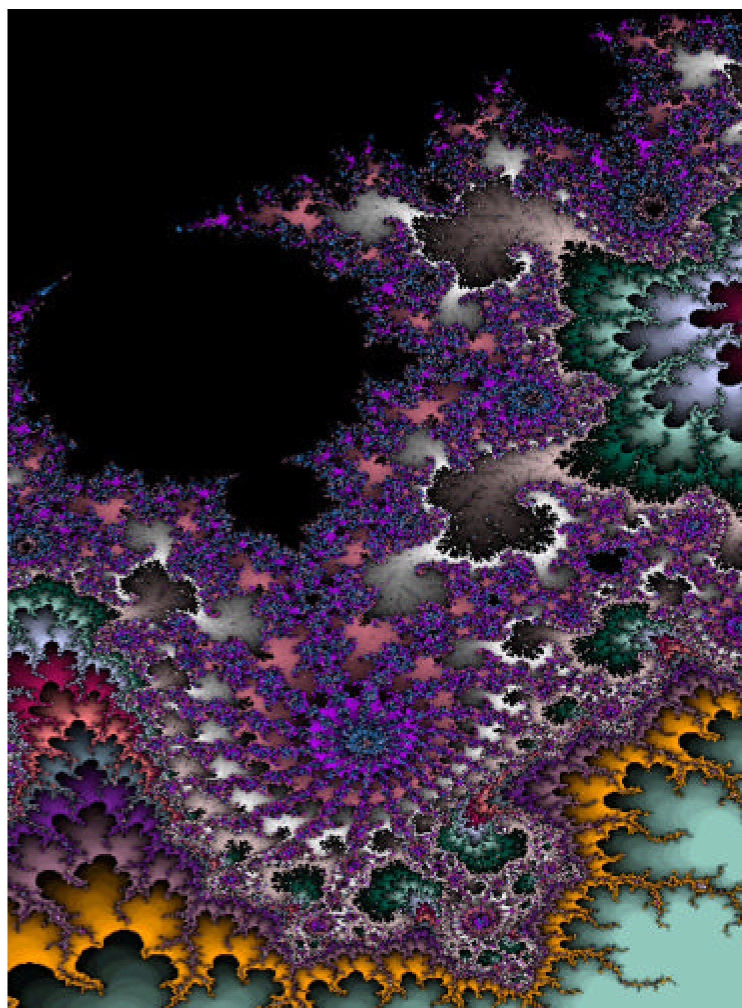
CHAPITRE 5

Analyse et conception orientées objets



5.1 *Motivations*

Bien avant d'écrire un programme, il faut en définir les composantes et les fonctionnalités. Cette définition passe par plusieurs phases, qui existent aussi bien dans un paradigme orienté objets que dans un paradigme conventionnel. En revanche, les deux philosophies considèrent les choses sous un angle radicalement différent. Alors que dans un cadre conventionnel, on analyse un sous-système par rapport à un ensemble de fonctions, et que l'on définit des algorithmes auxquels on appliquera ultérieurement les données, dans un environnement orienté objets, on définit le catalogue d'objets que comprend le système. Une fois ce catalogue défini, on examine quelles relations doivent exister entre les objets, et quels services doivent offrir chacun de ces objets.

Gardons à l’esprit qu’une conception orientée objets correspond bien plus étroitement à la réalité que les conceptions de type traditionnel : un objet a un cycle de vie “naturel”, il va donc naître, vivre, - et au cours de sa vie, entretenir des relations diverses avec d’autres objets -, puis mourir. La modélisation objets tend à décrire aussi précisément que possible ce cycle de vie.

Il faut insister sur le fait que la conception orientée objets n’est pas une alternative à une conception algorithmique : cette méthode transcende l’algorithmique et offre des outils plus performants pour décrire un système. La décomposition algorithmique ou fonctionnelle tend à s’appliquer à un niveau de détail plus fin, lorsqu’il s’agit finalement de coder des détails du système. La décomposition algorithmique est relativement peu importante, sauf dans les phases de codage et d’optimisation de performances.

5.2 *Aspects de la modélisation*

On définit en principe trois aspects dans une modélisation orientée objets :

- Statique : c'est un aspect orienté vers les objets dans le système, et leurs relations mutuelles. Cet aspect peut être décrit à l'aide d'un diagramme d'objets, où les noeuds sont les objets, et les liaisons entre noeuds les relations entre les objets.
- Fonctionnel : il s'agit de la description de l'évolution des données au travers du programme. Un modèle fonctionnel est défini généralement par un diagramme de flux de données.
- Dynamique : C'est la description des aspects du système qui se modifient au cours du temps. Le modèle dynamique contient des diagrammes d'état, dans lequel les noeuds sont des états, et les liaisons des transitions d'un état vers un autre. Les transitions sont induites par des évènements.

Ces divers aspects peuvent se traduire en vues différentes; un même aspect peut aussi donner naissance à des vues différentes dans certains cas.

5.3 *Quelques règles de base*

Quelle que soit la méthode utilisée lors de la définition du modèle, il y a certaines règles de base communes à tout design qu'il est nécessaire de respecter. La première est la définition des relations entre les objets. Ces relations peuvent être de plusieurs types, à la base :

- La **contenance**, un objet contient un autre objet. Par exemple, une fenêtre affichant un message informatif dans un système à multi-fenêtrage (Open Look, MOTIF, Mac, Windows) contient des boutons Help, Ok, Cancel, etc... Ce type de relation est également appelé relation de type "**has a**". La relation inverse est souvent notée ”**is part of**”. Ce type de relation est aussi appelé **aggrégation**.
- L'héritage d'**implémentation**. On exprime par là que l'objet descendant (qui hérite le comportement) est implémenté sous forme de l'objet ancêtre. Il ne s'agit que d'une relation d'implémentation. Ainsi, une liste de personnes peut être implémentée à l'aide d'un tableau de dimension variable, d'une liste, voire même à l'aide d'ensembles. Il s'agit d'un **détail d'implémentation**. Le fait que ma liste soit implémentée d'une manière ou d'une autre n'a pas d'incidences sur le contenu informatif et fonctionnel de ma liste. Par contre, le choix peut s'avérer important dans l'optique de la réutilisation de code, ou de facilité de l'implémentation de certaines fonctionnalités (p. ex. accès direct aux personnes par hachage, par recherche binaire, etc...) Ce type de relation est souvent appelé relation de type “**is implemented in terms of**”. De nombreux auteurs déconseillent l’utilisation de ce type de relation, préférant lui substituer la contenance; il est toutefois possible que certains détails d’implémentation du langage utilisé imposent l’utilisation de l’héritage d’implémentation dans un problème donné (utilisation de membres protégés en C++, par exemple). Certains langages ne permettent pas de définir une telle forme d’héritage (par exemple Java).
- L'héritage de **définition**. Cette relation exprime que l'objet descendant **EST** un objet ancêtre, avec une certaine **spécialisation**. Ainsi, pour reprendre ma liste de personnes, je peux définir, à partir de l'objet “Personne”, les spécialisations suivantes : “Personne amie”, “Relation d'affaires”, etc.. La personne amie aura une adresse, comme la personne de base, mais peut-être la complétera-t-on par une date de naissance/anniversaire, le nom des enfants, etc... La relation d'affaires aura pour attribut complémentaire sa raison sociale. Mais les deux objets sont également des personnes, et tout ce qui est valable pour une personne est également valable pour ces deux spécialisations. On appelle fréquemment ce type de relation une relation de type “**is a**” ou ”**is kind of**”.
- La **connaissance mutuelle ou non**. Cette relation implique que deux objets se connaissnet, ou encore que l’un connaît l’autre; ceci peut se traduire par une référence ou un pointeur sur l’autre objet.

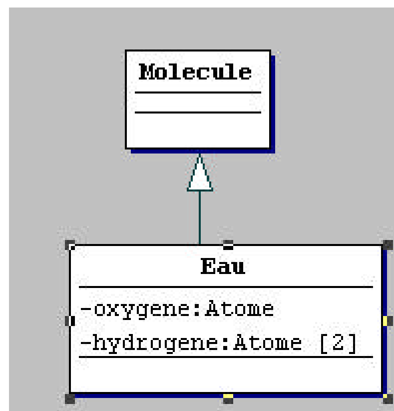
  L’héritage et l’aggrégation sont le plus souvent combinés; ainsi, la figure5.1, page55 montre qu’une molécule d’eau est une spécialisation (héritage) d’une molécule générique, et se compose de trois atomes, deux d’hydrogène et un d’oxygène. Le code implémentant ce modèle en C++ est donné ci-après :

```
// Generated by Together
#ifndef MOLECULE_H
#define MOLECULE_H
class Molecule {
```

```
};
#endif //MOLECULE_H

#ifndef EAU_H
#define EAU_H
#include "Molecule.h"
class Eau : public Molecule {
private:
  Atome oxygene;
  Atome hydrogene[2];
};
#endif //EAU_H
```

FIGURE 5.1 Héritage et aggrégation combinés



L'héritage de définition est une notion très importante en programmation orientée objets, et c'est aussi une notion controversée. Les notions que sous-entend ce type de relation sont multiples et parfois trop complexes pour être immédiatement détectées. Ainsi, hériter d'une définition implique que l'on hérite également des messages qui sont liés à cette définition. Cet héritage de messages va se traduire, dans l'implémentation, par un héritage de méthodes qu'il faudra soit

- reprendre telles quelles (en admettant que le descendant puisse se satisfaire de l'implémentation utilisée pour son ancêtre)

- spécialiser (si le descendant nécessite un comportement spécifique de la méthode)

Cet héritage implique que l'on puisse surcharger une méthode, c'est-à-dire redéfinir son comportement sans en changer la définition. Surcharger une fonction a souvent pour corollaire que l'on peut également surcharger un opérateur, du fait de la relation intime existant entre une fonction et un opérateur.

L'héritage peut être multiple. Ceci signifie qu'un objet hérite deux définitions simultanément. Il EST donc (**is a**) les deux objets tout à la fois. Ainsi, un véhicule à moteur peut être une automobile, ou un bateau. On peut décider, dans le cadre d'un modèle, qu'un véhicule

amphibie EST à la fois un véhicule automobile et un bateau, auquel cas l'héritage multiple est approprié. Néanmoins, l'héritage multiple est une notion parfois délicate à manipuler. L'héritage multiple n'est en fait que rarement nécessaire, et s'il l'est, il s'impose de lui-même.

Ensuite, il y a les communications entre les objets, qui sont des dépendances indirectes. Dans un système multi-fenêtre, l'existence d'une fenêtre est liée à certaines conditions, comme le démarrage d'une application, la pression d'un bouton dans une autre fenêtre réclamant de l'aide. Il y a là une relation de type cause / conséquence. Remarquons qu'une cause n'a pas forcément qu'une seule conséquence, mais peut en avoir plusieurs, ou aucune. Tout dépend de l'état du système à un instant donné. On voit d'emblée que la seule définition statique ne suffit pas, il faut encore modéliser le comportement au cours du temps.

Certains liens de communication, toutefois, sont statiques, comme la notion d'**association** entre deux instances. Mr Dupont travaille pour la société ACME. Il n'y a là aucun lien informatique entre les objets, mais bien un lien du point de vue de la signification du modèle. Ce type de lien (on parle souvent d'association, ou de relation de type ”**has knowledge of**”) peut être unique, mais aussi multiple. En pratique, on implémente souvent une association de ce type par un pointeur d'un objet sur l'autre. Cette manière d'implémenter ne correspond que partiellement à la réalité. Le pointeur tend à sous-entendre que l'association est partie intégrante de l'objet qui inclut ce pointeur, alors qu'en réalité, l'association est un concept totalement indépendant des entités concernées. Il ne s'agit pas forcément d'un détail d’implémentation, car un pointeur ne permet de figurer l'association que dans un sens. Cette restriction d’implémentation peut être ou ne pas être significative : il se peut que le problème ne nécessite pas d'association bidirectionnelle, ou du moins que l'association ne doive jamais être faite dans l’autre sens.

Enfin, un objet peut constituer une partie d’un autre (**is part of**) ou dépendre directement ou indirectement de l’existence d’un autre (**depends upon**). Il peut aussi ressembler suffisamment à un autre objet (**is analogous to**) pour que l’on puisse se demander si les deux objets ne devraient pas dériver d’une même classe.

Cette brève liste n’est pas forcément exhaustive, et certains types de relation peuvent être malaisés à associer avec l’une ou l’autre de ces catégories. Il est néanmoins intéressant de faire cette association, car elle permet ensuite de passer à l’étape ultérieure de la modélisation, qui ne sera plus seulement descriptive, mais qui devra déboucher sur une description formelle du modèle.

5.4 *Généralisation*

Dans un design orienté objets, le premier jet est en général inapproprié. Cela ne signifie pas que le premier jet est faux, mais qu'il ne correspond pas à un modèle optimal. Un modèle optimal est celui qui tire parti au maximum des potentialités du paradigme “orienté objets”. Ce modèle conduira à une implémentation qui permettra de réutiliser un maximum de composantes au cours de projets futurs.

Si je désire fabriquer une liste d'amis, je peux établir un modèle où figurent, dans chaque objets, les éléments qui m'intéressent, comme adresse privée, numéro de téléphone, nom des enfants, anniversaire, informations sur l'épouse, etc... Si plus tard je veux me fabriquer une liste de relations d'affaires, je ne peux pas, ou malaisément réutiliser le modèle défini précédemment, parce qu'il contient une pléthore d'informations inutiles. Il eût mieux valu définir tout d'abord un modèle de personne générique, et ensuite le spécialiser pour qu'il corresponde à mes besoins. Vraisemblablement, l'implémentation de ma liste d'amis eût été plus complexe, et eut nécessité plus de temps; mais lors de la phase suivante, le gain de temps eût largement compensé la perte initiale.

Cet effort de réflexion s'appelle la **généralisation**; il s'agit de définir, dans notre modèle, les composantes génériques et les spécialisations de ces composantes qui permettent l'implémentation du modèle. Il vaut toujours la peine de faire cet effort de réflexion, qui peut mettre en évidence des réutilisations possibles d'une fonctionnalité, et permettre de favoriser cette réutilisation par un empaquetage différent, ou par l'addition de fonctionnalités non directement nécessaires à l'implémentation actuelle, mais permettant la réutilisation plus tard. Par contre, il se peut que pour des raisons non techniques, on soit amené à faire des compromis lors de l'implémentation (délais, coût du produit original, certitude qu'il n'y aura pas de réutilisation possible dans le futur, etc...).

Enfin, admettons que le modèle réellement optimal est un objectif inatteignable, comme la perfection en toutes choses. Mais rien n’interdit d’essayer de s'en approcher. Il n'est d'ailleurs pas certain, et même peu probable que **le** modèle optimal existe. Il est plus vraisemblable qu'il existe une collection de modèles qui sont tous acceptables, et qui présentent tous des points forts et des inconvénients. Un même problème peut de ce fait conduire à des modélisations radicalement différentes, sans que l'une ou l’autre modélisation soit forcément plus faible que l'autre. Ce qui ne justifie de loin pas le fait de s'arrêter au premier modèle réalisé. Ce dernier est - sauf problème trivial ne requérant pas vraiment l'utilisation d'un modèle -, presque forcément insuffisant et immature. Le polissage d'un modèle est un travail de longue haleine, qui ne peut pas être le fruit du travail d'une seule personne. Il est nécessaire de recourir à de séances d'inspection du modèle, ou plusieurs personnes indépendantes, mais néanmoins au courant de la problématique, examinent le modèle de manière critique pour essayer d'en mettre en évidence les faiblesses. Cette inspection est ensuite discutée en groupe, et des améliorations sont proposées pour conduire à un peaufinage, ou éventuellement à un redesign complet. Le processus est pénible, mais il se traduit par une implémentation plus facile, et moins sujette à trébucher sur des problèmes reconnus tardivement. Même un effort limité dans ce sens en vaut la peine. Un modèle inapproprié conduit forcément à un design boiteux, et à de graves difficultés d'implémentation. Les efforts consentis dans la définition du modèle sont récompensés au centuple lors du codage.

5.5 *Formalisme*

Inspecter un modèle sous-entend que l'on dispose d'une manière plus ou moins standardisée de définir ledit modèle. Autrement, comment d'autres personnes pourraient-elles juger des faiblesses et établir une critique valable?

Il faut donc définir un formalisme avec lequel on définira le modèle. Idéalement, on se servira d'un formalisme défini comme un standard au sein de l'entreprise, ou du groupe de travail dans lequel le projet doit s'effectuer. Il existe plusieurs méthodes utilisables, mais une de ces méthodes s'est généralisée récemment, qui est UML. Dans la suite de ce cours, nous allons présenter en introduction une méthode très simple, qui est la méthode CRC, qui se prête relativement bien à des séances de brain-storming en groupes. Cette méthode permet de définir des classes élémentaires, mais décrit très mal les relations entre les diverses classes.

Dans la deuxième partie, nous allons décrire un peu plus en détail la méthode UML, qui n'est pas incompatible avec CRC, mais qui va beaucoup plus loin. Ainsi, on peut imaginer débuter un projet avec des cartes CRC, en rester là si le problème est suffisamment simple, ou passer à UML dans une deuxième phase. Dès que le problème devient complexe, il est avantageux de passer d'emblée à UML.

5.6 *Quelques règles à respecter lors de la normalisation*

Ne pas commencer une modélisation simplement en jetant quelques classes sur le papier et en les joignant par des liens d'association. Une modélisation sous-entend la compréhension du problème, et le contenu du modèle décrit en réalité la solution.

Le plus simple sera le modèle, au mieux il sera utilisable.

Nommez avec beaucoup de soins les éléments du modèle. Les noms convoient une quantité appréciable d'informations lors des inspections en groupe. Eviter à tout prix les risques d'ambiguïtés qui peuvent conduire à des pertes de temps et d'énergie considérables. Un nom devrait pouvoir se passer de tout commentaire.

Eviter à tout prix de lier le modèle à l'implémentation. Eviter d'utiliser un pointeur pour indiquer une association dans le modèle, même s'il est évident que la réalisation se fera ainsi. Utiliser explicitement la notion d'association en lieu et place, pour bien indiquer ce que vous désirez faire, et non comment vous allez le faire.

Au besoin, savoir violer la règle ci-dessus. Une généralisation excessive peut se révéler inappropriée. Un projet dont on sait pertinemment qu'il se déroulera sous DOS ne doit pas nécessairement tenir compte d'une hypothétique implémentation sous UNIX. Si il est possible de le faire à peu de frais, tant mieux. Mais il faut savoir faire la part des choses intimement liées à l'implémentation. Ainsi, tout ce qui concerne des aspects de communications inter-taches, d'aspects liés au système de gestion de données, de problèmes de concurrence dans un environnement "multi-threads" peut influencer le modèle dans le sens d'une implémentation dépendante du système d'exploitation. Vouloir modéliser ces concepts avec toute la généralité idéalement requise peut s'avérer coûteux, et compliquer inutilement le modèle.

KISS. Keep It Simple, Stupid. Brian Kernighan affirmait également "Small Is Beautiful". Actuellement, il serait bien emprunté de répéter cette affirmation en conjonction avec les implémentations modernes de UNIX (Solaris, HP-UX, OSF/1) qui occupent allégrement 900 MBytes en configuration minimale... Il n'en reste pas moins que les choses simples sont plus faciles à comprendre, donc à entretenir. En résumé, préférez rechercher une solution simple si la solution à laquelle vous êtes parvenus semble un tant soit peu compliquée. En particulier, on évitera les modèles faisant appel à des imbrications de classes trop complexes, ou à des classes avec un trop grand nombre de méthodes.

5.7 *Révisions en groupe*

Un modèle objets ne peut généralement pas être réalisé par une personne seule, sauf problème particulièrement trivial. Il faut donc essayer, aussi tôt que possible, d'impliquer des tierces personnes dans l'analyse d'un modèle. Ce type de révision permet de découvrir très tôt des erreurs fondamentales, et de souligner les probables causes de difficultés d'implémentation. Une intervention d'une tierce personne à ce stade du développement peut économiser des semaines de travail inutile lors de l'implémentation ou du test.

La révision en groupe est un concept relativement récent dans le développement de logiciels. On doit ce concept, sous sa forme la plus élaborée, à un certain Mr. **Fagan**, qui a défini un certain nombre de règles selon lesquelles une révision en groupe devait se dérouler pour être efficace. Un groupe de révision comprend en principe quatre personnes avec des rôles bien précis : le **lecteur** lit le texte à réviser, les **experts** émettent les critiques que leur inspire la lecture du document, et l'**auteur** éclaircit les points qui seraient restés obscurs à la lecture du document. En principe, cet exercice ne prétend pas trouver des solutions, mais des erreurs.

Cette méthode, quoique très fastidieuse, est de plus en plus utilisée dans le cadre de grands projets logiciels, car elle permet (en théorie du moins) de gagner du temps lors de l'implémentation et du test, selon le principe qui veut qu'une erreur tôt détectée est plus facile à éliminer qu'une erreur qui apparaît lors du test final. En pratique, la manière de conduire la révision peut faire toute la différence entre le succès ou l'échec de la méthode. Le rôle de lecteur est dans cette optique fondamental.

***Test**: Analyse et Conception OO*

A quoi correspond l'aspect statique de la modélisation ?

A quoi correspond l'aspect dynamique de la modélisation ?

A quoi correspond l'aspect fonctionnel de la modélisation ?

Citer les principales relations que l'on peut définir entre des objets

Définir l'héritage de définition et l'héritage d'implémentation

Quel est l'intérêt de se livrer à une généralisation ?

Quel buts poursuivent les examens en groupe ?