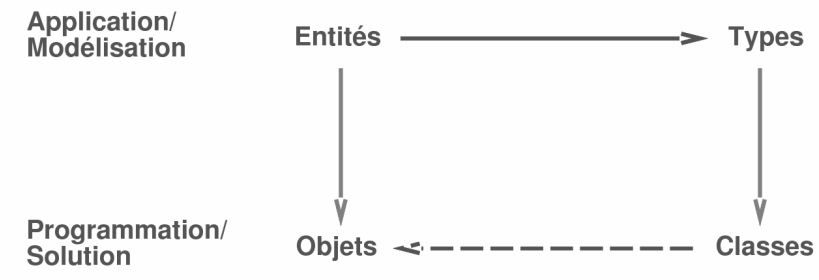

FIGURE 1.3: Des types abstraits à l'objet

Ce que nous manipulerons dans les applications, ce sont des instances de la classe – des exemplaires du type – que nous appellerons des objets. Ces objets sont des choses bien concrètes tout comme les variables réelles ou entières habituellement déclarées au début des programmes. Nous notons immédiatement que ces objets incluent les aspects statiques – accès à de l'information – et dynamiques – le comportement – et qu'il n'y a plus comme précédemment opposition dans le logiciel entre données et traitements : les données sont *encapsulées* dans les objets.

Revenons à notre question initiale *qu'est ce que nous voulons faire avec ?* La réponse prendra la forme d'une description fonctionnelle. Ces fonctions seront appelées par la suite les *méthodes*. Prenons quelques exemples simples, pour une structure « conteneur » comme la `Pile`, nous trouverons probablement : empiler ou dépiler une donnée ; tester si la pile est vide ; connaître le nombre de données empilées (la profondeur de la pile). Pour manipuler une instance d'un type `Vecteur`, nous aurons besoin de connaître sa dimension, d'accéder à ses éléments, d'effectuer des opérations algébriques, ...

L'approche *types abstraits de données* constitue un mode de description très riche, beaucoup plus proche de l'idée de modélisation que le découpage en tâches élémentaires. En effet, tout logiciel est une *implémentation* d'un modèle qui représente un système (physique, économique, *etc.*) et tout modèle décrit des entités et les relations entre ces entités. Pour un domaine d'application donné, les entités pourront être spécifiées sous forme de types abstraits de données puis codées une fois

pour toutes sous forme de classes. Résoudre un problème précis consistera donc à formaliser les relations entre des entités puis à les coder en manipulant les objets correspondants.

```
TYPE: Pile      -famille d'éléments "empilables"

Fonctions:
pile_vide:      Pile ⟶ Booleen
profondeur:     Pile ⟶ Entier
empiler:        Pile X Element_empilable ⟶ Pile
dépiler:        Pile ⟶ Element_empilable
```

```
TYPE: Vecteur   -famille d'éléments

Fonctions:
dimension:      Vecteur ⟶ Entier
élément:        Vecteur X Entier ⟶ Element
addition:       Vecteur X Vecteur ⟶ Vecteur
...             ...
```

FIGURE 1.4: Exemples de types abstraits de données

Un des chevaux de bataille de l'approche objet qui apparaît bien ici comme très générale, est justement la *réutilisabilité*. En revanche nous voyons bien que le logiciel ne pourra pas être issu d'une petite application qui évoluera peu à peu (la conception incrémentale évoquée plus haut). Il faudra décrire et réaliser toutes les entités du modèle avant de pouvoir écrire une application même « simple » qui manipule des objets d'où l'importance primordiale de la phase de spécification dont l'adéquation avec le cahier des charges doit être validée.

La spécification des nouveaux types sera d'autant plus facile que nous disposerons d'une théorie sous-jacente (mathématique, physique, ...). Par exemple pour spécifier un type `Vecteur` nous nous attacherons à disposer des mêmes propriétés que l'entité vecteur qui intervient dans les spécifications du logiciel. Nous ne nous préoccuperons pas dans cette phase de la représentation informatique qu'il faudra adopter (tableau, *etc.*). S'interroger sur le statut *in fine* des objets peut aussi aider à leur dé-