Développement de logiciel à objets avec UML

Jean-Marc Jézéquel, Noël Plouzeau, Yves Le Traon
IFSIC – Université de Rennes I

Septembre 2005 – *Revision* : 1.6

Preface

Ce document a pour objectif de présenter à un niveau assez basique une vision générale du développement de logiciels à objets fondée sur l'utilisation de modèles, acquise en une vingtaine d'années de recherche et de pratique des technologies objets. Sa structure s'appuie sur une quinzaine d'années d'expérience d'enseignement dans ce domaine. Pour en savoir plus, et obtenir notamment du matériel additionnel (copie de slides etc.), consulter http://www.irisa.fr/prive/jezequel et suivre le lien "enseignement".

Cette version (Révision – *Revision* : 1.6) se fonde sur la norme UML 1.4. Nous mentionerons cependant briévement les principaux traits de la nouvelle version de ce langage, UML 2.0, dont l'adoption devrait se faire dans l'industrie d'ici quelques années.

Nous tenons à remercier les nombreux relecteurs qui nous ont fait part de leurs commentaires, avec une mention spéciale pour Anne-Claire Guillou et Placide Fresnais.

Table des matières

Table des figures

Chapitre 1

Ingénierie de la réalisation des systèmes logiciels

1.1 Introduction

Pris dans un sens large, le génie logiciel englobe les «règles de l'art» de l'ingénierie de la réalisation des systèmes manipulant de l'information, qu'on appelle encore systèmes à logiciel prépondérant. Autrefois cantonné à des domaines limités comme l'informatique de gestion, le calcul scientifique, ou encore le contrôle d'engins high-tech, le logiciel est aujourd'hui omniprésent. Il y a, par exemple, plus de logiciel dans une automobile de moyenne gamme récente ou dans le moindre téléphone portable que dans une capsule Apollo.

Pour les systèmes de plus en plus complexes qu'on cherche à construire, il s'agit de trouver un compromis entre un système parfaitement bien conçu (qui pourrait demander un temps infini pour être construit) et un système trop vite fait (qu'il serait difficile de mettre au point et de maintenir), en conciliant trois forces largement antagonistes : les délais, les coûts, et la qualité. Les délais doivent être tenus pour que le projet aboutisse, la qualité doit être à un niveau compatible avec la criticité du système, et le coût doit rester en rapport avec les moyens financiers disponibles. Il ne s'agit donc pas de produire un «bon» système dans l'absolu, mais bien de produire, en suffisamment peu de temps, un système suffisamment bon et suffisamment bon-marché compte tenu du contexte dans lequel il sera utilisé.

La maîtrise des processus d'élaboration du logiciel se décline en trois axes :

- Programming-in-the-Small, c'est à dire la maîtrise de la construction d'éléments de logiciels corrects. Comme nous allons le voir au paragraphe 1.2.1, ce n'est pas aussi simple que cela en a l'air.
- Programming-in-the-Large, c'est à dire la maîtrise de la complexité structurelle

due à la taille des objets construits. Bien évidement, cette complexité n'est pas le propre de l'informatique : on retrouvera le même type de problèmes (et de solutions) partout où il faut gérer de grands projets, par exemple le génie civil.

– Programming-in-the-Variability, c'est à dire la maîtrise de la malléabilité du logiciel (l'aspect *soft* du *software*). Un système logiciel n'est en effet pas rigide, il évolue dans le temps (maintenance évolutive) et dans l'espace (variantes régionales ou fonctionnelles : c'est la notion de lignes de produits).

Avant de voir ce qu'apporte l'approche objet dans la réalisation des systèmes logiciels, il convient de bien comprendre la nature des problèmes posés sur ces trois axes. C'est l'objet de ce chapitre introductif. Le reste de cet ouvrage est ensuite découpé en deux grandes parties.

La première est composée des chapitres 2 à 6 où on trouvera une présentation des principes de l'approche objet avec UML (Unified Modeling Language), ainsi que des différents axes de modélisation en UML (statique, fonctionnel, dynamique, déploiement).

La seconde partie s'intéresse aux démarches de construction du logiciel avec UML, en abordant la problématique du cycle de vie du logiciel et la construction d'un modèle d'analyse (chapitre 7), la notion de design patterns (chapitre 8), la conception de logiciel avec les design patterns et UML (chapitre 9) et enfin la validation de logiciels conçus par objet (chapitre 10).

1.2 Programming-in-the-Small

1.2.1 La programmation, c'est facile!

Il semble que personne ne s'étonne de trouver sur les rayonnages des librairies de nombreux livres avec des titres accrocheurs du type *Teach Yourself X in 24 hours* ou *X for complete dummies* lorsque X est un langage de programmation ; alors qu'on a du mal à trouver l'équivalent lorsque X est quelque chose comme *Brain Surgery* ou *Nuclear Engineering*.

Il s'agit du paradoxe bien connu qui existe entre l'apparente facilité de la programmation d'une petite application et la difficulté de la construction de grands systèmes informatiques, difficulté qui avait donné naissance à la notion de «crise du logiciel» dès la fin des années 1960. Ce paradoxe est sans doute lié à la différence entre la simplicité apparente du texte d'un programme et la l'infinie complexité potentielle de son exécution. Sans même aller chercher jusqu'à Turing et les problèmes d'indécidabilité inhérent à l'informatique, on peut facilement exhiber [19] de très petits programmes ayant des comportements infiniment complexes, comme par exemple :

```
Entrer un nombre naturel n ;
Tant que n est différent de 1 faire
```

```
  Si n est pair alors n := n/2 sinon n := (3n+1)/2 ;
Fait.
```

Nous invitons le lecteur qui voudrait se faire une idée de la complexité sous-jacente de ce programme à essayer de prouver qu'il se termine *quelque soit le nombre qu'on lui donne en entrée.* De manière générale, la vérification formelle de programmes informatiques, même apparemment simples, peut se révéler difficile, voire impossible. Seule l'utilisation de techniques de construction systématique de programmes (fondées sur l'utilisation d'éléments de spécifications formelles) permet de s'affranchir dans une certaine mesure de ces difficultés.

1.2.2 Conception de boucles correctes avec les assertions

Les éléments de spécifications formelles servant à construire des fragments de programmes prouvables sont les notions de :

- Pré-conditions et Postconditions d'opérations, pour définir les domaines d'entrée des opérations et en spécifier le résultat, et qui seront présentés en détail plus loin.
- Invariants et variants de boucles, pour prouver leur correction (en deux étapes: correction partielle + correction totale).

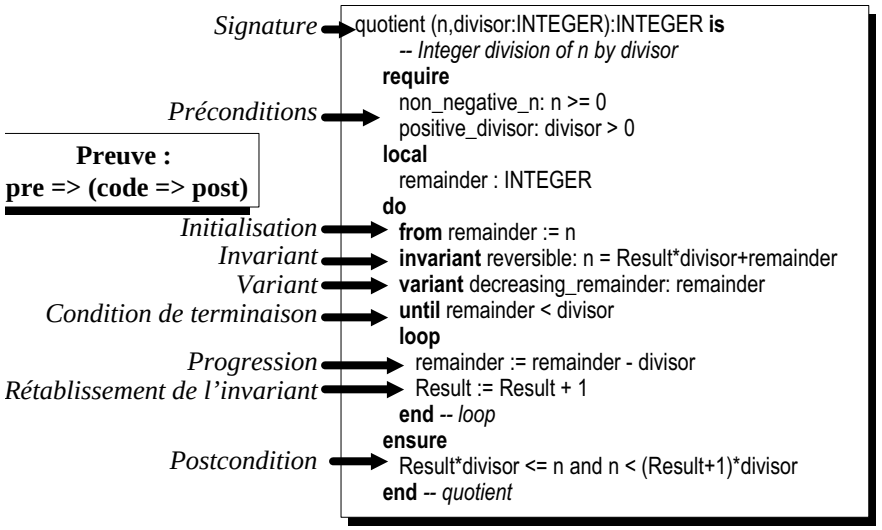


FIG. 1.1 – *Preuve de programme : exemple de la fonction « quotient » en langage Eiffel*

La notion d'invariant de boucle Un invariant de boucle caractérise ce que la boucle essaye de faire, sans décrire comment. C'est une expression booléenne qui doit être vraie

lors de l'initialisation des variables de la boucle, maintenue à chaque itération de boucle, et toujours vraie à la terminaison de boucle. Par exemple, un invariant de boucle dans la figure 1.1 pourrait être $n = Result \times divisor + remainder$.

Un invariant de boucle décrit des propriétés qui doivent rester vraies aux bornes de la boucle (initial, final, et états intermédiaires), ainsi il peut avoir un rôle ressemblant beaucoup à la notion d'hypothèse inductive employée en mathématiques. Il peut donc aussi jouer un rôle important pour dériver le corps de la boucle. Un invariant de boucle peut aussi être utilisé pour raisonner à propos de la correction de la boucle. À la fin de la boucle de la la figure 1.1, nous avons par exemple à la fois :

$$\text{invariant: } n = Result \times divisor + remainder$$

et

$$\text{termination condition: } remainder < divisor,$$

ce qui implique trivialement la postcondition. Donc, si on atteint la fin de la boucle, alors le résultat calculé ce sera bien celui que nous voulions. Cette propriété est appelée *correction partielle*. Voyons maintenant comment prouver la correction totale.

La notion de variant de boucle La correction totale est simplement la correction partielle *plus* la preuve que la boucle termine (c'est-à-dire que la condition de terminaison deviendra finalement vraie). L'idée de variant de boucle est de caractériser la façon dont chaque itération fait approcher la boucle de sa terminaison.

Un *variant* de boucle est une expression entière positive dont la valeur décroit strictement à chaque itération. Par définition, elle ne peut pas descendre en dessous de zéro, donc le nombre d'itérations d'une boucle munie d'un variant est borné et donc la boucle terminera obligatoirement. La valeur du reste dans notre exemple de la figure 1.1 est un variant qui convient parfaitement. En effet, ses valeurs forment une suite strictement décroissante d'entiers positifs.

Le variant de boucle ayant été trouvé, nous avons prouvé que la boucle se termine. Cette preuve complète la preuve de correction partielle donnée précédemment : en d'autres termes, la boucle termine et calcule le résultat attendu.

On peut donc construire des programmes de telle manière à ce qu'ils soient prouvables, et cette preuve est même en partie automatisable (Atelier B, etc.). Cependant ceci nécessite des outils spécifiques, assez coûteux à mettre en oeuvre. Or, la preuve étant au moins aussi complexe que le code, on a en fait autant de chances de se tromper dans la preuve que dans le code. Le pire étant bien sûr de se tromper à la fois dans le code et dans la preuve...

En pratique, ces techniques de preuve restent réservées à des (petits) sous-systèmes (très) critiques. L'approche prépondérante dans l'industrie reste le recours aux tests. Mais bien sûr, le test ne permet pas de prouver la correction de logiciels, il permet

simplement d'exhiber des fautes potentielles. Il faut en effet voir que si l'on voulait tester exhaustivement le petit programme mentionné au début de l'introduction (c'est-à-dire essayer ce programme pour toutes les valeurs d'entrée possible sur un ordinateur donné, soit pour un ordinateur 32 bits toutes les valeurs de 1 à 10^{31}) on aurait besoin de 10 milliards de cas de tests pour 5 lignes de code!

1.3 Programming-in-the-Large

1.3.1 Notion de complexité structurelle

Il apparaît donc qu'il est relativement simple d'écrire sur un coin de table un petit programme «marchant à peu près» (d'où le succès d'outils comme Visual Basic, Perl, Tcl/Tk, etc.), alors qu'il est relativement coûteux, et donc pas toujours rentable, d'essayer d'obtenir d'un logiciel, surtout quand il possède une nature répartie, des qualités de correction, fiabilité, robustesse, etc.

Pratiquement n'importe qui est capable de construire la passerelle de la figure 1.2 (a). De bons bricoleurs pourront aller plus loin (pont suspendu en (b)), mais déjà pour le petit pont en (c), cela demande plus de talent (et à l'échelle de l'humanité, cela a demandé quelques milliers d'années d'évolution). Quant au Golden Gate (en 1.2 (d)), sa réalisation a demandé d'autre talents que le bricolage. En bref, pas plus qu'un grand pont moderne n'est l'extrapolation d'une passerelle, ou un gratte-ciel celle d'une niche de chien bricolée, un grand logiciel ne s'obtient pas avec les mêmes méthodes qu'un petit programme.

1.3.2 Gérer la complexité structurelle due à la taille

Or, la taille des logiciels augmente de manière exponentielle : elle est multipliée par 10 tous les 10 ans parce que les systèmes qu'on développe sont de plus en plus complexes (voir figure 1.3).

> *Si l'automobile avait suivi le même développement que l'ordinateur, une Rolls-Royce coûterait aujourd'hui 100$, pourrait rouler un million de kilomètres avec un litre d'essence, et exploserait une fois par an en tuant tout le monde à bord.* Robert Cringely.

Même si en moyenne le sort des projets informatiques n'est pas particulièrement brillant (voir figure 1.4), chaque décennie, on a donc trouvé le moyen de gagner un ordre de grandeur sur la complexité de ce qu'on savait faire, ce qui comparé à d'autres domaines n'est pas si mal!

Pour maîtriser cette complexité, on a abordé le problème à la fois sur un plan technique et organisationnel.

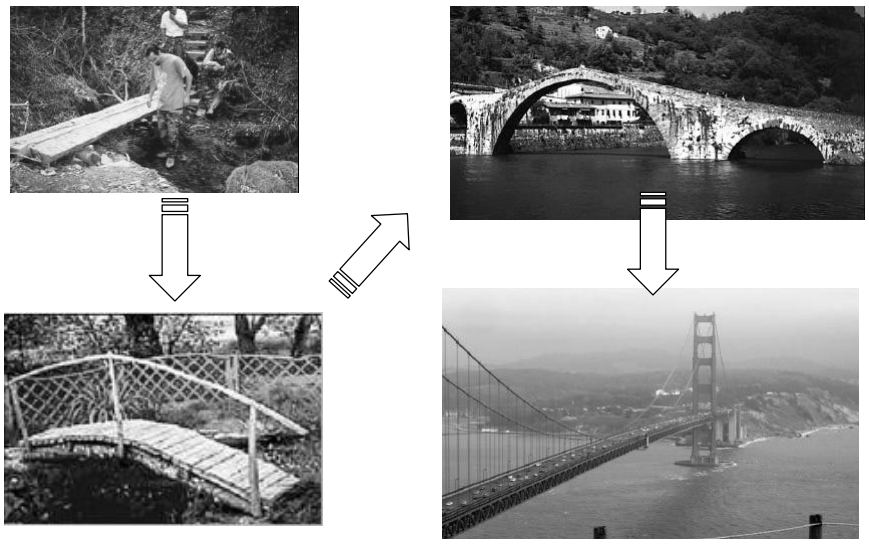


FIG. 1.2 – *Programming-in-the-Large : Gérer la complexité due à la taille*

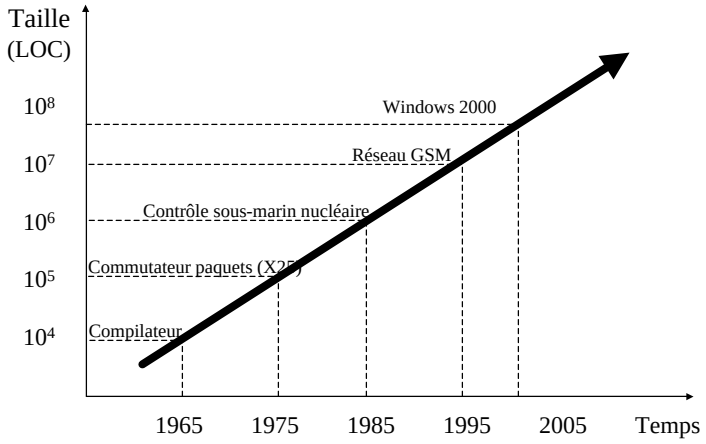


FIG. 1.3 – *Augmentation exponentielle de la taille du logiciel*

FIG. 1.4 – *Sort des projets informatiques*

Aspects techniques : en s'appuyant techniquement sur la modularité, c'est à dire les notions d'encapsulation et de masquage d'information, permettant d'avoir un couplage faible entre modules, qui peuvent ainsi être développés indépendamment.

Aspects organisationnels : où comment s'organiser au delà de la réalisation avec des méthodes d'analyse et de conception, et les notions de validation et vérification (voir figure 1.5).

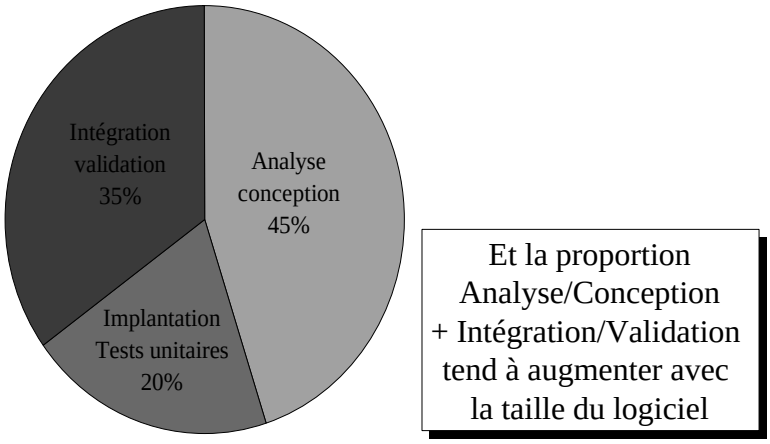


FIG. 1.5 – *Coûts du développement*

Les principaux problèmes se posent alors en termes de gestion de ressources (humaines, etc.) et de synchronisation entre tâches. C'est le domaine de l'ingénierie de la

conduite de projet, avec ses compromis entre respect des délais, respect des coûts et réponse aux besoins/assurance qualité.

Malgré son importance, ce domaine ne sera pas abordé plus avant ici, car il doit faire l'objet d'une attention spécifique. Dans la suite, nous nous concentrerons donc sur les aspects techniques du génie logiciel.

1.4 Programming-in-the-Variability

1.4.1 Malléabilité du logiciel

Même si le logiciel partage bien des problèmes avec d'autres domaines dans la gestion de la complexité structurelle, il a en revanche des spécificités quant à sa malléabilité. C'est d'abord un produit abstrait (papier, fichier, programme) qu'il est matériellement très simple de faire évoluer. Déplacer quelques lignes de programme est beaucoup plus facile que de déplacer un pilier de pont ou d'ajouter un radiateur dans une maison ; par contre l'impact est très certainement moins bien maîtrisé ! D'autre part, on rencontre fréquemment une certaine difficulté à spécifier les besoins. L'automatisation apportée par l'informatique force à modéliser/comprendre des processus de fonctionnement et d'organisation de systèmes complexes (avion, entreprise, etc.) dont il est très difficile d'obtenir une vision globale (complète) et cohérente. De plus, un logiciel doit évoluer pour conserver son utilité : c'est la problématique de la maintenance corrective et évolutive. L'automatisation fait en outre souvent apparaître de nouveaux besoins, de nouvelles possibilités et des défauts qui n'avaient pu être envisagés avant l'automatisation. Les coûts élevés des logiciels en font des investissements lourds qui ne sont supportables que si leur durée de vie est longue. D'après Swanson & Beath (Maintaining Information Systems in Organizations, 1989), la durée de vie moyenne d'un système est de 6.6 ans, et 26% des systèmes sont âgés de plus de 10 ans.

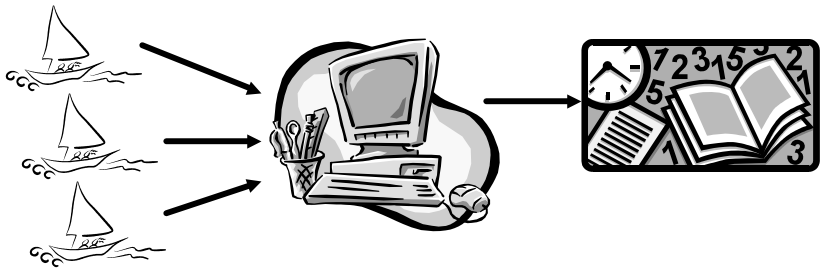
1.4.2 L'exemple des barques sur le lac de Jackson

Le problème de la maintenabilité des logiciels a explosé au début des années 80, au moment où la taille des logiciels faisait qu'aucun individu donné n'était plus capable de maîtriser l'ensemble d'un projet. Pour montrer qu'en fait ce problème était orthogonal à celui de la maîtrise de la taille du logiciel, M. Jackson [12] a construit un tout petit exemple possible mettant en évidence la source du problème (voir figure 1.6).

Il s'agit d'une entreprise qui propose des barques à louer sur un lac, et qui souhaitent réaliser l'enregistrement des départs (S) et retours (E) de barques sur un terminal, avec un horodatage automatique, afin d'obtenir chaque jour un rapport avec :

- le nombre de sessions
- la durée moyenne d'une session

- Barques à louer sur un lac
  - Enregistrement des départs (S) et retours (E) sur un terminal, avec horodatage automatique



- On veut chaque jour un rapport avec :
  - le nombre de sessions
  - la durée moyenne d'une session

FIG. 1.6 – *Problème de la maintenabilité*

Selon l'approche « droit au but » utilisant une décomposition fonctionnelle en vogue dans les années 1980 (avec par exemple Structured Analysis and Design Technique : SADT), on considère que le système a une fonction, qui doit être décomposée récursivement en sous-fonctions (reliées par flots de données), jusqu'à tomber sur des fonctions suffisamment simples pour pouvoir être implantées directement. Il faut remarquer que les langages de programmation de cette époque (cf. Pascal) possèdent cette même structure hiérarchique, ce qu'il fait qu'ils sont en très bonne adéquation avec une analyse et conception fonctionnelle descendante. Il est hors de doute que cette approche modulaire et hiérarchique permet de gérer effectivement la complexité due à la taille (programming-in-the-large). Ce n'est donc pas ce point qu'on regarde ici, et une solution simple peut directement être mise en oeuvre en utilisant un compteur totalisant les départs de barques d'une part, et une variable accumulant les temps totaux (heure de retour moins heure de départ) d'autre part (voir figure 1.7).

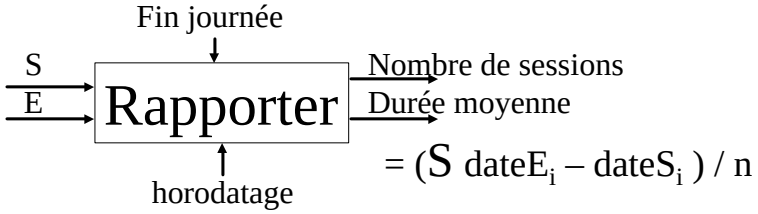


FIG. 1.7 – *Approche « droit au but »*

Ceci permet d'implanter le programme suivant, qui une fois mis en production rem-

plira parfaitement les fonctionnalités demandées.

```
begin
  open message stream;
  get message ;
  number := 0; totaltime := 0
  do while not end-of-stream
    if code = 'S' then
      number := number + 1 ;
      totaltime := totaltime - starttime ;
    else
      totaltime := totaltime + endtime ;
    endif
    get message ;
  enddo
  print 'NUMBER OF SESSIONS = ', number ;
  if number / = 0 then
    print 'AVERAGE SESSION TIME= ', totaltime / number ;
  endif
  close message stream ;
end
```

Le client est donc tout à fait satisfait de ce logiciel. Cependant, inévitablement ses besoins vont changer, ou, du moins, l'utilisation quotidienne du logiciel va lui donner de nouvelles idées sur les fonctionnalités qui seraient désirables. Il revient donc vers son fournisseur avec successivement les demandes de modifications suivantes. Il voudrait maintenant que le logiciel lui donne

1. Nombre de sessions depuis le début de la journée
2. Durée de la session la plus longue
3. Un état pour le matin, un autre pour le soir
4. La possibilité de corriger la perte de S ou de E

De son point de vue, il s'agit à chaque fois d'un tout petit changement vis-à-vis du cahier des charges initial. Du point de vue du fournisseur, il en va tout autrement. En effet, alors que le premier point est relativement simple à mettre en œuvre (il suffit de rajouter deux lignes dans le programme), il en va tout autrement pour les trois points suivants qui nécessitent quasiment de tout reprendre à partir de zéro.

1.4.3 Robustesse aux changements

Bien que ce problème de la location des barques sur un lac soit artificiel, il faut se rendre compte que dans la réalité, c'est encore pire! Nokia rapporte par exemple à

propos de son infrastructure GSM que :

- 50% des requirements (besoins numérotés) ont changés après le gel du cahier des charges
- 60% de ceux-ci ont changés au moins 2 fois!

Donc avoir affaire à un cahier des charges mouvant est plutôt la règle que l'exception. Le principal problème du génie logiciel ne se pose plus en termes de «donnez-moi une spécification immuable et je produis une implantation de qualité» mais plutôt en «comment réaliser une implantation de qualité avec des spécifications continuellement mouvantes». De plus, il arrive de plus en plus fréquemment de ne plus avoir à produire un produit donné à un moment donné, mais simultanément toute une gamme (ou famille) de produits avec pour prendre en compte des variations de fonctionnalités ou d'environnements. Si on n'y prend pas garde, en particulier dans la manière dont on conçoit son logiciel, on risque d'avoir à faire face à des coûts de maintenance extrêmement élevés (voir figure 1.8).

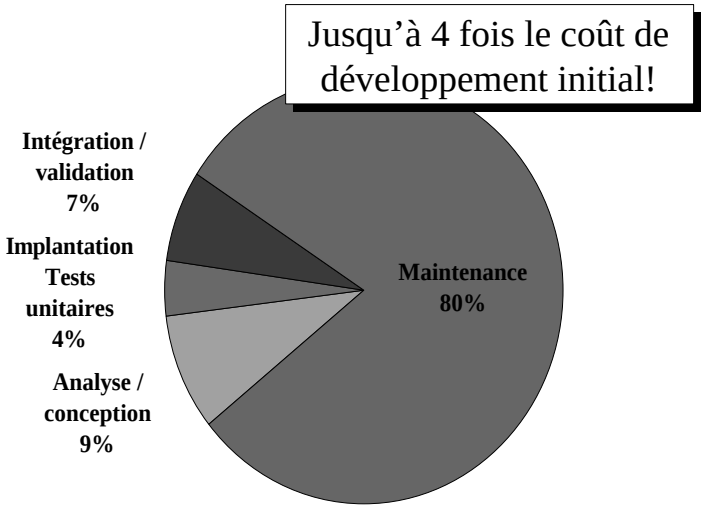


FIG. 1.8 – *Coût de la maintenance*

On a donc affaire à un problème de *discontinuité* dans le processus qui permet de passer de la spécification (domaine du problème) à la réalisation (domaine de la solution), c'est-à-dire qu'un changement, aussi infime soit-il, peut arbitrairement provoquer un très grand changement dans la réalisation (cf. transformation continue vs. transformation discontinue en figure 1.9).

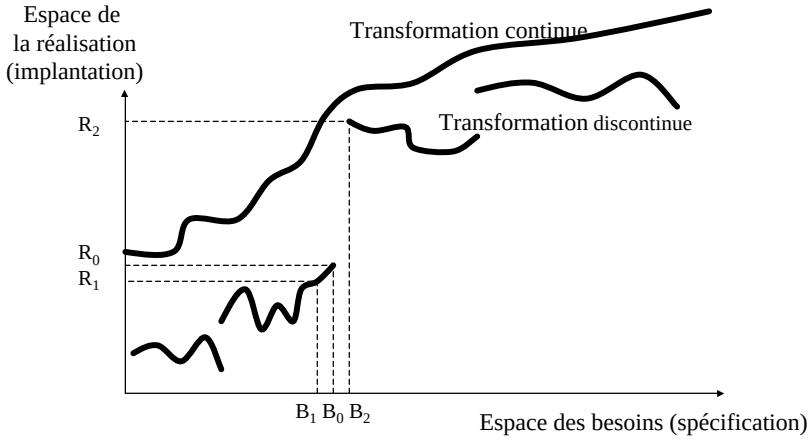


FIG. 1.9 – *Problème de continuité dans le développement*

1.4.4 Solution : approche par modélisation

Du point de vue technique, le problème vient du fait que les fonctionnalités, qui forment la partie du système visible par l'utilisateur, sont donc en réalité ce qui bouge le plus. Ce serait donc une erreur de construire l'architecture du système en suivant cette découpe fonctionnelle : toute remise en cause fonctionnelle pourrait conduire à des changements majeurs dans cette architecture, qui est donc fragile vis-à-vis des changements fonctionnels.

Une approche possible pour circonvenir ce problème consiste à s'appuyer sur la notion de modélisation du domaine du problème et du domaine de la solution. Il s'agit de modéliser d'abord ce qui est stable dans un système, c'est-à-dire les entités et événements selon Jackson (JSD) ; puis seulement ensuite ajouter les fonctionnalités. La maintenance n'est donc pas traitée différemment du développement des fonctionnalités initiales. Le système est plus stable car par définition les entités et les événements transcendent les fonctionnalités, l'architecture du système est donc plus robuste aux changements : il a moins de chances d'être impacté par des changements de besoins fonctionnels, ce qui assure une meilleure continuité entre le domaine du problème et le domaine des solutions.

Du point de vue organisationnel, la prise en compte du fait que dans les systèmes les cahiers des charges ne peuvent en réalité que très rarement être figés (et même alors) implique de s'organiser pour travailler en terrain mouvant. Ce sont les notions de *gestion contrôlée des changements* et *traçabilité*, qui permet de retrouver quelle partie du logiciel correspond à quel besoin et réciproquement.

Chapitre 2

Origines et objectifs d'UML

2.1 Introduction

Comme on l'a vu, les problèmes de l'ingénierie du logiciel se posent aujourd'hui en termes de lignes de produits et de maintenabilité, avec, comme facteur déterminant, des coûts d'évolution du logiciel souvent très largement supérieur aux coûts de développement(jusqu'à quatre fois plus).

L'approche la plus largement employée aujourd'hui pour faire face à ce problème repose sur la modélisation, qui est censée apporter :

- une meilleure continuité entre spécification et réalisation;
- une meilleure communication entre les acteurs d'un projet;
- une meilleure résistance aux changements.

Or cette même idée se retrouve aussi aux origines de l'approche à objet. En effet, le langage Simula a été inventé dans les années soixante pour faciliter l'écriture de programmes de simulation, c'est-à-dire de modélisation de problèmes physiques. Il faut savoir que la version aboutie de ce langage, baptisée Simula 67 (conçu en 1967) possédait déjà, avec trente ans d'avance, les concepts popularisés plus tard par Java : objet, classe, héritage, liaison dynamique...

Mais l'approche objet trouve aussi ses sources dans les domaines suivants :

- les systèmes d'exploitation, avec la notion de *moniteurs* pour protéger l'accès à des données ;
- les types de données abstraits, pour pouvoir décrire les propriétés d'opération indépendamment de leur implantation (ce qui correspond aux *classes abstraites* dans les langages à objets ;
- l'intelligence artificielle, domaine où Minsky [16] a montré que l'intelligence pouvait apparaître comme une propriété émergente d'un système résultant de la com-

position d'unités autonomes de connaissance (les *frames*), elles même simples et non-intelligentes.

Une fois admise l'idée que la modélisation joue un rôle crucial dans le développement de logiciels, il faut encore disposer d'un langage suffisamment abstrait pour être indépendant de telle ou telle plate-forme matérielle ou logicielle, tout en étant suffisamment précis pour permettre de capturer aussi précisément que nécessaire les aspects fondamentaux d'un problème. Depuis les années soixante-dix, où est apparue l'idée qu'il était nécessaire de réfléchir avant de programmer, de très nombreux langages et méthodes ont été proposés pour répondre à ce problème. Schématiquement, on voit apparaître trois générations.

Vers la fin des années 70, une standardisation autour d'approches centrées chacune sur une des dimensions de l'informatique :

- les fonctions à assurer, avec par exemple la méthode SADT ;
- la structure statique, avec par exemple les diagrammes entité-relations (popularisé en France par la méthode Merise) ;
- le comportement dynamique, avec par exemple les machines à états.

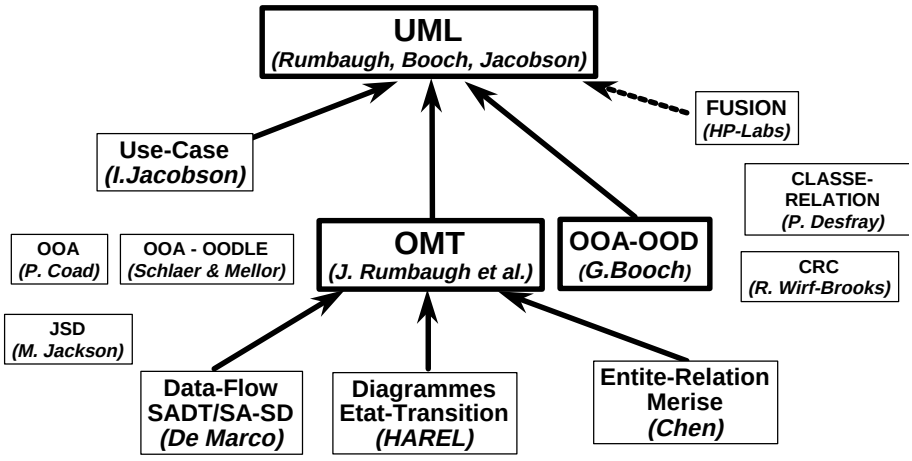


FIG. 2.1 – *Généalogie de UML*

Vers la fin des années quatre-vingts, les limitations de ces approches monodimensionnelles deviennent patentes. D'une part, l'évolution continuelle des services demandées à un logiciel donné et la complexité croissante de son interaction avec le monde extérieur rendent illusoire l'idée même de cahier des charges figé qui étaient la base de ces

approches. Il n'y a plus de spécification à partir de laquelle on pourrait dériver une réalisation quasi optimale (voir le problème des barques sur le lac au chapitre précédent). D'autre part, lorsqu'on développe de grands systèmes, on ne peut plus simplement se limiter à l'une ou l'autre de ses dimensions, mais on doit prendre en compte les trois simultanément. De plus, il apparaît une quatrième dimension due à la décentralisation des logiciels grâce aux réseaux.

Pour répondre à ces défis, Jim Rumbaugh et ses collègues de la société General Electric proposent alors la méthode OMT (Object Modelling Technique), fondée sur une approche par objet mais combinant autour de celle-ci les approches existantes pour les trois dimensions : services, architecture statique et comportement dynamique. Ceci est rendu possible par l'apparition à cette époque des premiers langages à objets réellement opérationnels dans un contexte industriel (Eiffel, C++, etc.). Beaucoup d'autres propositions allant dans le même sens sont faites à la même époque. Grady Booch fait évoluer son entreprise Rational Software du rôle de fournisseur de composants logiciels pour Ada au rôle de fabricant d'ateliers de génie logiciel. Ivar Jacobson formalise dans un livre ses trente ans d'expérience dans le développement d'applications par objets chez Ericsson (il avait commencé dès la fin des années 60, en codant en assembleur des conceptions par objets). Ceci conduit au début des années quatre-vingt-dix à un véritable foisonnement de méthodes d'analyse et de conception par objet (voir figure 2.1).

Devant le succès de ces approches dans l'industrie, une très forte pression se développe pour leur standardisation. Les premières tentatives ayant échouées, c'est finalement le marché qui fait sa loi, quand Jim Rumbaugh et Ivar Jacobson rejoignent Grady Booch chez Rational Software. Tous trois s'engagent dans un processus de convergence de leurs méthodes qui conduira en 1997 à la standardisation par l'OMG du langage de modélisation unifiée UML (Unified Modelling Language). Celui-ci s'imposera alors très rapidement dans l'industrie. Dès l'année 2001 près de 90% des projets logiciel dans le monde l'utilisent à l'une ou l'autre étape du développement. Cette rapidité d'adoption n'est pas très surprenante car UML ne fait finalement qu'une synthèse (ou du moins tente de le faire) des meilleures pratiques existantes pour le développement par objet de logiciels.

2.2 Modélisation UML

2.2.1 Un peu de méthodologie...

Il faut noter qu'UML n'est pas une méthode, c'est-à-dire un guide montrant comment en partant d'une expression des besoins on peut arriver à un logiciel validé. UML est simplement un langage permettant de décrire différents artefacts utilisés lors du développement de logiciels conduit dans une approche itérative, incrémentale, et pilotée par les cas d'utilisation (scénarios d'emploi, rôles assumés par les utilisateurs) : modèle

d'expression des besoins, modèle d'analyse (construction d'un modèle idéal du monde), modèle de conception (passage du modèle idéal au monde réel), modèle d'implantation, modèle de tests pour la validation (voir chapitre suivant). Bien sûr, en fonction de l'activité menée lors du développement de logiciels, on utilisera pas UML exactement de la même manière. Par exemple le fait qu'un attribut soit privé pourra tout à fait avoir sa place dans un modèle implantation, voire de conception détaillée, mais en aucun cas dans un modèle d'analyse qui est censé abstraire tout détail relatif à l'implantation.

2.2.2 Objectifs d'une modélisation UML

Modéliser avec UML consiste à construire un modèle quadridimensionnel sur les dimensions des services, de l'architecture statique, du comportement dynamique, du déploiement (installation). Un peu comme en dessin industriel où l'on dispose de plusieurs types de vues pour représenter des pièces tridimensionnelles (de face, de profil, en coupe, en perspective, etc.), on va conduire une modélisation en UML par projection selon ces quatre points de vue principaux, qui disposent chacun d'un certain nombre de types de vues.

- Aspects statiques du système (le *qui*) :
  - description des objets et de leurs relations ;
  - modularité, contrats, relations, généricité, héritage ;
  - structuration en paquetages.
- Perception du système par l'utilisateur (le *quoi*) :
  - cas d'utilisation.
- Aspects dynamiques du système (le *quand*) :
  - diagrammes de séquences (scénarios) ;
  - diagrammes de collaborations (entre objets) ;
  - diagramme d'états et transitions (Harel) ;
  - diagrammes d'activités.
- Vision implantation (le *ou*)
  - diagrammes de composants et de déploiement.

Nous allons maintenant détailler dans les quatre chapitres suivants l'expression de ces points de vue en UML.

Chapitre 3

Aspects statiques du système

3.1 Rappels sur les notions d'objets et de classe

Rappelons qu'un objet est l'encapsulation d'un état avec un ensemble d'opérations travaillant sur cet état. C'est l'abstraction d'une entité du monde réel, :

- qui a une existence temporelle : création, évolution, destruction ;
- qui a une identité propre à chaque objet ;
- qui peut être vue comme une machine, avec une mémoire privée et une unité de traitement, et rendant un ensemble de services.

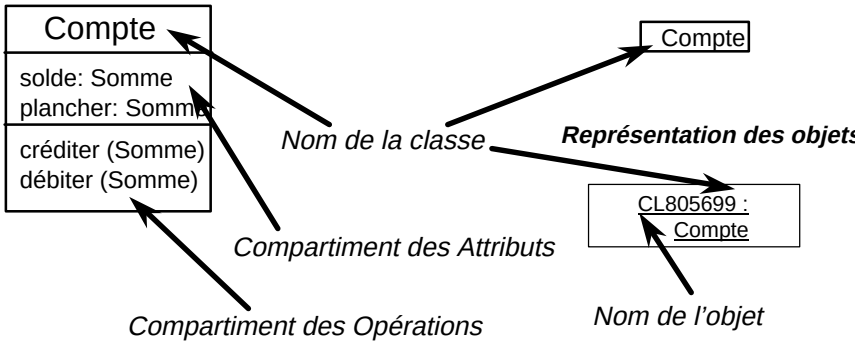


FIG. 3.1 – *Notations UML pour classes et objets*

3.1.1 Une classe est un type

Formellement une classe est l'implantation d'un type de données abstrait : c'est la description des propriétés et des comportements communs à un ensemble d'objets (classe d'équivalence au sens mathématique).

Un objet est une instance d'une classe (on a donc un rapport analogue à celui qui unit une variable et son type). Chaque classe a un nom, et un corps qui définit (voir figure 3.1) :

- les attributs possédés par ses instances (voir figure 3.2),
- les opérations définies sur ses instances, et permettant d'accéder, de manipuler, et de modifier leurs attributs (voir figure 3.3).

Remarquons finalement que dans certains systèmes, une classe peut elle-même être un objet (p. ex. en Smalltalk), et donc instance d'une classe (qu'on appelera sa méta-classe).

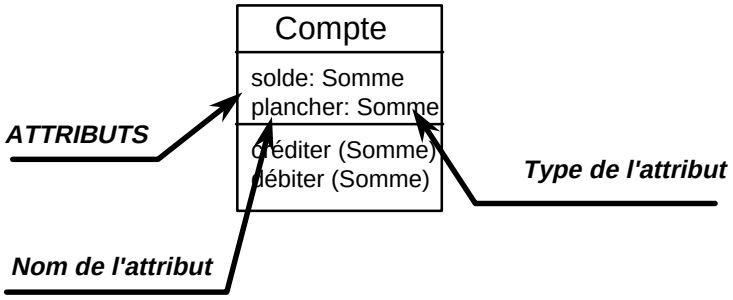


FIG. 3.2 – *Représentation des attributs*

Les *attributs dérivés* sont des attributs dont la valeur peut être déduite d'autres éléments du modèle, par exemple l'âge si l'on connaît la date de naissance (on précédera le nom d'une barre oblique dans ce cas, p. ex. `/age`). En terme d'analyse, cela indique seulement une contrainte entre valeurs et non une indication de ce qui doit être calculé et de ce qui doit être mémorisé.

3.1.2 Une classe est un module

Nous avons vu que la modularité est l'outil principal pour gérer la complexité des systèmes. C'est un concept présent depuis longtemps en informatique, avec les *subroutines* du langage Fortran, ou la notion d'unité de compilation (p. ex. le fichier du langage C).

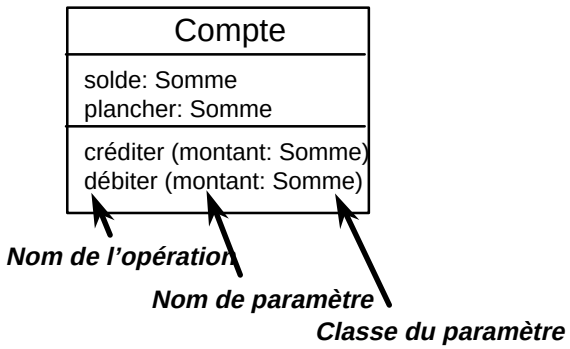


FIG. 3.3 – *Représentation des opérations*

Un progrès majeur a été effectué lorsque la notion de module a été intégrée aux langages, avec Modula-2 (notion de module), Ada83 (notin de paquetage, *package*), puis bien sûr à tous les langages à objets.

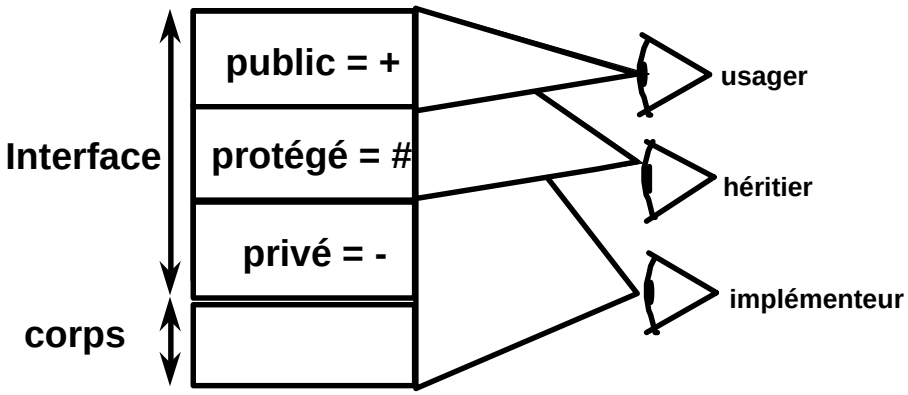


FIG. 3.4 – *Visibilité*

La notion de module permet :

- le masquage d'information (voir figure 3.4) et donc l'abstraction : utiliser un module n'impose pas la connaissance des détails de mise en œuvre ;
- l'encapsulation (facilite les modifications à portée locale)

Cette dissociation entre interface et implantation est la condition *sine qua non* pour obtenir des composants réutilisables (voir figure 3.5).

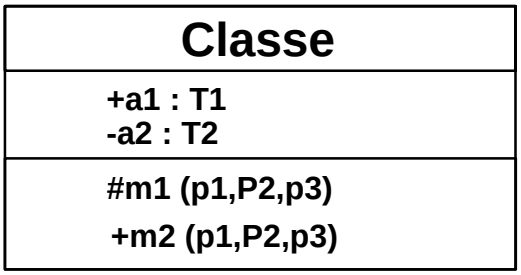


FIG. 3.5 – *Visibilité : Exemple*

3.1.3 Principe de l'approche à objet

L'approche à objet structure les systèmes autour des objets plutôt qu'autour des fonctions, en se fondant sur la construction de modèles. Ceux-ci facilitent la communication entre le donneur d'ordre (qui fournit le cahier des charges et validera le logiciel *in fine*) et les activités de développement et de maintenance (vérification). Les modèles assure une certaine forme de continuité entre les différentes activités du cycle de vie des logiciels (voir Jackson et la méthode JSD). L'objectif est d'obtenir ainsi des systèmes modulaires et maintenables, en particulier dans le contexte de la création de lignes de produits (c'est-à-dire d'un ensemble de logiciels qui partagent de larges parties commune mais contiennent des spécificités adaptées à tel ou tel segment de marché ou clients particuliers). Il s'agit donc de permettre de procéder par assemblage de briques de base plutôt que de faire du développement *ad hoc*, et de produire des canevas d 'application capables de répondre à des gammes de besoins plutôt que des programmes répondant à un besoin précis à un instant précis.

Ceci a bien sûr un impact très fort sur les métiers du développement du logiciel, avec une découpe qui n'est plus verticale entre analystes, concepteurs, implémenteurs, les testeurs, mais qui s'articule plutôt autour des notions d'experts métiers, d'architectes, de réalisateurs de composants, et d'assembleurs de composants.

Ceci a aussi un impact extrêmement fort sur les besoins qualitatifs pour les composants qui sont développés et assemblés. On distingue des facteurs de qualité externe (c'est-à-dire l'objectif qu'on cherche à atteindre pour un composant) et les facteurs de qualité interne (c'est-à-dire les moyens qu'on utilise pour atteindre ces critères de qualité externe).

3.1.4 Facteurs de qualité externes

– **Correction.** C'est l'aptitude des composants à mener à bien les tâches pour

lesquels ils ont été conçus. Vérifier la correction d'un composant n'est bien sûr possible que si l'on dispose de sa spécification (son « contrat »).

- **Réutilisabilité.** Elle permet de considérer des composants logiciels comme des briques de base assemblables pour bâtir de nouveaux développements. Il s'agit de récupérer des composants créés antérieurement, plutôt que de tout reconstruire de zéro. Les économies attendues se situent moins dans les phases de développement initiales (un bon éditeur de texte muni de fonctions de *couper-coller* serait tout aussi efficace) que pour les phases de test, d'intégration et surtout de maintenance du cycle de vie du logiciel. Mais pour réutiliser un composant dans une application critique, il faut avoir suffisement confiance en ce composant. Mais comment mesurer un degré de confiance ?
- **Extensibilité.** Elle permet d'ajouter facilement de nouvelles fonctions (services) à des systèmes préexistants. L'extensibilité repose ici sur l'idée d'une certaine *continuité* (au sens mathématique du terme) dans la transformation du domaine du problème vers le domaine de la solution : un petit changement dans le problème à traiter doit induire un petit changement dans la solution (l'implantation). Ceci permet de s'adapter à un cycle de vie incrémental, minimisant les ruptures entre développement initial et maintenance, ce qui est l'idée maîtresse de l'utilisation de la technologie objet dans un contexte de génie logiciel.
- **Compatibilité.** C'est la facilité avec laquelle des composants peuvent être combinés et assemblés pour constituer des systèmes utiles.
- **Robustesse.** C'est la faculté de pouvoir « fonctionner » même dans des conditions anormales, c'est à dire non prévues dans les spécifications. Pour des composants logiciels, cela se résume à essayer d'éviter de provoquer des catastrophes quand les choses dérapent. Un mécanisme de gestion discipliné des exceptions joue un rôle fondamental pour améliorer la robustesse.
- **Intégrité.** C'est la faculté que peuvent avoir des composants logiciels de protéger de toute corruption la cohérence de leur état interne. Si ce problème est largement résolu par la notion d'encapsulation pour des composants utilisés dans un cadre séquentiel, il se pose de manière accrue dans un cadre ou plusieurs flots d'exécution peuvent traverser simultanément un même objet.
- **Testabilité.** C'est la facilité avec laquelle on peut préparer des suites de validation pour un composant logiciel. La testabilité s'appuie sur le paradigme sous-jacent de programmation par contrat.
- **Efficacité** C'est le bon usage des ressources (p. ex., processeur ou mémoire), pour réaliser les bons compromis entre forces conflictuelles. Dans la plupart des systèmes, l'efficacité doit être jaugée à l'aune du coût total de réalisation et de maintenance d'un composant, y compris en ressources humaines.
- **Portabilité.** C'est la facilité avec laquelle un composant peut être porté dans

différents environnements logiciels et matériels.

- **Ergonomie** C'est la facilité avec laquelle on peut apprendre à se servir d'un composant logiciel. Il faut qu'il soit plus facile et plus sûr d'apprendre à utiliser un composant que de le refaire. Une interface intuitive et une bonne documentation sont ici des prérequis majeurs.

3.1.5 Les concepts de l'approche à objet

Pour aller vers la réalisation des objectifs de qualité externe, on a besoin d'un certain nombre de concepts qui sont apparus au fur et à mesure de l'histoire de l'informatique, pour résoudre des problèmes bien concrets :

- la modularité (*cf.* ci-dessus) ;
- les contrats (exprimés en langage OCL) ;
- les relations entre objets (abstraction de l'implantation) ;
- la généricité (modules paramétrés) ;
- l'héritage (classification, réutilisation) et liaison dynamique.

Nous allons détailler ces aspects dans les sections suivantes.

3.2 Spécification de contrats avec OCL : Object Constraint Language

3.2.1 Problème de la validité des composants

Le problème de la validité des composants s'articule en plusieurs catégories.

Validité intracomposant Il s'agit du problème de la validité d'un composant isolé. On fait l'hypothèse d'un environnement correct, et le problème est équivalent à la correction d'une réalisation vis-à-vis d'une spécification. Du fait des problèmes d'indécidabilité évoqués au chapitre introductif, on sait que ce problème n'est pas si facile que cela dans le cas général.

Validité intercomposants Il s'agit du problème de la validité d'un assemblage de composants, où on fait l'hypothèse que chaque composant est valide vis-à-vis de sa spécification. C'est donc le problème de l'intégration de systèmes.

Pour comprendre la complexité de la Validité Inter-composants, il est utile de revenir sur un exemple fort instructif, à savoir la destruction du lanceur Ariane 5 lors de son premier vol de qualification, le 4 Juin 1996,12:34 TU. La cause ultime est la réutilisation dans Ariane 5 d'un composant matériel et logiciel ayant donné toute satisfaction pour Ariane 4, mais qui avait une contrainte de domaine de validité incompatible avec Ariane 5 (voir figure 3.6).

- H0 -> H0+37s : nominal
- Dans SRI 2:
  - BH (Bias Horizontal) > 2^15
  - convert_double_to_int(BH) fails!
  - exception SRI -> crash SRI2 & 1
- OBC disoriented
  - Angle attaque > 20°,
  - charges aérodynamiques élevées
  - Séparation des boosters

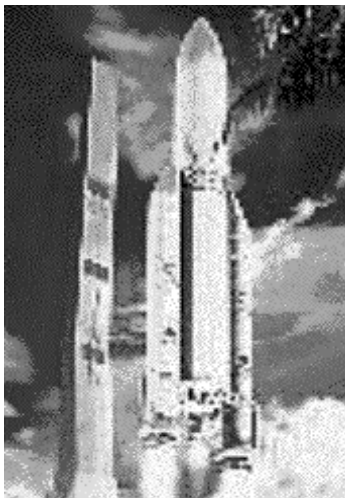


FIG. 3.6 – *Ariane 501 Vol de qualification Kourou, ELA3 – 4 Juin 1996,12:34 UT*

De H0 à H0+37s, tout se passe de manière essentiellement nominale (conforme aux prévisions). Puis dans le système de référence inertielle numéro 2 (SRI2), la variable BH (Bias Horizontal, qui représente la vitesse horizontale de la fusée telle que lue sur le capteur) devenant supérieure à 2^{15}, une opération de conversion vers un entier court (sur seize bits) échoue, ce qui déclenche une exception. À son tour cette exception bloque les systèmes de référence inertiels numéro 2, puis comme il s'agit d'un problème logiciel, du numéro 1 dans la foulée (les mêmes causes produisant le même problème). Les deux systèmes de référence inertiels vident alors leur état mémoire sur le bus partagé. Ceci désoriente complètement l'ordinateur de bord principal. Interprétant alors ces données mémoire comme des données de vol, l'ordinateur principal tente de faire braquer arbitrairement la fusée avec des angles d'attaque supérieure à 20°, ce qui provoque des efforts aérodynamiques si élevés qu'il y a risque de séparation prématurée des boosters. Pour éviter cela, l'ordinateur de bord déclenche l'autodestruction de la fusée à H0 + 39s (coût de l'échec : 500 millions d'euros).

Il est intéressant d'analyser la cause première qui, par une succession d'enchaînements fortuits a finalement provoqué l'explosion du lanceur (voir l'article [13] pour plus de détails). En effet, contrairement aux apparences :

- ce n'est pas une erreur de programmation : l'absence de contrôle de la conversion est une décision de conception datant du début des années quatre-vingt.
- ce n'est pas une erreur de conception, car cette décision était justifiée par les caractéristiques physiques d'Ariane 4 et et les contraintes temps-réel globales.
- ce n'est pas un problème de test d'intégration ; comme toujours, un test bien choisi aurait pu mettre en évidence le problème (ceci a d'ailleurs été fait *a posteriori*).

La taille gigantesque de l'espace des tests (comme expliqué au paragraphe 1.2.1), confrontée aux ressources limitées allouées aux opérations de validation (et à la pression pour la mise sur le marché le plus vite possible du produit) a limité le nombre des tests d'intégration. De plus, sachant que le système de référence inertiel était inutile à cette étape du vol (ce qui est en passant particulièrement vexant), ce n'est pas un point sur lequel les ingénieurs de validation ont particulièrement fait porter leurs efforts.

La cause première est en effet la réutilisation dans Ariane 5 d'un composant de Ariane 4 ayant une contrainte « cachée », à savoir une restriction du domaine de définition d'une fonction (fonction dont le domaine de définition était valide si et seulement si $abs(BH) < 32768$). Compte tenu des données de vol, cette contrainte était parfaitement satisfaite pour Ariane 4, mais elle l'était plus pour Ariane 5.

3.2.2 Notion de contrats entre composants

Il est donc clair que pour permettre à des composants d'interopérer, le typage (par la seule signature des méthodes) est insuffisant : on a besoin de pouvoir exprimer des restrictions sur les valeurs d'entrées et de sorties possibles, et de préciser la sémantique, c'est-à-dire ce que fait une méthode (le *quoi*) sans entrer dans le détail du *comment*. Cette indépendance vis-à-vis de l'implantation est inspirée par la notion de type abstrait de données, où une spécification contient en plus de la signature :

- des préconditions : spécification des conditions nécessaires pour qu'un client soit autorisé à appeler une méthode. L'appelant de la méthode doit garantir ces conditions lors de l'appel, car sinon le travail de la méthode ne peut pas être effectué. Plus formellement, une précondition caractérise l'ensemble des états initiaux pour lesquelles un problème peut être résolu. Elle spécifie le sous-ensemble de tous les états possibles qu'une méthode devrait pouvoir gérer correctement.
- des postconditions : propriétés garanties par une méthode, spécification de ce qui sera vrai à la fin d'une exécution d'opération si la précondition est vérifiée. La postcondition décrit donc les propriétés que la méthode doit garantir à la fin d'un appel correct. Elle fournit une spécification formelle de ce que la méthode est censée accomplir. Elle peut donc aider à la construction de cette méthode, et être utilisée dans la preuve constructive de sa correction.
- des invariants : propriétés vraies pour l'ensemble des instances de la classe. Les invariants de classe n'appartiennent pas syntaxiquement à une méthode particulière. Ils caractérisent des propriétés que le module englobant doit respecter avant et après l'appel d'une de ses méthodes. On peut considérer qu'un invariant de classe est à la fois une précondition et une postcondition de chacune des méthodes de la classe. Dans un état stable, chaque instance doit vérifier les invariants de sa classe de classe (à l'entrée et à la sortie des corps d'opération).

En tant qu'outils de spécification, préconditions et postconditions ont pour objectif de décrire le *quoi* plutôt que le *comment*.

Cette spécification forme un véritable *contrat* entre un composant et ses clients. Dans les systèmes répartis, au delà de ces simples aspects, on distingue habituellement quatre niveaux de contrats logiciels [3] :

1. elémentaire (syntaxique) : la compilation du programme se fait sans erreurs ;
2. comportemental (services) : définition de pré- et postconditions ;
3. synchronisations : p. ex. expressions de chemin (*path expressions* [McHale]), etc. ;
4. qualité de service (quantitative) : négociation dynamique possible

3.2.3 Introduction à OCL

Le langage OCL (*Object Constraint Language*) [24] a été créé spécialement pour répondre au besoin de formaliser des contrats sur les modèles UML. OCL permet d'exprimer des contraintes restreignant les domaines de valeurs pouvant être ajoutées aux éléments du modèle UML. Une contrainte OCL est une expression booléenne (sans effet de bord) portant sur :

- les opérations usuelles sur types de base (Boolean, Integer...) ;
- les attributs d'instances et de classes ;
- les opérations de *query* (fonctions sans effet de bord) ;
- les associations du modèle UML ;
- les états des *statecharts* associés.

Le langage OCL est fortement typé et sans effet de bord. Le langage consiste en un ensemble de types et d'opérations prédéfinis, regroupés et documentés sous la forme d'un package UML appelé *UML_ OCL*. Le concepteur d'un modèle UML est censé importer ce package dans son modèle pour disposer des fonctionnalités d'OCL. Le package *UML_ OCL* contient des types élémentaires tels les booléens, entiers, réels et chaîne de caractères, ainsi que des types plus complexes tels les types énumérés ou les collections. Tous les types OCL sont des sous-types du type `OclAny` qui définit les propriétés communes à tous les objets OCL.

À ces types prédéfinis, viennent s'ajouter des types « importés » qui correspondent aux classes, interfaces et type de données du modèle. En effet, une expression OCL est toujours écrite dans un contexte qui peut être soit un type, soit une opération, et qui sert de point d'origine lorsque l'on veut naviguer dans un modèle. Si le contexte est un type, le mot-clé self dans une expression se rapporte à un objet de ce type. Si le contexte est une opération, self désigne le type qui possède cette opération. Tous les types OCL, qu'ils soient prédéfinis ou importés sont accessibles via une instance du type OCL `OclType`.

Une expression OCL est toujours écrite dans un *contexte* qui peut être soit un type, soit une opération. Si le contexte est un type, le mot-clé `self` dans une expression se rapporte à un objet de ce type. Si le contexte est une opération, `self` désigne le type qui possède cette opération.

---

context Type
-- *ceci définit un invariant pour Type*
inv: expression−ocl

OCL permet de spécifier le comportement d'une opération par des préconditions et des postconditions. Le mot-clé `result` ne peut apparaître que dans une postcondition, où il désigne le résultat de l'évaluation de l'opération :

---

context Type::opération(param1 : Type, ... , param2 : Type) : Type
-- *ceci spécifie une opération OCL par ses pré et postconditions*
pre: expression−ocl
post : expression−ocl

Par exemple, la contrainte suivante a pour contexte la classe `CompteBancaire` et spécifie un invariant que les objets de cette classe devront respecter :

---

context CompteBancaire
-- *les découverts ne sont pas autorisés au delà d'un certain plancher*
inv: solde $\geq$ plancher

Les deux expressions OCL qui suivent ont pour contexte respectif les opérations `créditer` et `débiter` de la classe `CompteBancaire`. Par exemple, les opérations `créditer` et `débiter` sont spécifiées par les préconditions et postconditions suivantes, où la notation `@pre` dans `solde@pre` désigne la valeur de l'attribut `solde` avant l'exécution de l'opération (idem `old` dans le langage Eiffel).

---

context CompteBancaire::créditer(montant : **Integer**)
pre: montant $>$ 0
post: solde = solde**@pre** + montant

---

context CompteBancaire::débiter(montant : **Integer**)
pre: montant $>$ 0 **and** montant $\leq$solde − plancher
post: solde = solde**@pre** − montant

3.2.4 Représentation des contraintes OCL

Il est possible de faire apparaître des contraintes OCL directement dans le modèle UML en les incluant entre des accolades et en les accrochant à un élément de modèle (voir figure 3.7).

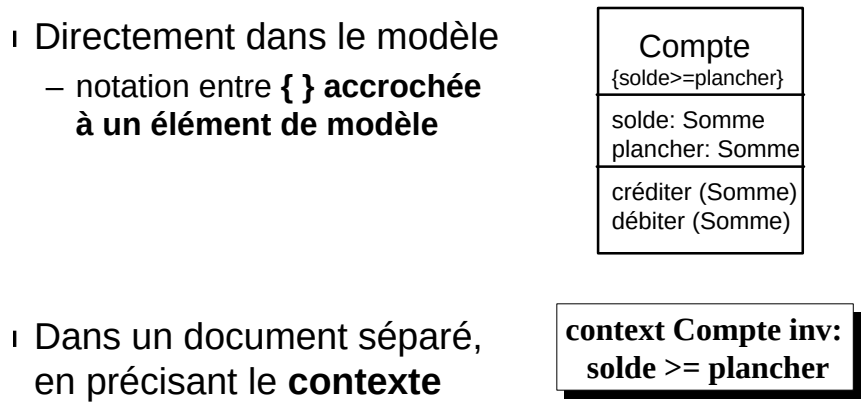


FIG. 3.7 – *Représentation des contraintes OCL*

Ainsi, il est donc possible d'être à la fois abstrait (aucun préjugé sur la manière dont serait implantée la classe) *et* aussi précis que nécessaire avec UML (voir figure 3.8).

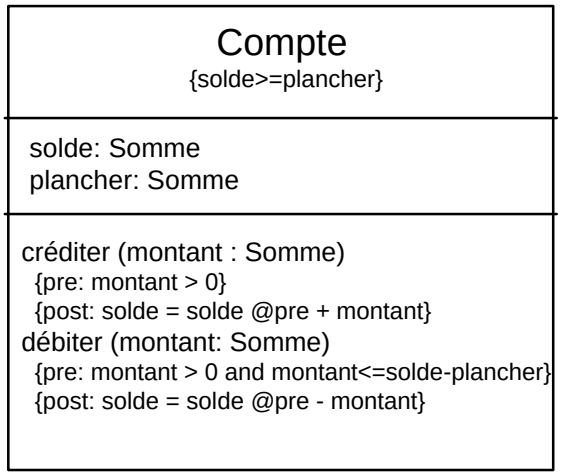


FIG. 3.8 – *Etre abstrait et précis avec UML*

A cet égard, le principal intérêt d'UML est de permettre au modélisateur de choisir lui même le niveau de précision dont il a besoin en fonction de ses contraintes de projet (en particulier du compromis qualité et coûts comme évoqué au paragraphe 1.3), depuis un modèle tout à fait imprécis qui reste toutefois de l'UML correct) jusqu'à un modèle complètement formalisé sur lequel il serait possible de faire des preuves, et ceci *sans changer de formalisme* (en accord avec la continuité évoquée au paragraphe 1.4).

3.2.5 Conception par contrat

Les préconditions, postconditions et invariants jouent un rôle crucial dans la répartition des responsabilités de la construction d'un système modulaire. Ces assertions permettent en effet la formalisation du *contrat* (voir [15]) liant l'appelant d'une méthode (le client) et l'implantation de cette méthode (le contractant ou fournisseur). Ce contrat peut alors s'énoncer de la manière suivante :

> *Pourvu que le client appelle la méthode d'un contractant dans un état où l'invariant de classe et la précondition de cette méthode sont respectés, ce contractant promet que lorsque que la méthode aura été exécutée, le travail spécifié dans la postcondition aura été effectué, et l'invariant de classe sera toujours respecté.*

Si l'une des parties ne respecte pas les termes du contrat, l'ensemble du programme doit être considéré comme invalide. En d'autres termes, il y a un bogue quelque part, et une exception doit être levée. Dans les environnements de programmation les plus courant (Eiffel, Java avec par exemple iContract, C++), on peut faire générer du code pour aider à identifier la partie fautive ainsi que la faute elle-même. Le contrat peut en effet être rompu de deux manières différentes selon qu'il s'agisse de la précondition ou de la postcondition.

- Si la précondition a été violée, alors le coupable est le client, car il a rompu le contrat. Du point de vue de l'application, ce client est défectueux. Le contractant n'a alors pas besoin de remplir sa part du contrat, mais devrait autant que possible signaler la faute en levant une exception (voir figure 3.9).

```
Unhandled exception: Routine failure. Exiting program.
Exception history:
================================================================================
Object Routine
Type of exception          Description          Line
================================================================================
#<BANK_ACCOUNT5f0c0>                            BANK_ACCOUNT:deposit
precondition violated      positive_amount      63
--------------------------------------------------------------------------------
#<USER 5f000>                                   USER:test
Routine failure                                 90
--------------------------------------------------------------------------------
#<DRIVER 5f010>                                 DRIVER:make
Routine failure                                 18
--------------------------------------------------------------------------------
```

FIG. 3.9 – *Contract Violations: Preconditions*

- Si la précondition a été satisfaite mais que la postcondition a été violée, alors l'implantation de la méthode ne remplit pas ses engagements : il s'agit d'un bogue dans l'implantation de la méthode, donc sa localisation est assez aisée (par exemple entre les lignes 63 et 70 dans la figure 3.10).

```
Unhandled exception: Routine failure. Exiting program.
Exception history:
================================================================================
Object Routine
Type of exception          Description                Line
================================================================================
#<BANK_ACCOUNT5f0c0>                                  BANK_ACCOUNT:deposit
postcondition violated     deposited                  70
--------------------------------------------------------------------------------
#<USER 5f000>                                         USER:test
Routine failure                                       90
--------------------------------------------------------------------------------
#<DRIVER 5f010>                                       DRIVER:make
Routine failure                                       18
--------------------------------------------------------------------------------
```

FIG. 3.10 – *Contract violations: Postconditions*

Le principe de la conception par contrat fournit un guide méthodologique pour construire des systèmes à la fois robustes, simples et modulaires, tout en permettant d'éviter d'avoir à recourir à la programmation défensive (qui consiste à ne jamais faire confiance au code appelant ou appelé). En effet, les responsabilités étant clairement assignées, les contrôles de validité peuvent être localisés en un seul endroit et non répétés de manière défensive tout au long du code, ce qui conduit à des programmes plus simples, plus court, plus lisible, et donc plus facilement maintenables.

Les contrats sont aussi particulièrement utiles pendant les activités d'intégration du logiciel. En effet, la source des fautes étant immédiatement localisée, il y a d'une part peu de temps perdu en déboggage effectif ; mais surtout comme le responsable de la faute est identifié sans ambiguïté, on ne perd pas de temps en discussions stériles visant à trouver un bouc émissaire. L'expérience montre que ceci améliore considérablement l'ambiance de travail dans les équipes de développement.

3.3 Relations entre classes

3.3.1 Le concept de relation

Comme on l'a vu plus haut, le principe de l'approche à objet est de structurer les systèmes autour d'objets les plus simples possibles pris chacun individuellement, mais interagissant entre eux de manière sophistiquée. Les *relations* entre objets sont donc au moins aussi importantes que les objets eux-mêmes. Le concept de relation peut être appréhendé selon deux points de vue.

- Vue ensembliste : une relation met en correspondance des éléments d'ensemble (des personnes avec les comptes que celles-ci possèdent dans la figure 3.11 : chaque compte est possédée par exactement 1 personne, mais une personne peut posséder un nombre quelconque de comptes).
- Vue constructioniste : une relation permet la description d'un concept à l'aide d'autres concepts.

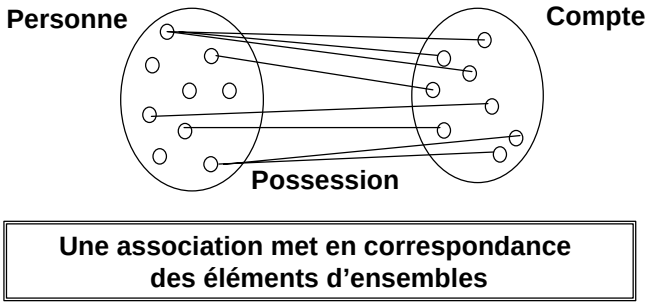


FIG. 3.11 – *Vue ensembliste d'une relation : Graphe de la relation*

Une relation est un lien stable entre objets, c'est-à-dire qui « survit » à un appel de méthode (en pratique elle aura besoin d'être stockée dans le tas).

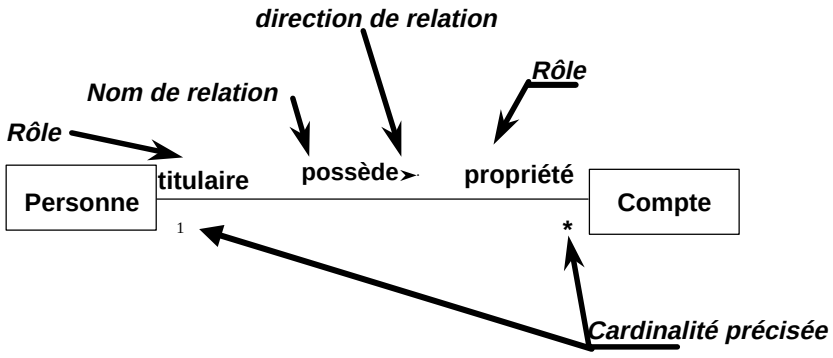


FIG. 3.12 – *Représentation des relations : direction, rôle, cardinalité*

Un lien entre objets est modélisé en UML par un trait reliant leurs classes, portant un nom (le nom de la relation[1]), des rôles (optionnels, ils indiquent quel rôle joue chaque classe dans la relation), et des cardinalités. Une cardinalité indique combien d'instances d'une classe sont en relation avec combien d'instances de l'autre. Par exemple dans la figure 3.12, on exprime qu'une *Personne*s peut possécer un nombre quelconque de *Compte*s (utilisation du symbole *), et que dans la relation *possède*, la classe *Compte* joue le role de *propriété*, et la classe *Personne* celui de *titulaire*.

Il est possible de représenter les différents types de cardinalité comme indiqué en figure 3.13.

1. Il peut y avoir plusieurs sortes de relations entre deux mêmes classes, d'où l'importance de les nommer.

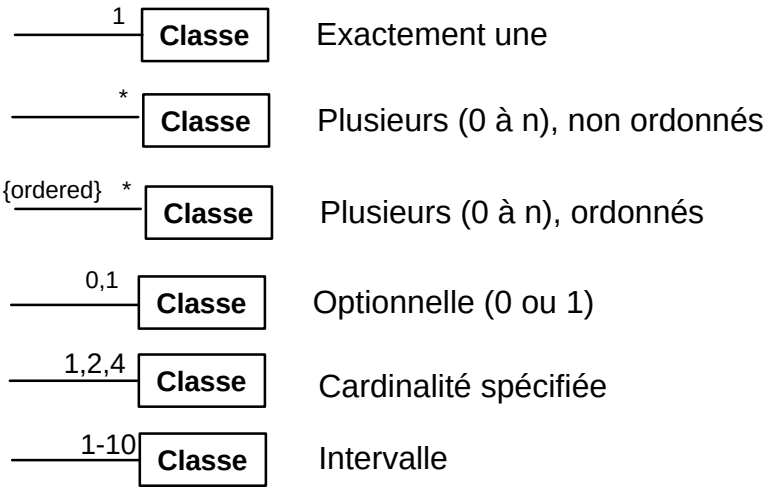


FIG. 3.13 – *Cardinalité d'une relation*

3.3.2 Cas particuliers de relations

Relations réflexives

Voir figure 3.14.

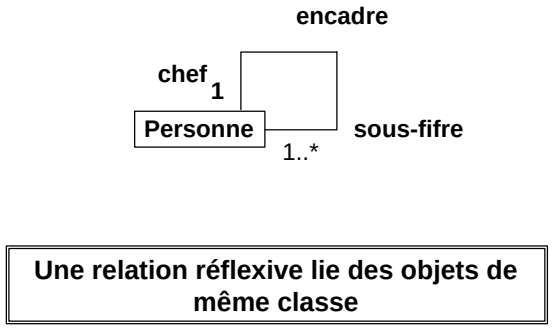


FIG. 3.14 – *Cas particuliers de relations : Relations réflexives*

Composition et agrégation : notion de tout et parties

Voir figure 3.15.

Autre vues de la composition et agrégation : différentes formes suggérant l'inclusion (voir figure 3.16).

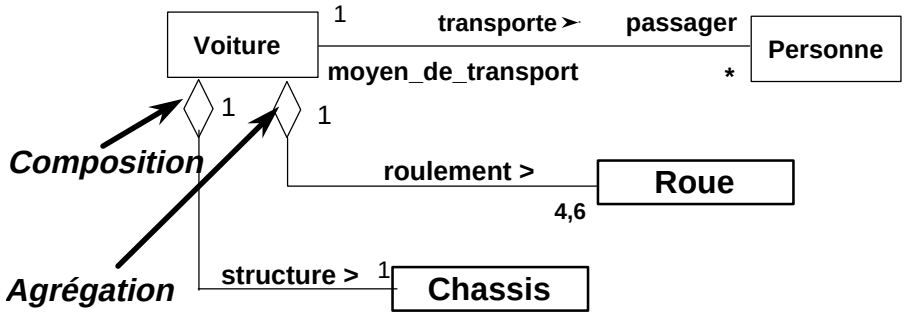


FIG. 3.15 – *Composition et Agrégation : notion de tout et parties*

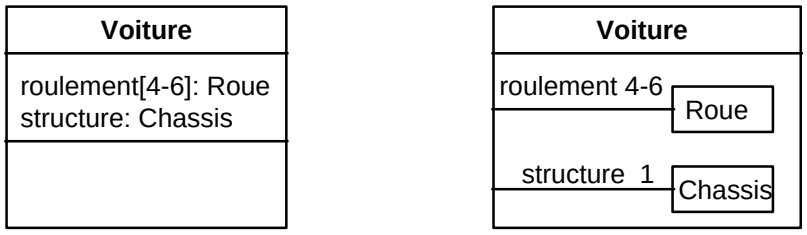


FIG. 3.16 – *Autre vues de la composition/agrégation*

Relations n-aires

Il s'agit de relations entre plus de 2 classes (voir figure 3.17).

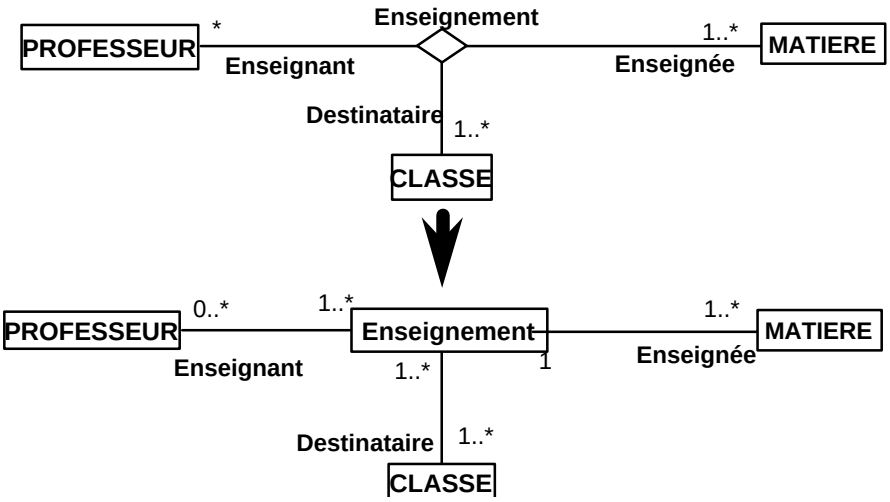


FIG. 3.17 – *Relations n-aires*

Relations attribuées

Ce type de relation est particulièrement utile quand l'attribut (par ex. la note qu'obtient un étudiant dans une matière) ne peut être associé à aucune des deux classes en particulier, car l'attribut porte en fait sur la relation (voir figure 3.18).

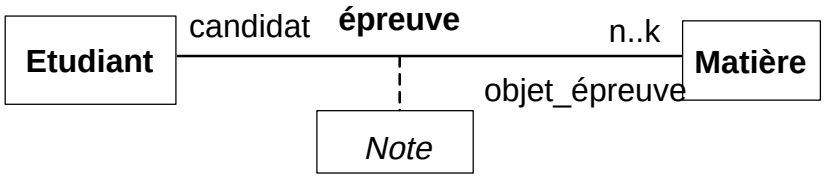


FIG. 3.18 – *Relations attribuées*

Les relations en tant que classes

Voir figure 3.19.

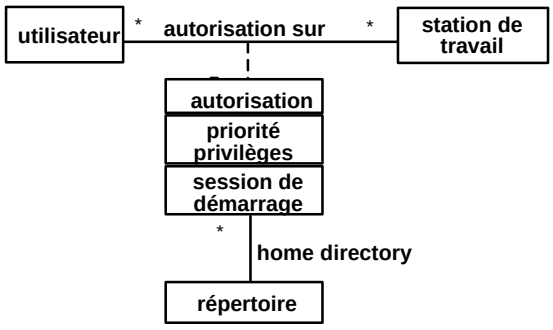


FIG. 3.19 – *Les relations en tant que classes*

Les relations qualifiées

Un qualifieur de relations est un attribut spécial qui permet, dans le cas d'une relation 1-vers-plusieurs ou plusieurs-vers-plusieurs, de réduire la cardinalité. Il peut être vu comme une clé qui permet de distinguer de façon unique un objet parmi plusieurs (voir figure 3.20). C'est l'abstraction des solutions d'implantation de type "mémoire associative" (tables de hachage et autres dictionnaires).

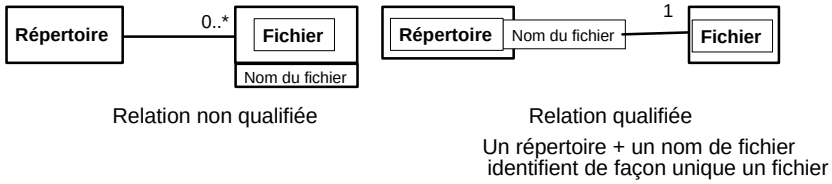


FIG. 3.20 – *Qualifieurs de relations : Exemple*

3.3.3 Contraintes OCL navigant les relations

Afin de pouvoir décrire des assertions non locales à une classe, il est possible d'utiliser dans une expression OCL, en plus des attributs et des opérations *isQuery*, les noms figurant sur les bouts opposés des associations attachées à la classe qui sert de contexte; si les noms sont omis, c'est le nom de la classe attachée au bout de cette association qui est utilisé. Le type de cette expression dépend des caractéristiques du bout de l'association (cardinalité et ordonnancement); nous préciserons ce typage dans la section 3.3.3. Ainsi, dans le contexte `Personne`, l'expression `self.voiture` désigne l'ensemble des voitures possédées par la personne, et son évaluation renvoie une collection d'objets,

plus précisément un `Set(Voiture)`. Inversement, dans le contexte `Voiture`, l'expression `self.personne` est du type `Personne` et désigne le propriétaire de la voiture.

OCL défini 3 sous-types de collections :

Set est obtenu par navigation d'une relation 0..*, chaque élément est présent au plus une fois ;

Context Personne
inv: propriété.**notEmpty** -- *propriété retourne un Set[Voiture]*

Bag si plus d'un pas de navigation, un élément peut être présent plus d'une fois ;

Sequence permet la représentation d'une relation ordered (ordonnée), c'est un donc Bag ordonné.

Les opérations sur les collections qui sont prédéfinies dans le packageage *UML_OCL* sont nombreuses (union, intersection, sélection, etc.). Nous nous contentons ici de rappeler leurs principales caractéristiques. Syntaxiquement, les opérations portant sur des collections se distinguent par l'utilisation du symbole →. Cette notation est du simple sucre syntaxique dans la plupart des cas, sauf lorsqu'elle est utilisée comme raccourci pour indiquer une conversion implicite en collection, plus précisément en `Set`. L'exemple suivant illustre cette particularité :

-- dans l'expression suivante, size est l'opération String::size:Integer
-- l'évaluation retourne la taille de 'toto' en nombre de caractères
`'toto'`.**size** = 4

-- dans l'expression suivante, size est l'opération Collection::size:Integer
-- une conversion implicite asSet est effectuée, et le résultat est le nombre d'éléments de cette collection
`'toto'`→**size** = `'toto'`→**asSet**→**size** = **Set**{`'toto'`}→**size** = 1

Voici quelques opérations de base sur collections :

isEmpty vrai si la collection n'a pas d'éléments ;

notEmpty vrai si la collection a au moins un élément ;

size nombre d'éléments dans la collection ;

count (elem) nombre d'occurrences de elem dans la collection.

Opération collect

Il est possible de combiner les traversées successives de plusieurs propriétés et cela permet de construire rapidement de puissantes expressions de navigation dans le modèle. L'opération `collect` accomplit cette fonction. Différentes syntaxes sont possibles :

---

collection →**collect**(elem:T | expr)
collection →**collect**(expr)

collection . expr

Par exemple, si l'on veut indiquer que le propriétaire d'une voiture doit toujours faire partie de ses passagers :

context Personne
  inv: propriété→**collect**(passager)→**count**(self)=1

ou en raccourci :

context Personne
  inv: propriété . passager→**count**(self)=1

Opération select

Syntaxes possibles

collection →**select**(elem:T | expr)
collection →**select**(elem | expr)
collection →**select**(expr)

Sélectionne le sous-ensemble de collection pour lequel la propriété *expr* est vraie, p. ex.

Set {1, 2, 3, 4, 5} →**select**(i : **Integer** | i > 3) = **Set**{4, 5}

ou bien en navigant une relation, pour dire par exemple que toute personne doit posséder au moins une voiture dont le kilométrage est inférieur à 100.000 km :

context Personne **inv**: propriété→**select**(v: Voiture | v.kilometrage<100000)→**notEmpty**
-- ou en raccourci :
context Personne **inv**: propriété→**select**(kilometrage<100000)→**notEmpty**

Opération forAll

Syntaxes possibles :

collection → forall (elem:T | expr)
collection → forall (elem | expr)
collection → forall (expr)

L'opération *forAll* rend vrai si *expr* est vraie pour chaque élément de collection, p. ex. :

Bag{1, 2, 3, −2, 4}→**forAll**(i | i > 0) = **false**

ou bien en naviguant une relation, pour dire par exemple qu'une personne ne peut posséder que des voitures dont le kilométrage est inférieur à 100.000 km :

```
context Personne inv: propriété→forall(v: Voiture | v.kilometrage<100000)
-- ou en raccourci :
context Personne inv: propriété→forall(kilometrage<100000)
```

Autres opérations OCL

exists (expr) Rend vrai si *expr* est vraie pour au moins un élément de la collection ;

includes(elem), excludes(elem) vrai si *elem* est présent (resp. absent) dans la collection ;

includesAll(coll) vrai si tous les éléments de coll sont dans la collection ;

union (coll), intersection (coll) opérations classiques ensemblistes

asSet, asBag, asSequence conversions de type.

Opération Iterate

Toutes les opérations pour le traitement des collections s'appuient en fait sur une opération de base, l'opération `iterate`. Elle utilise la forme syntaxique suivante :

```
collection→iterate(elem : Type; acc : Type2 = expression |
                                        expression−avec−elem−et−acc)
```

L'identificateur `elem` va prendre la valeur de chaque élément de la collection, et `acc` joue le rôle d'un « accumulateur » des résultats intermédiaires obtenus pendant le calcul. Par exemple :

```
context Banque
   -- les fonds d'une banque sont égaux à la somme des soldes des comptes
   inv: fonds = self.compteBancaire→iterate(c:CompteBancaire;acc:Integer=0|acc + c.solde)
```

Nous donnons ci-dessous l'interprétation de l'opération `iterate` telle qu'elle est décrite dans [24].

```
iterate(elem : T; acc : T2 = value)
        {
        acc = value;
        for (Enumeration e = collection.elements() ; e.hasMoreElements(); )
                {
                elem = e.nextElement(); acc = <expression−avec−elem−et−acc>
                }
        }
```

Si l'ordre de parcours de la collection est connu lorsqu'il s'agit d'une `Sequence` (du premier au dernier élément de la séquence), rien n'est spécifié dans le cas d'un `Bag` ou

d'un `Set`. Il est donc possible d'écrire en OCL des expressions dont les résultats seront non déterministes (pour cause de sous-spécification).

L'opération `iterate` est importante car elle permet de construire la plupart des autres opérations OCL. Par exemple, l'opération `select` est spécifiée par :

```
context Set(T)::select(expr-avec-elem):Set(T)
        post: result = self->iterate(elem; acc : Set(T) = Set{} |
                        if expression-avec-elem
                                then acc->including(elem)
                                else acc
                        endif)
```

où l'expression `acc->including(elem)` renvoie un `Set` contenant tous les éléments de `acc` ainsi que `elem`.

De la même manière, le nombre d'éléments dans une collection est donné par l'opération prédéfinie `size` dont la spécification est la suivante :

```
context Collection::size:Integer
   post: result = self->iterate(elem; acc : Integer = 0 | acc + 1)
```

L'opération `forAll`, qui réalise l'opérateur $\forall$ sur une collection :

```
context Collection(T)::forAll(expr-avec-elem): Boolean
   post: result = self->iterate(elem; acc : Boolean = true | acc and expr-avec-elem)
```

L'opération `exists`, qui réalise l'opérateur $\exists$ sur une collection :

```
context Collection(T)::exists(expr-avec-elem): Boolean
   post: result = self->iterate(elem; acc : Boolean = false | acc or expr-avec-elem)
```

Typage lors de traversées successives de plusieurs propriétés

Dans le contexte d'une *Banque*, remarquons que le type du résultat de l'expression :

```
self . clients .compteBancaire
```

est `Bag(CompteBancaire)`. En effet, cette expression est, rappelons-le, un raccourci pour :

```
self . clients ->collect(c : Client | c.compteBancaire)
```

elle-même équivalente à l'expression :

```
self . clients ->iterate(c : Client ; acc : Bag(CompteBancaire) = Bag{} |
                 acc->union(c.compteBancaire))
```

Cette particularité est liée à la manière dont le type d'une navigation au travers d'une association est calculé. Nous avons indiqué que ce type dépendait de la multiplicité de l'association (*0..1*, *0..**) qui indique s'il s'agit d'un type scalaire ou d'une collection et des contraintes d'ordre sur cette association. Ainsi, la traversée d'une association *0..** retourne toujours un `Set`, sauf dans le cas où l'association est *{ordered}*. Dans ce cas, le résultat est une `Sequence`. Les résultats du type `Bag` apparaissent lors de la navigation successive de plusieurs associations *0..**. Intuitivement, la traversée d'une association *0..1* suivie de celle d'une association *0..** conduirait également à un `Set` plutôt qu'à un `Bag`. Ce typage est détaillé dans le tableau 3.21.

A — 1 — B	a.b : B
A — * — B	a.b : Set(B)
A — {ordered} * — B	a.b : Sequence(B)
A — * — B — 1 — C	a.b.c : Set(C)
A — {ordered} * — B — 1 — C	a.b.c : Sequence(C)
A — * — B — {ordered} * — C	a.b.c : Bag(C)
A — {ordered} * — B — {ordered} * — C	a.b.c : Sequence(C)

FIG. 3.21 – *Navigations multiples, associations et typage*

3.4 Généricité et héritage

3.4.1 La généricité (classes paramétrées)

La généricité permet de définir des modules paramétrés par un ou plusieurs types, ce qui est particulièrement utile pour les classes conteneurs comme les listes, tableau, et autres dictionnaires. La liste des paramètres génériques formels d'une telle classe est donnée dans une boite en pointillés venant s'enficher en haut et à droite de la représentation normale d'une classe (voir figure 3.22).

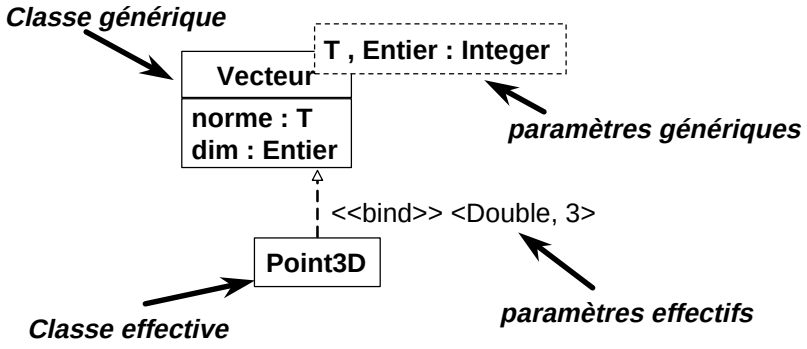


FIG. 3.22 – *Représentation de la généricité*

Une classe générique n'est pas instanciable directement car c'est seulement un patron pour définir une série de classes avec la même structure. Pour obtenir une classe effective à partir d'une classe générique, ou doit fournir des paramètres de types effectifs pour chaque paramètre générique comme illustré ci-dessus.

3.4.2 Héritage (généralisation)

L'héritage est ce qui caractérise les langages dit «orientés objets» par opposition aux langages dits «basés objets». L'héritage permet le partage d'attributs et d'opérations entre classes dans des relations hiérarchiques Super-classes/Sous-classes (voir figure 3.23).

L'héritage a une nature duale. L'héritage permet en effet d'une part la programmation par extension (par opposition à la programmation par ré-invention : une sous classe permet d'étendre un module en fournissant plus de fonctionnalités ; et d'autre part de représenter des relations *est une sorte de*, c'est-à-dire de faire de la *classification*. Le système de type des langages à objets est ainsi fondé sur l'héritage : la compatibilité de type pour l'affectation est définie suivant la relation d'héritage.

Extension de module

Au contraire de la généricité, qui permet la réutilisation de modules fermés, l'héritage permet de combiner et de customiser des éléments existants pour construire de nouvelles classes, sans perturber l'existant. Grâce à l'héritage, un module est toujours ouvert à

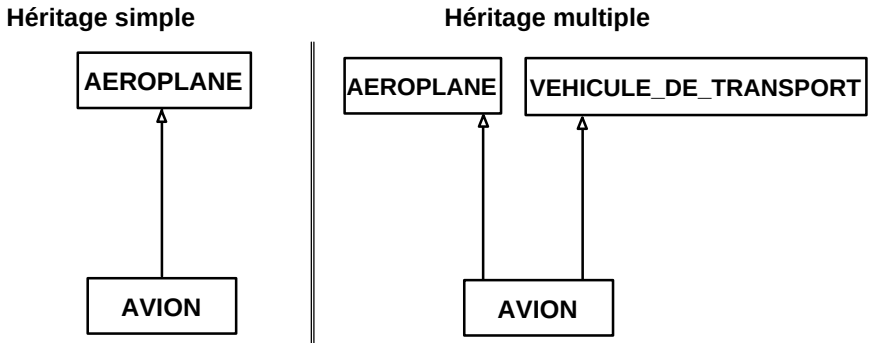


FIG. 3.23 – *Héritage (généralisation)*

de nouvelles modifications par sous classage.

Considérons l'exemple d'une classe RANDOM_GENERATOR fournissant une séquence pseudo-aléatoire de nombres réels compris entre 0 et 1. Si on a maintenant besoin d'entiers pseudo-aléatoires, on peut dériver de cette classe une nouvelle classe MULTI_RANDOM_GENERATOR ayant les mêmes fonctionnalités que RANDOM_GENERATOR plus une méthode `last_random_integer` ajoutant la fonctionnalité voulue sans perturber les programmes existant déjà.

Sous-typage

L'héritage fournit un moyen naturel de classifier des objets, c'est-à-dire d'expliciter la similarité et d'en tirer parti. Dans le cas le plus simple, si B hérite de A, alors B défini un sous-type de A dans le sens où un objet de type B peut-être utilisé partout où un type A est attendu (principe de substituabilité).

Le cas des classes génériques est un peu plus complexe car une classe générique définit un patron de type plutôt qu'un type effectif. On a alors la règle suivante (récursive) pour redéfinir la conformance de type entre classes génériques : une classe générique B[U] est un sous type de A[T] si et seulement si B est une sous classe de A et U de T. Cette règle est récursive car U et T peuvent elles-mêmes être des classes génériques. Cette règle elle facilement extensible dans le cas où de multiples paramètres génériques formels sont utilisés.

Bien sur la notion de classification est essentiellement subjective et ne peut être parfaite. C'est l'exemple célèbre des autruches qui sont des oiseaux qui ne peuvent pas voler.

> *Classification is humanity's attempt to bring a semblance of order to the description of an otherwise rather chaotic world. (B. Meyer)*

Il semble que la gestion de structures d'héritage imparfaites (où des sous classes ne sont pas des véritable sous types) un soit un problème incontournable que UML n'essaie pas

de résoudre, en laissant la sémantique précise de l'héritage sous-spécifiée de manière à ne pas rendre UML incompatible avec les langages à objets d'implantation comme Eiffel, Java, C++, ou C#.

3.4.3 Classes abstraites et interfaces

Une classe *abstraite* est une classe avec au moins une méthode *abstraite*, c'est-à-dire dont l'implantation est absente (voir figure 3.24).

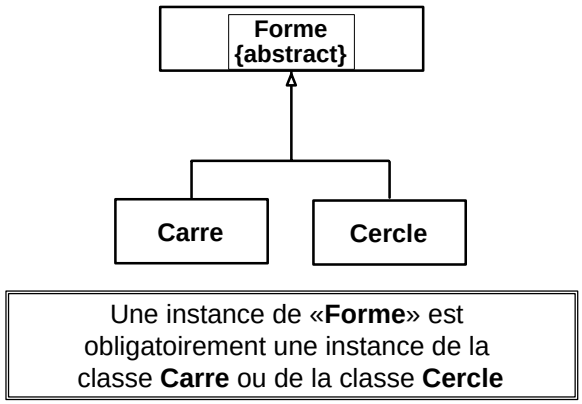


FIG. 3.24 – *Représentation de classes abstraites*

Les classes abstraites ne peuvent donc pas être instanciées, et servent à structurer un système en capturant les points communs entre classes, et en abstrayant leur différences. Cependant une classe abstraite définit un type (ou un patron de type s'il s'agit d'une classe abstraite générique) et des entités peuvent tout à fait être déclarées de ce type.

Une méthode abstraite (*deferred* en Eiffel, *pure virtual* en C++, *abstract* en Java ou en C#) a une spécification (un nom, une signature, et possiblement des préconditions et postconditions), mais pas d'implantation (voir figure 3.25).

On appelle *Interface* une classe qui ne possède pas d'attributs d'instances et dont toutes les méthodes sont abstraites, et UML permet de les représenter sous forme d'une « lollipop » qui suggère qu'un composant offrant (implantant) cette interface peut être enfiché («pluggué») dans un réceptacle qui a besoin de cette interface pour fonctionner (voir figure 3.26).

3.4.4 Héritage des relations et des contrats

Les relations sont héritées par les sous classes : (voir figure 3.27).

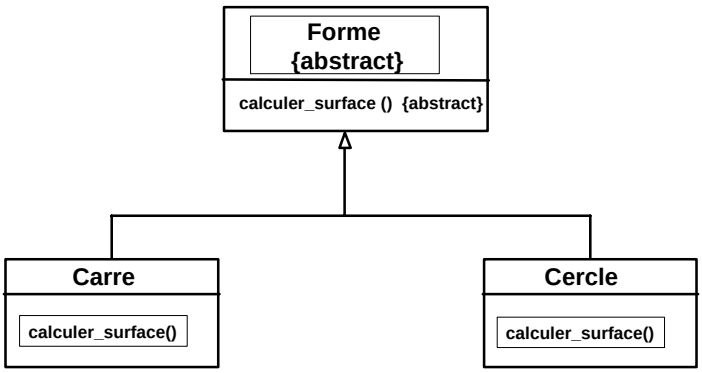


FIG. 3.25 – *Représentation des opérations abstraites*

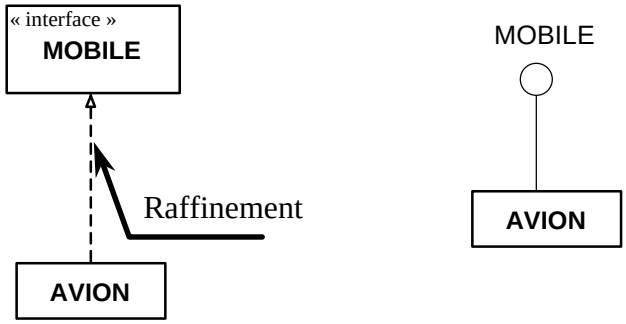


FIG. 3.26 – *Interfaces et « lollipop »*

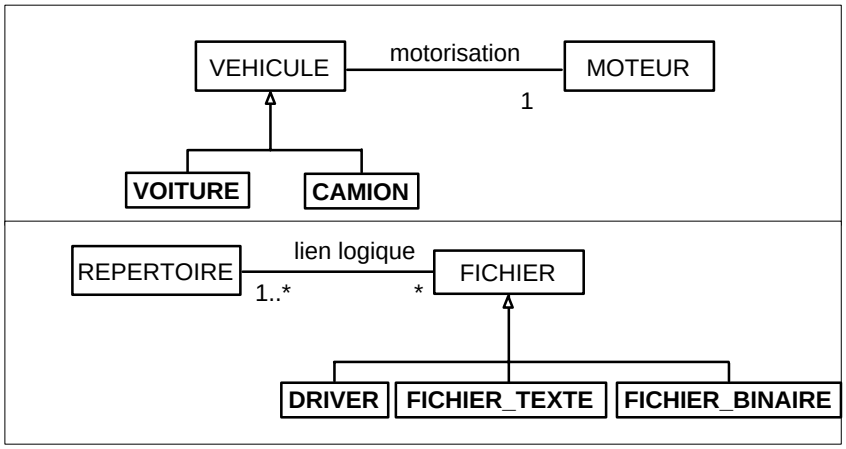


FIG. 3.27 – *Héritage des relations*

De même, il y a héritage des contrats, car pour pouvoir substituer une sous-classe à sa super-classe (Liskov Substituability Principle), il faut que la sous-classe respecte les contrats de sa super-classe (analogie avec la notion de sous-contactance ou sous-traitance: il ne faut pas que cela soit visible par le client). Ceci implique que dans la sous-classe, les pré-conditions pourront seulement être affaiblies (demander moins au client), et les post-conditions renforcées (offrir plus au client).

3.4.5 Polymorphisme et liaison dynamique

Le polymorphisme est la possibilité de changer de forme.

Le polymorphisme en UML est un polymorphisme de référence ; c'est-à-dire qu'une entité `x` déclarée d'un type non primitif T peut-être utilisée pour référencer un objet de type S (pourvu que la classe S soit descendante de la classe T). L'entité `x` est dite de type *statique* T, et capable d'assumer le type *dynamique* S.

Une entité peut acquérir un nouveau type dynamique soit :

- au moment de sa création,
- ou au travers une instruction d'affectation et à l'occasion des passages de paramètres effectifs vers paramètres formels lors des appels de méthode.

Liaison dynamique

La liaison dynamique des méthodes aux entités un mécanisme qui détermine quelle méthode doit être effectivement appelée en fonction du type dynamique de l'objet cible

et du nom de la méthode, en prenant en comptant une redéfinition possible le long des différents chemins d'héritage : l'effet de l'appel d'une opération d'un objet dépend de sa forme effective à l'exécution.

Si l'on a :

```
f : FORME; c: CERCLE; k: CARRE;
f := c; f := k
```

alors, l'appel `f.imprimer` donnera un résultat différent selon que f est CERCLE ou CARRE.

Ce mécanisme est en général réalisé à l'exécution car le type dynamique de la cible n'est pas toujours calculable à la compilation. Cependant, si une primitive n'est jamais redéfinie, ou bien si le type dynamique de la cible peut être connu à la compilation (c'est le cas par exemple des types feuilles, c'est-à-dire sans sous-classe) alors rien n'empêche le compilateur de remplacer la liaison dynamique par l'appel direct de la primitive au travers d'un simple appel de procédure, voire même d'une expansion du code en ligne.

Ce court-circuitage de la liaison dynamique reste dans tous les cas une optimisation de compilation qui est complètement transparente sémantiquement.

Outre le fait de permettre un espace de nommage réduit et uniforme, le polymorphisme allié à la liaison dynamique permet de laisser les systèmes logiciels ouverts aux modifications sans perturber l'existant, et ceci même après déploiement de l'application. Par exemple si un conteneur T contient des *Formes* (en fait des instances de sous-classes de *Forme*), alors on pourra écrire des programmes de type

```
Pour tout i faire T[i].imprimer
```

sans avoir à énumérer du coté client (traitement au cas par cas par une instruction de type *case* ou *switch*) chacun des types possibles pour `T[i]`.

Si un nouveau besoin apparaît, par exemple le fait de devoir prendre en compte un nouveau type de formes (des triangles), il suffit d'introduire une nouvelle classe *Triangle* héritant de *Forme*, d'y définir l'implantation ad hoc pour les méthodes déclarées dans *Forme* (comme `imprimer, déplacer, redimentionner`), et le tour est joué sans avoir besoin de faire des modifications globales partout où les traitements au cas par cas seraient apparus dans une programmation classique (voir figure 3.28).

Cette absence de traitements au cas par cas (rendus obsolètes et systématiquement remplacés par l'utilisation de l'héritage et de la liaison dynamique) est la caractéristique des logiciel conçus par objets, et permet d'obtenir une meilleure continuité entre le domaine du problème et le domaine de la solution (comme défini au chapitre introductif) car un petit changement dans le domaine du problème (par exemple ajout du concept de triangle) ne nécessitera plus qu'une modification locale (introduire une nouvelle classe *Triangle* héritant de *Forme*) et non plus potentiellement proportionnelle à la taille du logiciel (rechercher dans tout le logiciel les traitements au cas par cas sur les formes et modifier chacun d'entres eux pour leur faire prendre en compte la notion de triangle).

- T contient des FORMEs
  - en fait des instances de sous-classes de FORME
- Programmes de type :
  - T[1] <- carré
    T[2] <- cercle
    ...
    Pour tout i
      T[i].imprimer
- Si ajouts ultérieurs:
  - e.g. triangle
- Pas de modifications globales
  - case-less programming

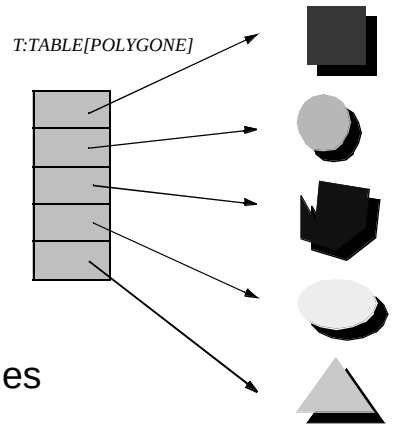


FIG. 3.28 – *Polymorphisme : exemple*

3.5 Compléments de modélisation

3.5.1 Les stéréotypes

Les stéréotypes permettent d'introduire de nouveaux éléments de modélisation instanciant soit :

- Des classes du méta modèle UML (pour les stéréotypes de base UML)
- Des extensions de classes du méta modèle UML (pour les stéréotypes définis par l'utilisateur)

Il peuvent être attachés aux éléments de modélisations et aux diagrammes (voir figure 3.29) : Classes, objets, opérations, attributs, généralisations, relations, acteurs, uses-cases, événements, diagrammes de collaboration ...

3.5.2 Les notes

Les notes permettent d'apporter des compléments de modélisation (c'est l'équivalent des commentaires dans un langage de programmation). Elles sont attachés à un élément du modèle ou libre dans un diagramme, et exprimés sous forme textuelle(voir figure 3.30). Elles peuvent être typées par des stéréotypes.

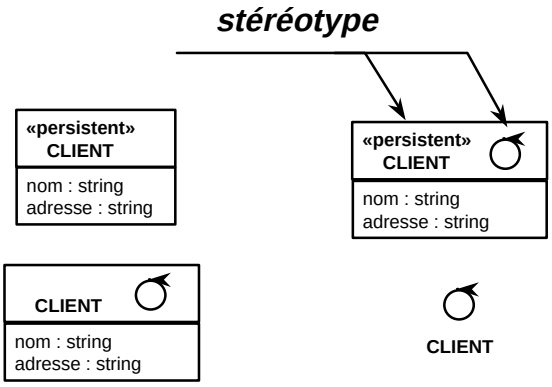


FIG. 3.29 – *Notations pour les stéréotypes*

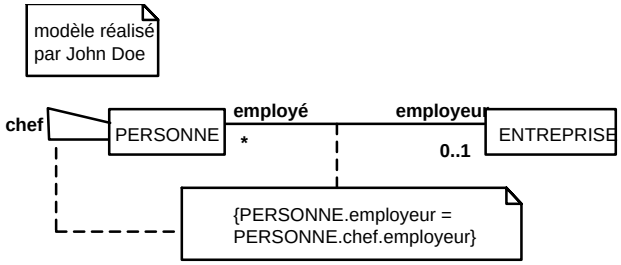
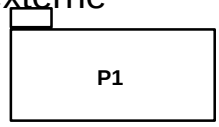


FIG. 3.30 – *Les notes*

3.6 Les paquetages (package)

Le package est un élément de structuration des classes qui apporte une modularisation à l'échelle supérieure (voir figure 3.31).

- Vue graphique externe



- Vue graphique externe et interne

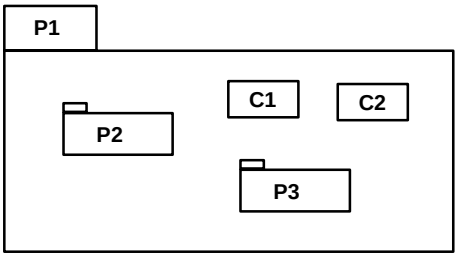


FIG. 3.31 – *Représentation d'un package*

Un package partitionne l'application : il référence ou contient des classes de l'application, et il référence ou contient d'autres packages (analogie package/classe avec répertoire/fichier dans les systèmes d'exploitation, voir figure 3.32).

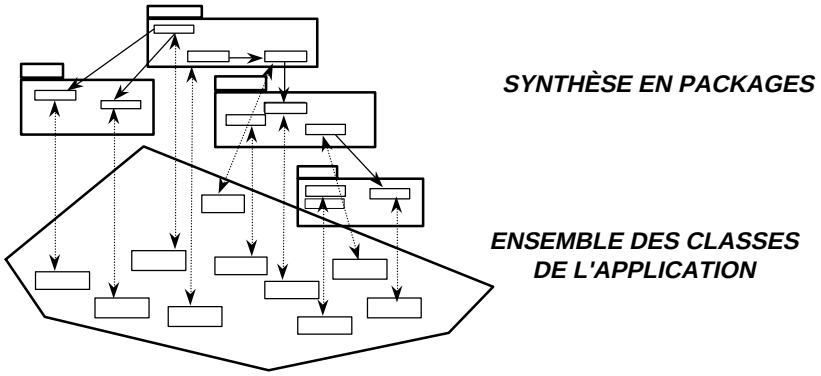


FIG. 3.32 – *Partitionnement d'une application*

Un package réglemente la visibilité des classes et des packages qu'il contient. Les

packages sont liés entre eux par des liens d'utilisation, de composition et de généralisation. Un package est la représentation informatique du contexte de définition d'une classe.

Il y a utilisation entre packages si des classes du package utilisateur accèdent à des classes du package utilisé. Pour qu'une classe d'un package p1 puisse utiliser une classe d'un package p2, il doit y avoir au préalable une déclaration explicite de l'utilisation du package p2 par le package p1 (voir figure 3.33).

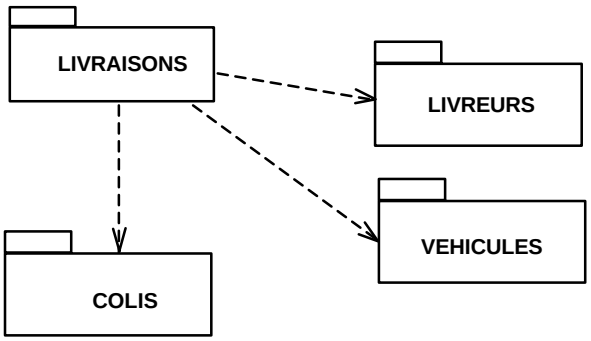


FIG. 3.33 – *Utilisation entre packages*

La notion de packages apporte des réponses aux besoins :

- Contexte de définition d'une classe
- Unité de structuration
- Unité d'encapsulation
- Unité d'intégration
- Unité de réutilisation
- Unité de configuration
- Unité de production

Chapitre 4

Axe fonctionnel

4.1 Cas d'utilisations

Le problème de l'expression des besoins a été un sujet longtemps négligé par les méthodes d'analyse et conception par objets (e.g. OMT), d'où la dérive de certains projets vers le « conceptuel », c'est à dire une perte de vue des objectifs du projet. La question de l'expression des besoins est pourtant fondamentale, et souvent pas si facile (cible mouvante). Dans son livre *Object-Oriented Software Engineering*, Ivar Jacobson a proposé pour cela la technique des acteurs et des cas d'utilisation, qui a été intégrée a UML.

Les cas d'utilisation ont pour objectif de :

- Se comprendre
- Représenter le système
- Exprimer le service rendu
- Décrire la manière dont le système est perçu

Pour cela, on imagine le système « fini », et on essaye de montrer comment on interagit avec lui. Un diagramme de cas d'utilisation représente de manière synthétique les interactions d'un système avec son environnement. Il peut être utilisé en amont d'une démarche de développement par objets pour transposer les exigences d'un cahier des charges sous une forme graphique.

4.1.1 Acteurs

Un *acteur* est une entité externe au système et amenée à interagir avec lui. Un acteur «joue un rôle» vis-à-vis du système. Un acteur peut représenter le rôle joué par une catégorie d'êtres humains (caissier, client, manager), ou par un autre système. L'identification des acteurs permet par définition de délimiter les frontières du système.

Un acteur est représenté en UML par une classe stéréotypée «Actor» (voir figure 4.1).

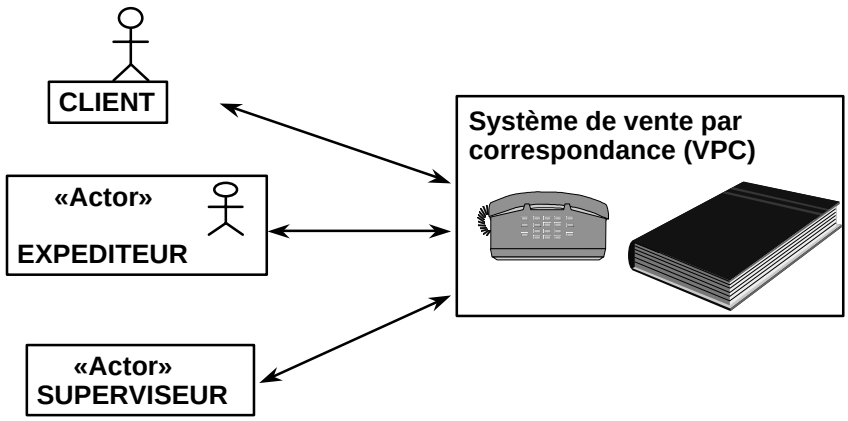


FIG. 4.1 – *Acteurs : notations*

4.1.2 Les cas d'utilisation (use-cases)

Un cas d'utilisation est une manière particulière d'utiliser le système. Il abstrait une séquence d'interactions entre le système et un ou plusieurs acteurs, qui s'expriment par des diagrammes de séquences (voir paragraphe 5.2). La compilation des cas d'utilisation décrit de manière informelle le service rendu par le système : ils fournissent une expression "fonctionnelle" du besoin (voir figure 4.2).

4.1.3 Relations sur les use-cases

Deux sortes de relations sont possibles entre cas d'utilisation :

Utilisation («uses») Utilisation d'autres use-cases pour en préciser la définition

Extension («extends») Un use-case étendu est une spécialisation du use-case père

La notation pour les relations sur les use-cases est illustrée en figure 4.3.

4.1.4 Intérêt des use-cases

> *There is a magic number: 7, plus or minus 2. This refers to the number of concepts that we humans can keep in mind at any one time. (H. A. Miller, 1958)*

L'intérêt des cas d'utilisation est de donner une vision générale sur le service rendu par le système afin d'éviter de partir dans toutes les directions pour le reste de la

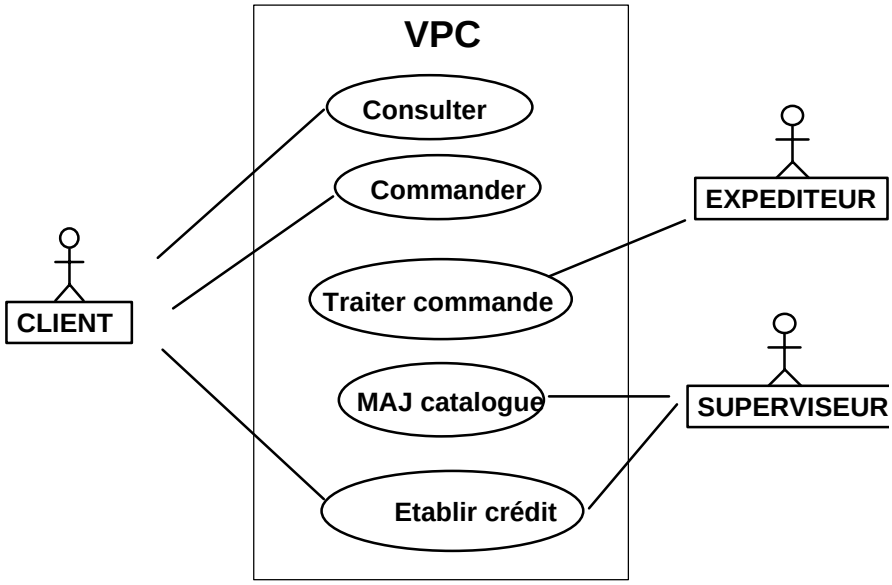


FIG. 4.2 – *Cas d'utilisation : exemple et notation*

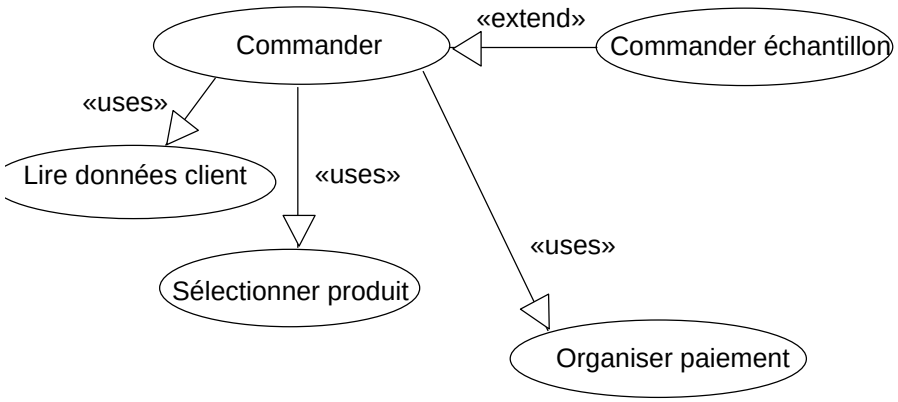


FIG. 4.3 – *Relations sur les use-cases : notation*

modélisation : il importe donc de se concentrer sur les besoins fondamentaux. Suite aux recherches de Miller en psychologie humaine, il semble que pour comprendre un système dans sa globalité (i.e. en avoir une vue holistique), il ne faut pas le décomposer trop finement. On considère ainsi que quelque soit la taille du système, on doive se limiter à seulement de 3 à 10 cas d'utilisation, ce qui en fonction de la taille du système conduira à abstraire plus ou moins chaque cas d'utilisation.

On pourra bien-sûr envisager une représentation détaillée par raffinement des cas d'utilisation. Ce serait le début de décomposition fonctionnelle (i.e. faire du SADT en UML). Le point clé est ici de ne pas aller trop loin dans cette décomposition (ça ne sert à rien) et surtout de ne pas architecturer le système selon cette logique (voir paragraphe 1.4.3).

On mentionnera pour finir que les cas d'utilisation sont utiles pour l'établissement de scénarios (qui sont vus comme des instances de use-cases), et donc pour préparer les tests fonctionnels de l'application (voir figure 4.4).

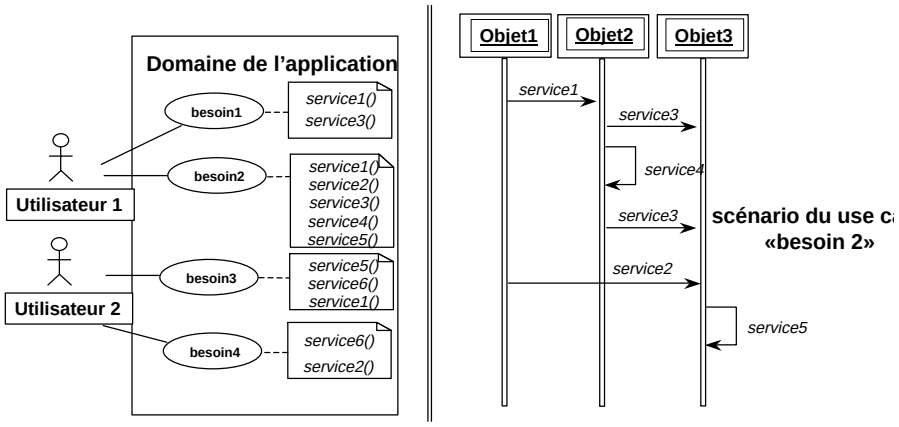


FIG. 4.4 – *Utiles pour l établissement de scénarios*

Chapitre 5

Axe dynamique du système

5.1 Introduction

Jusqu'ici, le système modélisé a été décrit statiquement: les diagrammes de classe par exemple décrivent les messages (méthodes ou opérations) que les instances des classes peuvent recevoir mais ne décrivent pas l'émission de ces messages. Elle ne montrent pas le lien entre ces échanges de messages et les processus généraux que l'application doit réaliser. L'axe dynamique permet donc de décrire comment le système évolue dans le temps.

On se focalise d'abord sur les collaborations entre objets. Rappelons que le principe de l'approche objet consiste à avoir des objets les plus simples possibles (unités autonomes de connaissance), et que la gestion de la complexité se fait par collaborations entre ces objets simples. La notion UML de collaboration (au niveau instances) est une vue sur un sous-ensemble des objets impliqués dans un sous-ensemble d'un traitement (voir figure 5.1).

UML offre deux sortes de diagrammes pour représenter les collaborations :

- Diagrammes de séquences (scénarios)
- Diagrammes de collaboration

5.2 Diagrammes de séquences (scénarios)

Les diagrammes de séquences d'UML dérivent des scénarios de OMT : ils montrent des exemples de coopération entre objets dans la réalisation de processus de l'application. Ils illustrent la dynamique d'enchaînement des traitements à travers les messages échangés entre objets, avec le temps représenté comme une dimension explicite, en général du haut vers le bas.

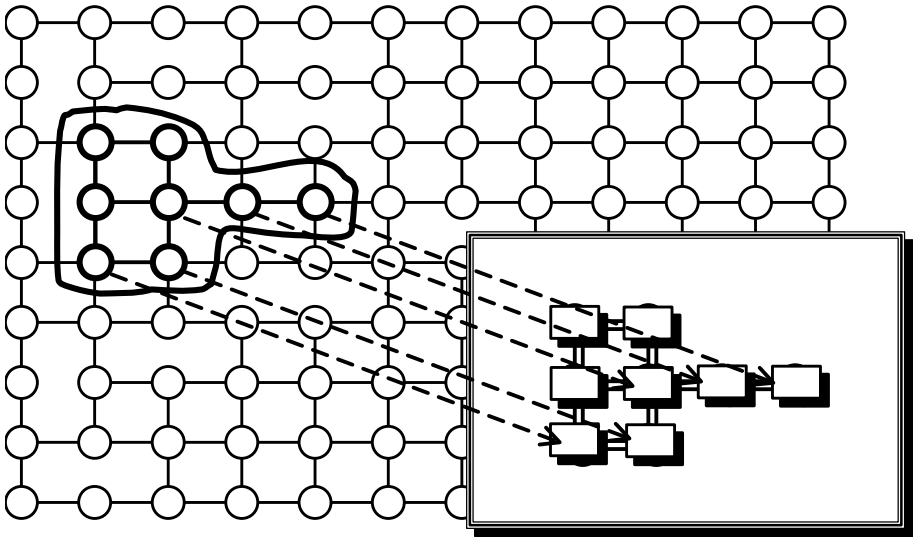


FIG. 5.1 – *Collaborations (au niveau instances)*

Les éléments constitutifs d'un scénario sont :

- Un ensemble d'objets (et/ou d'acteurs)
- Un message initiateur du scénario
- La chronologie des messages échangés subséquemment
- Les contraintes de temps (aspects temps-réel)

La syntaxe graphique est présentée en figure 5.2.

La «ligne de vie» représente l'existence de l'objet à un instant particulier. Elle commence avec la création de l'objet, et se termine avec la destruction de l'objet. L'activation est la période durant laquelle l'objet exécute une action lui-même ou via une autre procédure (voir figure 5.3). Les messages abstraient la notion de communication entre objets. Ils peuvent porter des paramètres, ou matérialiser un retour d'opération.

Quelques cas particuliers de messages sont aussi présentés en figure 5.3:

- Les messages entraînant la construction d'un objet
- La récursion
- Les destructions d'objets

De plus, comme un scénario se lit de haut en bas dans le sens chronologique d'échange des messages, des contraintes temporelles peuvent facilement être ajoutées au scénario (voir figure 5.4).

Les diagrammes de séquence permettent aussi des représentations plus complexes, comme des conditionnelles (voir figure 5.5).

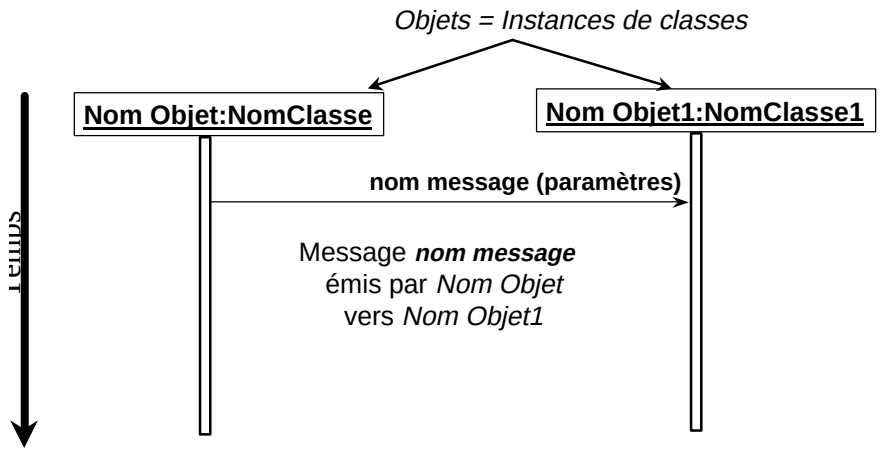


FIG. 5.2 – *Syntaxe graphique pour les diagrammes de séquence*

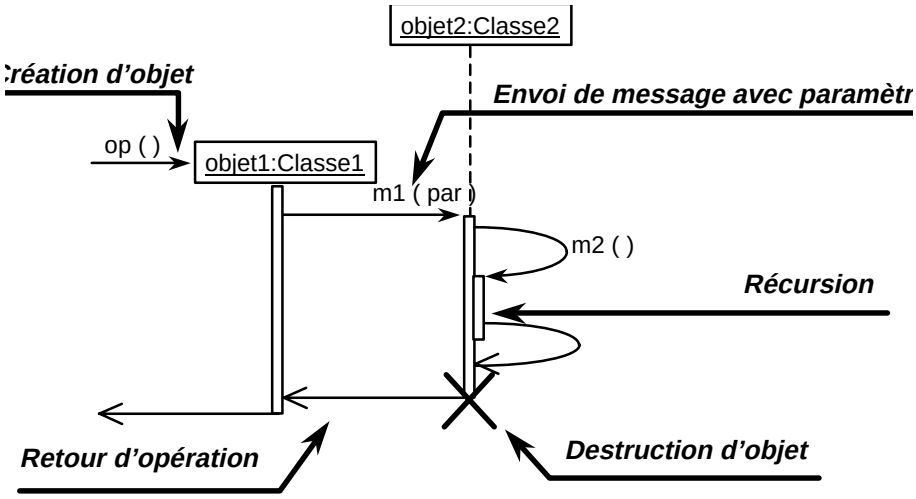


FIG. 5.3 – *Notations pour les diagrammes de séquence*

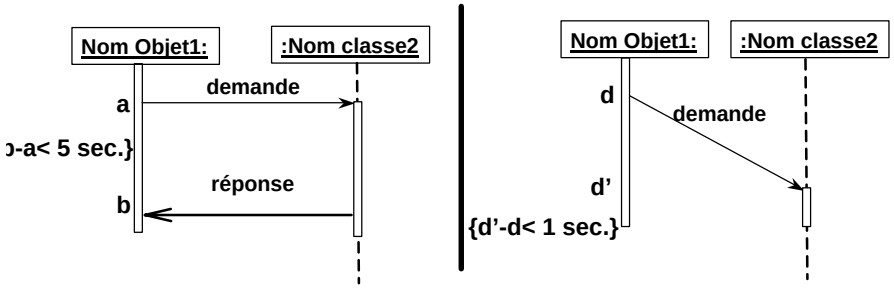


FIG. 5.4 – *Aspects asynchrones et temps réel*

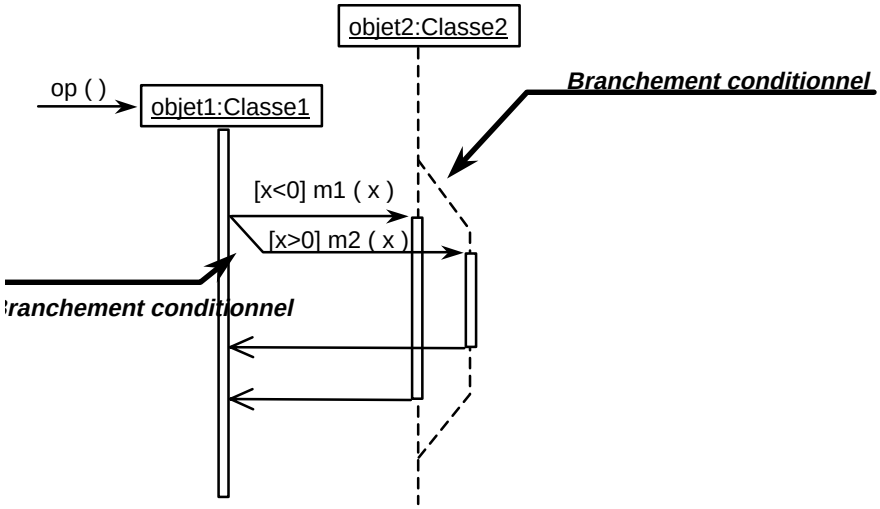


FIG. 5.5 – *Représentation de conditionnelles*

Cependant, leur intérêt principal est leur simplicité qui permet une compréhension intuitive de la dynamique d'une fraction de l'application par des non-experts. Aussi, il est recommandé de s'en tenir autant que possible à des diagrammes simples.

5.3 Diagrammes de collaboration

Comme les scénarios, les diagrammes de collaborations montrent des exemples de coopération entre des objets dans la réalisation de processus de l'application. Mais alors que les scénarios illustrent la dynamique d'enchaînement des traitements d'une application en introduisant la dimension temporelle, dans les diagrammes de collaborations en revanche, la dimension temporelle est représentée par numéros de séquence, ce qui laisse libre les 2 dimensions de l'écran (ou du papier) pour la représentation des objets et de leurs liens (voir figure 5.6).

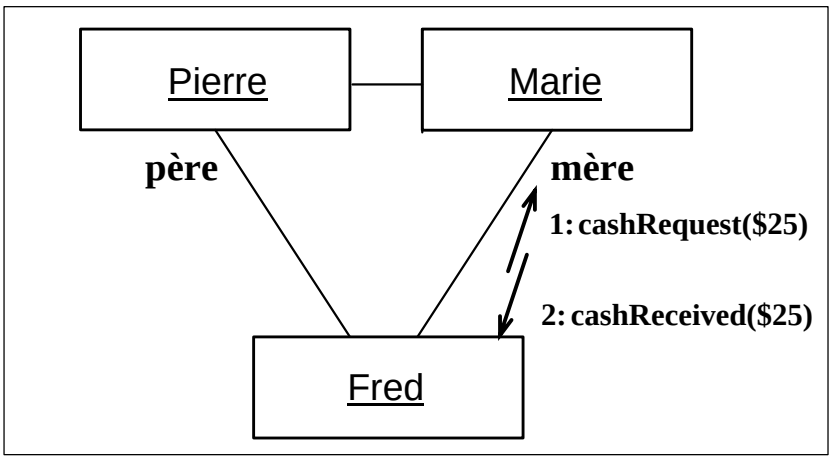


FIG. 5.6 – *Représentation d'une collaboration (niveau instance)*

Notons que ce diagramme de collaboration est équivalent au diagramme de séquence présenté en figure 5.7.

Les diagrammes de collaboration peuvent être attachés à :

- Une classe
- Une opération
- Un use-case

Ils s'appliquent aussi bien en spécification qu'en conception (illustration de design patterns).

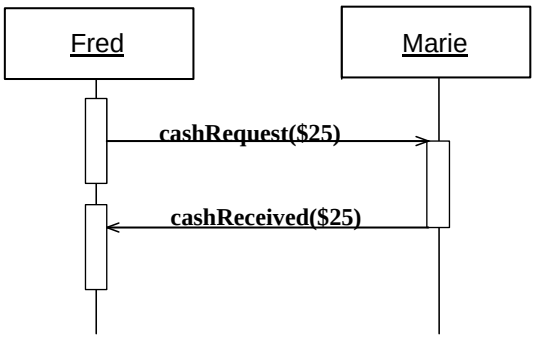


FIG. 5.7 – *Diagramme séquence équivalent*

Les éléments constitutifs des diagrammes de collaboration sont :
- Un acteur,
- Un ensemble d'objets, d'attributs et de paramètres,
- Des relations entre ces objets,
- Un message initiateur du diagramme provenant d'un acteur de l'application,
- Des interactions, portant des numéros de séquence des messages échangés entre les objets de cet ensemble suite au message initiateur.

Les questions auxquelles répondent les collaborations sont donc :
- Quel est l'objectif ?
- Quels sont les objets ?
- Quelles sont leurs responsabilités ?
- Comment sont-ils inter-connectés ?
- Comment interragissent-ils ?

5.4 Les diagrammes d'états

5.4.1 Introduction

On a vu comment modéliser à l'aide des diagrammes d'interactions un ensemble d'objets réagissant à un stimuli particulier. Réciproquement, on va maintenant voir comment modéliser toutes les interactions possibles d'un seul objet aux stimuli qui le concernent à l'aide des diagrammes d'états.

Un diagramme d'état est attaché à une classe, pour modéliser pour chacune de ses instances les dépendances qu'il y a entre leurs états et leurs réactions aux messages et autres événements qu'il reçoivent. Cela permet de modéliser une programmation réactive, ou « temps réel ».

Un diagramme d'état pour une classe peut-être vu comme une généralisation de tous les scénarios qui concernent les instances de cette classe, afin de produire une description systématique des réactions d'un objet aux changements de son environnement. Ils se présentent comme un réseau d'états et de transitions décrivant des automates étendus (connu sous le nom de Diagrammes de Harel). Dans leur syntaxe graphique (voir figure 5.8) :

les états sont représentés comme des boîtes avec des coins arrondis,

les transitions entre états sont représentés par des flèches,

les événements qui causent les transitions entre états sont représentés par une étiquette sur cette transition, avec la syntaxe :

```
événement (param:type,...) [conditions] / action; événements provoqués
```

l'état initial est désigné par une flèche partant d'un gros point noir.

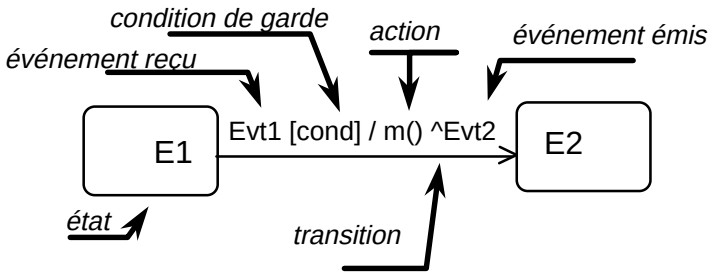


FIG. 5.8 – *Syntaxe graphique : diagramme d'états*

5.4.2 Notion d'événements

La notion d'événements permet de généraliser la notion de stimulis auxquels réagissent les objets (voir figure 5.9).

L'occurrence d'un événement déclenche une transition d'état : un événement est donc une abstraction d'une information instantanée échangée entre des objets et/ou des acteurs.

Un événement est instantané et correspond à une communication unidirectionnelle. Un objet peut réagir à certains événements lorsqu'il est dans certains états.

UML considère quatre sortes d'événements :

- Réception d'un appel d'opération par un objet, transcrit comme un événement déclenchant sur la transition.
- Réalisation d'une condition arbitraire (par exemple la température d'une chaudière devient supérieure à 65 degrés), qui se transcrit simplement par une condition de garde sur la transition.

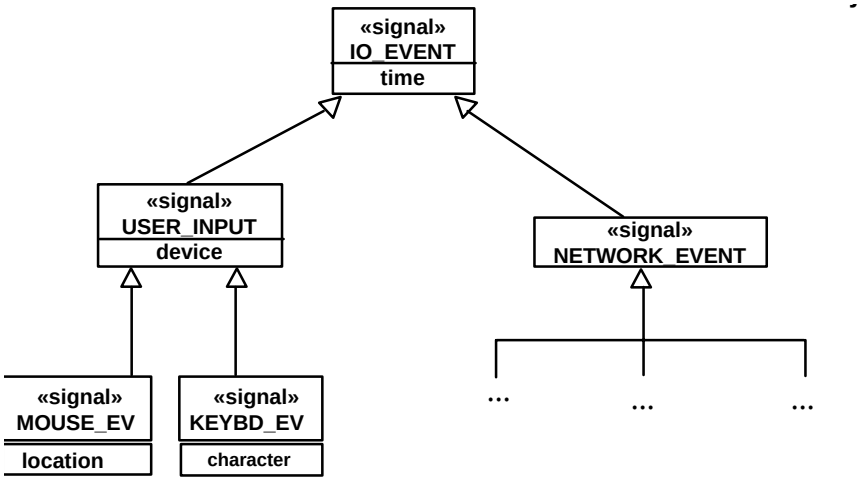


FIG. 5.9 – *Les événements*

- Réception d'un signal (instance d'une classe stéréotypée «signal») issu d'un autre objet transcrit en un événement déclenchant sur la transition.
- Période de temps écoulée, transcrit comme une expression du temps sur la transition (précédé du mot-clé *after*).

5.4.3 Notion d'action

Une action est une opération instantanée (conceptuellement) et atomique (ne peut être interrompue). Elle est déclenchée par un événement, et décrit le traitement associé à une transition (ou à l'entrée dans un état ou à la sortie de cet état, voir figure 5.10).

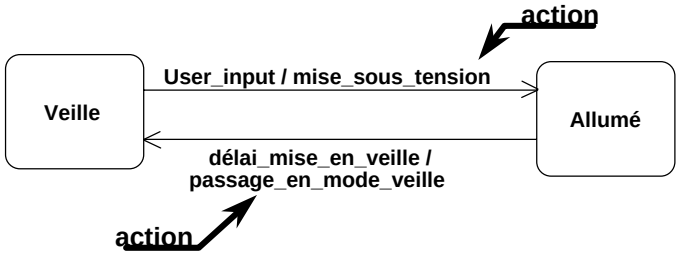


FIG. 5.10 – *Notion d'action*

5.4.4 Notion d'états

Un Etat est situation stable d'un objet parmi un ensemble de situations prédéfinies, qui conditionne la réponse de l'objet à des événements.

Un état peut avoir des variables internes, qui sont les attributs de la classe supportant ce diagramme d'états.

Comme un diagramme d'état plat devient rapidement illisible (voir figure 5.11) dès que la dynamique des traitements associés un objet devient complexe (explosion combinatoire des transitions), UML permet de structurer un état en sous-états représentés par imbrication graphique (voir figure 5.13).

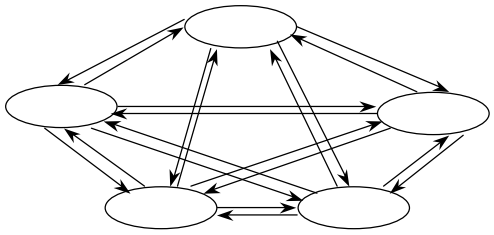


FIG. 5.11 – *Nécessité de structuration en sous-états*

5.4.5 Notion d'activité dans un état

L'Activité d'un état est une opération (facultative) se déroulant continuement tant qu 'on est dans l'état associé. Quitter l'état et terminer l'action sont alors synonymes. On le représente à l'aide du mot-clé *do/* suivi du nom de l'action directement à l'intérieur de la boîte représentant l'état (voir figure 5.12). Formellement, une telle activité est équivalente à 2 événements *démarrer l'activité* et *terminer l'activité* qui seraient sur respectivement toutes les transitions entrantes et sortantes de l'état en question.

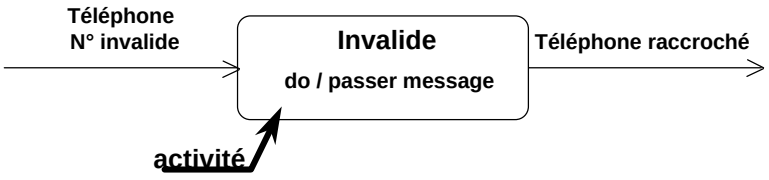


FIG. 5.12 – *Notion d'activité dans un état*

La figure 5.13 représente un exemple de diagramme d'états relativement sophistiqué, celui d'un téléphone.

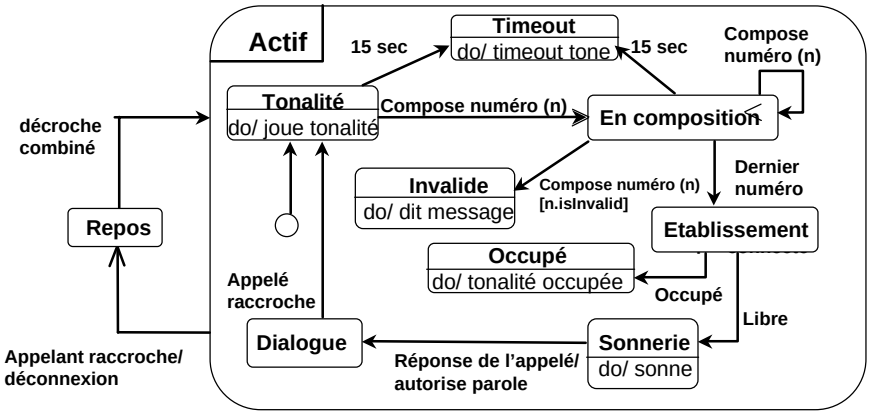


FIG. 5.13 – *Exemple de diagramme d'états*

5.4.6 Diagrammes d'états concurrents

Pour des raisons de compatibilité avec les diagrammes de Harel, UML permet l'utilisation de sous-états concurrents pour ne pas à avoir à expliciter le produit cartésien d'automates si 2 ou plus aspects de l'état d'un objet sont indépendants (par exemple l'objet est traversé par des activités parallèles et indépendantes). Les sous-états concurrents sont graphiquement séparés par pointillés, et forment des lignes de natation (« swim lanes ») comme illustré en figure 5.14.

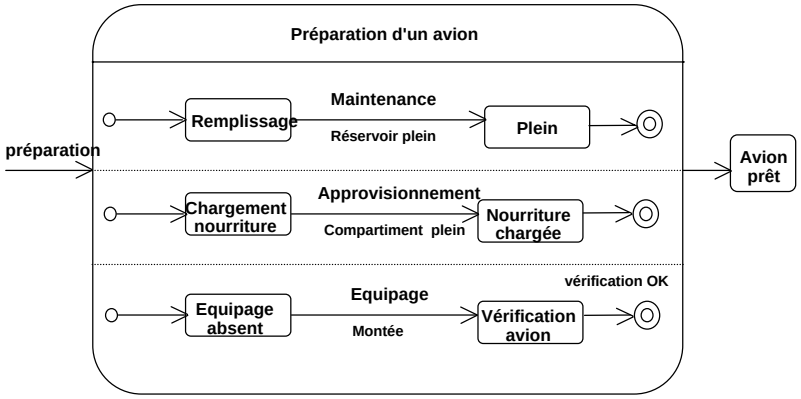


FIG. 5.14 – *Exemple de concurrence*

Notons cependant que l'utilisation de la notion de sous-états concurrents est en général révélatrice d'une médiocre décomposition par objet. Dans l'exemple de la figure 5.14, il eut mieux valu modéliser l'avion comme une agrégation de trois objets (un réservoir, un approvisionnement et un équipage) ayant chacun un automate simple à 2 états, l'automate résultant étant obtenu par composition implicite des automates agrégés. De plus, en passant, cela permettrait de modéliser facilement des avions à plusieurs réservoir (cardinalité 2, 3 ou même *) sur la relation vers la classe *réservoir*) sans avoir à expliciter graphiquement toutes les combinaisons comme on l'aurait fait avec des diagrammes de Harel.

5.5 Les diagrammes d'activité

Les diagrammes d'activité décrivent comment des activités sont coordonnées (description d'un flot de contrôle procédural). Par exemple, un diagramme d'activité pourrait être utilisé (comme un diagramme d'interaction) pour montrer comment une opération pourrait être implantée. Un diagramme d'activité sera particulièrement utile lorsque l'on sait qu'une opération aura à accomplir tout un ensemble de choses, et qu'on veut modéliser les dépendances essentielles entre celles-ci, avant de prendre une quelconque décision de conception sur l'ordre précis dans lequel elles devront s'accomplir.

Les diagrammes d'activités sont aussi utiles pour décrire comment des cas d'utilisation individuels peuvent se dérouler et peuvent dépendre d'autres cas d'utilisation. En effet, dans certaines situations, les cas d'utilisation peuvent ne pas pouvoir se produire dans un ordre quelconque, mais dépendre du flot de travail global (workflow) des utilisateurs.

Les diagrammes d'activité se présentent comme des réseaux d'actions et de transitions (voir figure 5.15) : ce sont formellement des automates dégénérés où les transitions entre états ne s'effectuent jamais sur événement asynchrone, mais seulement lorsque l'opération en cours est terminée. Un état est donc en fait un *état-action*, c'est à dire un raccourcis pour un état où il y a :

- une action interne
- au moins une transition sortante
- production d'un événement implicite : action accomplie

Dans un diagramme d'activités, il n'y a pas de production/réaction à des événements explicites.

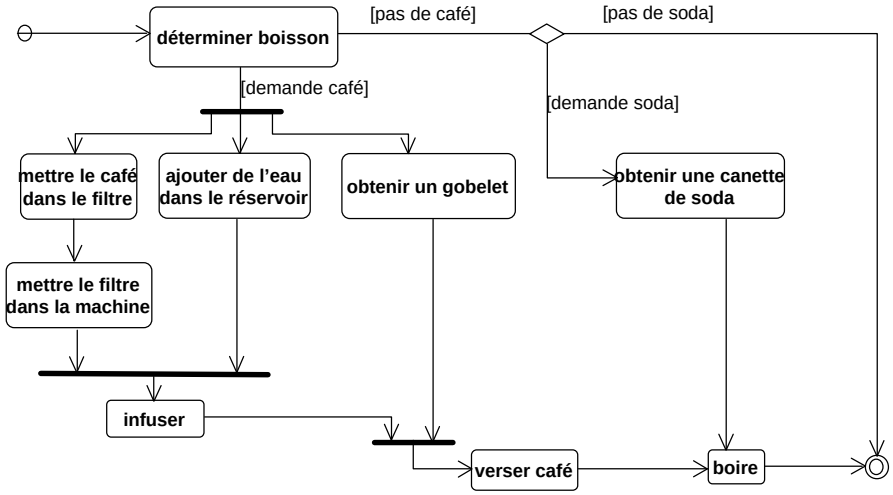


FIG. 5.15 – *Exemple de diagramme d activité*

Chapitre 6

Diagrammes d'implantation et de composants

6.1 La vision limité d'UML 1

Dans la version 1 d'UML on trouve seulement des diagrammes de composants (voir figure 6.1) et des diagrammes de déploiement (voir figure 6.2), ce qui est assez faible compte tenu de la structure actuelle des applications, de plus en plus souvent forgées à base de composants répartis.

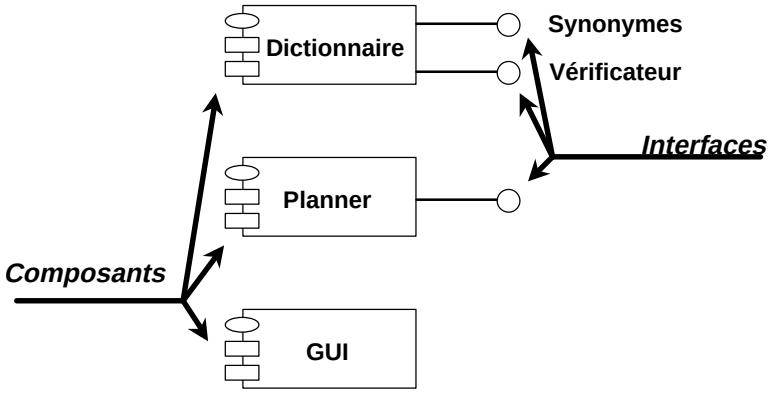


FIG. 6.1 – *Exemples de composants*

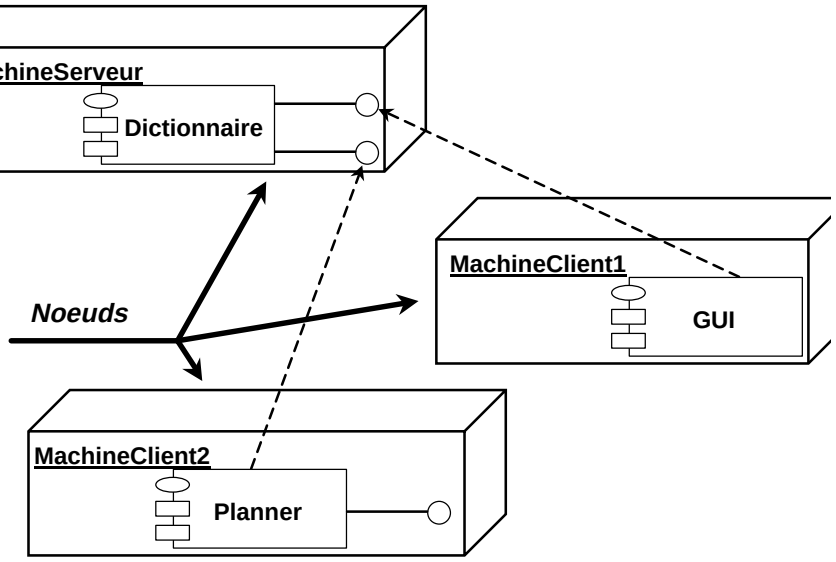


FIG. 6.2 – *Exemple de déploiement*

6.2 Concepts avancés

Le langage UML a récement fait l'objet d'une révision majeure appelée UML 2.0 que nous présentons succintement ci-dessous. Celle-ci n'est cependant pas encore largement supportée par les outils de génie logiciel, c'est pourquoi nous n'entrerons pas beaucoup plus dans le détail, sauf pour la notion de composant qui comme on l'a vu ci-dessus était vraiment très limitée en UML1.4.

6.2.1 De UML1 à UML 2.0

Lors de l'évolution de UML1 à UML2.0, les concepteurs d'UML2.0 ont cherché à améliorer la précision sémantique des modèles tout en clarifiant les concepts. Ils ont aussi travaillé sur l'alignement entre le MOF et UML. Cela a amené des changements majeurs dans l'architecture interne d'UML. En outre, ils ont amélioré la plupart des diagrammes. On notera en particulier la consolidation du modèle d'action sous jacent aux diagrammes comportementaux.

UML2 apporte entre autres les nouvelles fonctionnalités suivantes :

- Sémantique et syntaxe simplifiée.
- Meilleure organisation de la structure du langage.
- Amélioration de la structure des diagrammes de séquence.
- Amélioration du support pour les modèles exécutables.

- Meilleure modélisation des comportements.
- Meilleur mécanisme d'extension (grâce à l'addition de métaclasses pour définir des profils UML spécifiques au domaine).
- Ajout d'un support pour le développement orienté composant.

Les modifications suivantes ont été apportées dans le langage (liste non exhaustive) :

- Ajout de 4 nouveaux diagrammes : Diagrammes de composite (" composite structure diagram "), diagramme d'interaction (" interaction overview diagram "), diagramme temporel (" timing diagram ") et diagramme de paquetage " (Package diagram ").
- Certains diagrammes ont été renommés. Les diagrammes de collaboration de UML1.5 sont maintenant des diagrammes de communication en UML2.
- Dans les machines à état : ajout des nouveaux types d'états, point d'entrée, point de sortie et de pseudo état terminal.
- Les machines à état peuvent être étendues et les transitions redéfinies.
- Les paquetages peuvent être fusionnés (" package merge ").
- Des notions de contraintes de temps, expression du temps, d'intervalle de temps et d'observation du temps ont été ajoutées.
- De nouvelles notations ont été ajoutées pour alléger la présentation des partitions des diagrammes d'activités lorsque celles-ci sont trop complexes.

6.2.2 Composants et connecteurs

Le standard UML 2.0 propose un modèle de composant nettement amélioré par rapport à la version UML 1. Dans ce paragraphe, nous présentons les traits particuliers des composants en UML 2. Le méta-modèle d'UML 2 définit les entités suivantes dans les packages Structure::Components et Structure::Ports.

Un *composant* (component) est une entité instanciable qui interagit avec son environnement par l'intermédiaire de port et qui a une description de ce comportement fondé sur des machines à état. De manière interne un composant est constitué de parties (parts) et peut inclure des connecteurs internes (pour connecter ensemble des sous-composants).

Un *Port* est une entité qui émerge d'un composant ; elle se comporte comme un point d'interaction avec l'environnement du composant. Les interactions peuvent être bidirectionnelles. Un port est typé par une interface. Un port peut être requis (required) pour signifier que l'instance du composant doit être connectée à un port correspondant de l'environnement (c'est-à-dire une autre instance de composant qui fournit le service demandé). Un port qui est non requis offre des services facultatifs qui peuvent donc être laissés non connectés lors de l'exécution.

Un *connecteur* (connector) est une entité qui relie des ports d'instance de composants.

Une *interface* définit un type. Elle contient un ensemble d'opérations et/ou de contraintes. Une interface peut être fournie (provided) par une entité ou bien requise (required).

Une *partie* (part) est un fragment interne d'un composant.

6.2.3 Assemblage

Une application ou un composant est constitué de l'assemblage d'autres composants par l'intermédiaire de ports inter-connectés.

- Un connecteur d'assemblage (AssemblyConnector) permet d'assembler deux instances de composants en connectant un port fourni d'un composant au port requis de l'autre composant.
- Un connecteur de délégation (DelegationConnector) permet de connecter un port externe au port d'un sous-composant interne. L'invocation d'un service sur le port externe sera transmise au port interne auquel il est connecté.

Les composants doivent expliciter leurs besoins ainsi que leurs capacités. La proposition UML 2 définit des moyens d'indiquer qu'une entité dépend d'une autre à l'aide de la méta-entité Dependency. Les services d'un composant peuvent être invoqués seulement par l'intermédiaire des ports de ce composant. Si le port est un port d'interface (c'est à dire un port attaché à un interface) alors soit il y a une instance de classifier réalisant cette interface (qui doit fournir une méthode comme implantation d'opérations), ou bien le port est connecté au port d'une sous-partie interne (ou à une instance de composants internes). De plus, un composant peut décrire les interactions qui doivent se produire sur un port donné en attachant un comportement (par exemple une machine d'état) à ce port. Ceci fourni une sorte de spécification de contrat de niveau 3 sur un port. Les ports peuvent être connectés seulement si leurs spécifications de comportement sont compatibles.

6.2.4 Evénements (Events)

La plupart des modèles de composants inclut un mécanisme pour la signalisation d'événements entre le composant et son environnement. La proposition UML2 n'inclut pas de concepts spécifiques pour la signalisation d'événements aux composants mais s'appuie sur le concept de Signal déjà existant en UML. Il n'y de notion d'enregistrement en attente d'occurrence d'événements. Un cas particulier de modèle d'événement apparaissant dans certains modèles (par exemple Catalysis) inclut un concept de propriété ou une propriété est une partie observable d'un état de composant avec un système de notification de changement incorporé. Cette notion de propriété n'est pas incluse dans

le méta-modèle proposé (mais bien sûr pourrait être définie en utilisant le concept du package Behavior).

6.2.5 Flots (Streams)

La proposition UML ne prévoit pas de flots continus qui spécifieraient des propriétés globales (comme la performance) et/ou les items qui circulent dans le flot ne seraient pas observables (au moins à un haut degré d'abstraction).

6.2.6 Déploiement (Deployment)

A un certain moment de son cycle de vie, un composant sera configuré et installé en tant que sous-ensemble d'une installation d'une application. La notion de composant de UML est indépendante de la plate-forme et des modèles de composants propriétaires. Donc UML fournit des outils conceptuels pour mapper les entités représentant les composants indépendants des plate-formes sur les entités spécifiques aux vendeurs et aux plate-formes.

- Un Artéfact (Artifact) représente une entité spécifique à une plate-forme (par exemple un fichier d'archive contenant du Byte Code Java)
- Un nœud (Node) représente une entité capable d'héberger une exécution d'une instance de composant et de fournir des ressources de calculs et de mémoires.

Chapitre 7

Démarche de construction du logiciel avec UML

7.1 Introduction au cycle de vie du logiciel

Comme on l'a vu au chapitre introductif, la problématique du développement de logiciel au début du XXIème siècle est de concilier l'importance des aspects non fonctionnels (on a affaire en général à des systèmes répartis, parallèles et asynchrones dans lesquelles la qualité de service, la fiabilité, la latence, ou encore les performances sont des facteurs extrêmement importants), avec une demande de flexibilité accrue des aspects fonctionnels : c'est la notion de lignes de produits ou le problème n'est plus d'apporter une réponse optimale à un besoin précis s'exprimant à un moment donné, mais bien d'offrir la possibilité de construire des familles de produits pour répondre à des gammes de besoins évoluant librement dans le temps (révisions successives) et dans l'espace (variantes simultanées d'un même produit, par exemple la gamme de téléphones portables d'un constructeur).

7.1.1 Activités du développement de logiciel

La notion de cycle de vie du logiciel modélise l'enchaînement des différentes activités du processus technique de développement du logiciel. Tous les modèles différencient 5 grandes activités qui vont, selon le modèle, interagir différemment. Ce sont (voir figure 7.1) :

1. l'analyse : comprendre le problème, les besoins
2. la conception : trouver une architecture (décomposer en sous-problèmes) pour résoudre le problème
3. la réalisation : mettre en œuvre, fabriquer

4. l'intégration : faire marcher ensemble les différents morceaux
5. la validation : s'assurer qu'on a bien répondu au problème

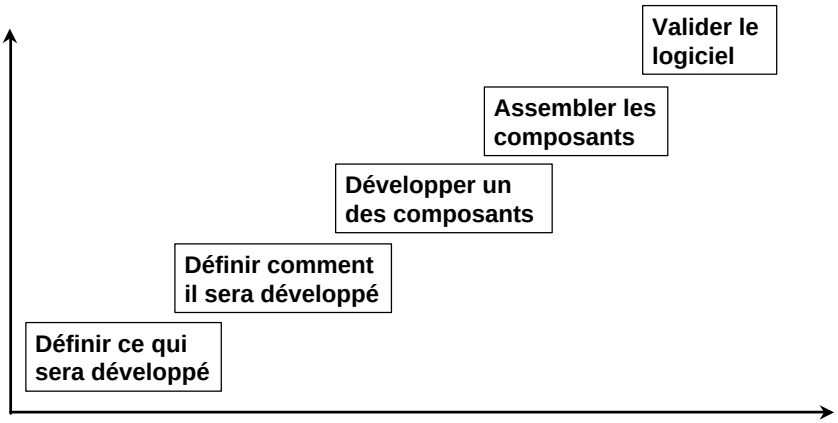


FIG. 7.1 – *Activités du développement de logiciels*

7.1.2 Cycle de vie en V

L'organisation de ces activités et leur enchaînement définit le cycle de développement du logiciel. Par exemple, l'AFNOR a normalisé un cycle de vie particulier, dit en V (voir figure 7.2), qui décrit un enchaînement purement séquentiel des différentes activités, en mettant toutefois en évidence la correspondance entre activités amont (analyse, conception) et aval (intégration, validation). Une variante du cycle en V plus connu aux USA est le cycle en cascade (Waterfall), où au lieu de remonter après la réalisation, on continue à descendre.

Utilisés avec succès depuis le milieu des années 70, ces types de cycles de vie peuvent cependant entraîner des problèmes si on les suit trop rigidement, ce qui est illustré par la célèbre caricature évoquée en figure 7.3.

En effet, cette vision du développement logiciel s'appuie sur une organisation « industrielle » pyramidale héritée du XIXème siècle (en caricaturant à nouveau, quelques cols blancs pour faire l'analyse, en dessous des cols bleus pour la conception, et en bas de l'échelle, les «codeurs»), qui bien que rassurante pour les managers, introduit une hiérarchie malsaine dans les rôles (comment devenir chef à la place du chef?). Les travaux de Jim Coplien (Coplien's organizational pattern : *Architects Also Implement* [23]) ont corroboré cette intuition en montrant que des entreprises qui découplaient les activités du développement de leur organisation hiérarchique se révélaient beaucoup plus efficaces. Une organisation possible est par exemple fondée sur l'idée de sphères concentriques, avec au cœur un petit noyau constitué des personnes les plus expérimentées

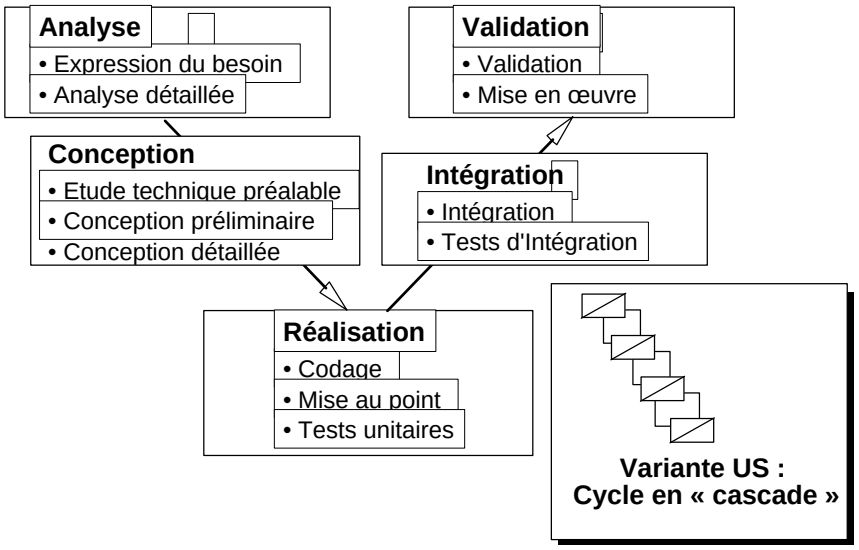


FIG. 7.2 – *Cycle de vie en V normalisé AFNOR*

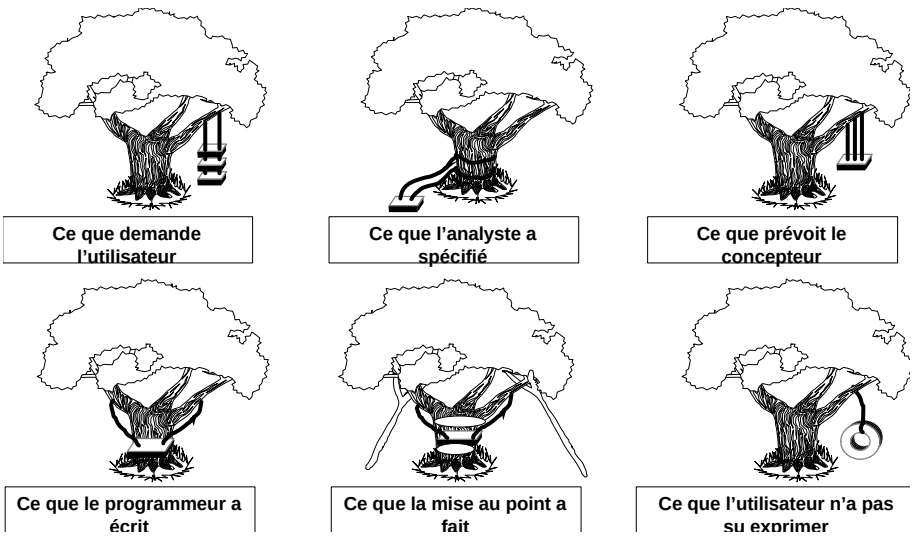


FIG. 7.3 – *Problèmes avec le processus classique...*

(les architectes) capables, après avoir fait les grandes lignes de l'analyse, de concevoir l'architecture du système et d'en réaliser (implanter et tester) la colonne vertébrale, puis autour de ce noyau un second groupe capable de développer les parties les plus critiques du système (analyse, conception, réalisation, test), et ainsi de suite jusqu'à la surface avec des parties de moins en moins critiques confiées à des gens de moins en moins expérimentés.

Un autre problème majeur des cycles de développement de type séquentiel est que, comme il faut attendre d'avoir fini une phase (qui est sensée produire un document gravé dans le marbre) pour commencer la suivante, le temps d'approbation des documents introduit un effet tampon souvent intolérable, et que donc en pratique on commence une phase avant d'avoir fini la précédente, ce qui pose bien-sûr problème si on est parti sur une fausse piste avec un important coût de la (non-) modification d 'un document « final ».

7.1.3 Cycle de vie en spirale

En pratique, cette manière de faire est irréaliste pour un projet innovant, donc à risques, c'est à dire pour lequel on ne connaît pas forcément tous les problèmes avant de tomber dessus, ce qui exige de fréquents retours en arrière et remises en cause. Pour résoudre ce problème B. Boehm [5] a proposé le cycle de vie en « spirale » (voir figure 7.4).

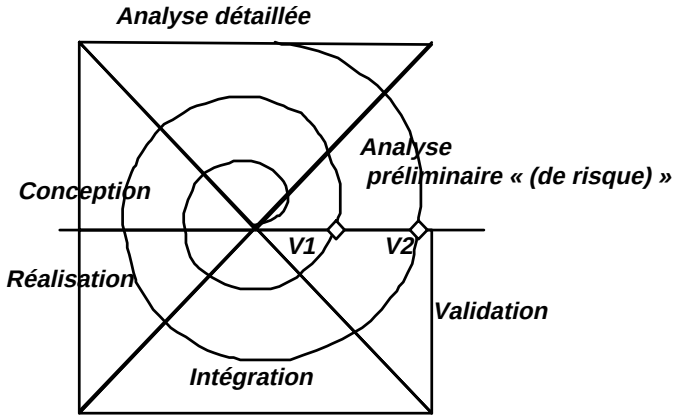


FIG. 7.4 – *Cycle de vie en « spirale »*

Le premier intérêt du cycle de vie en « spirale » et qu'il est particulièrement bien adapté aux développements innovants, c'est-à-dire ceux pour lesquels on ne sait pas forcément à l'avance quelle sera la structure exacte de la solution. D'autre part, avec un cycle de vie en spirale, les progrès sont tangibles : c 'est du logiciel qui « tourne » et

pas seulement des kilos de documents. Ceci permet par exemple d'avoir toujours sous la main une version qu'on peut démontrer, ne serait-ce que pour être présent dans la fenêtre de marché souhaité par le marketing, ou dans le pire des cas de s'arrêter « à temps », i.e. avant que l 'irréalisabilité du projet ait créée un gouffre financier.

En revanche, le cycle de vie en spirale donne des projets moins simple à manager, et pas forcément faciles à gérer en situation contractuelle. De plus s'il est mal contrôlé, on risque de retomber dans le hacking, ce contre quoi avait été inventées les méthodes de développement logiciel dans les années 70.

En termes d'organisation, adopter un cycle de vie en spirale permet de choisir trois modes d'organisation possibles, en choisissant de fixer pour la production des incréments 2 facteurs parmi les trois suivants :

- période (e.g. nouvelle release toutes les 2 semaines)
- fonctionnalités (e.g. releases découpées suivant use-cases)
- niveau de qualité (problème de la mesure)

7.1.4 L'exemple du Rational Unified Process (RUP)

Le Rational Unified Process (RUP) est un exemple de formalisation d'un tel processus fondé sur le cycle de vie en spirale. Il est issu d'une évolution des meilleures pratiques industrielles en la matière, par agglomération des meilleures idées provenant de différentes démarches (voir figure 7.5).

Les point-clés du Rational Unified Process sont les suivants :

- Développer seulement ce qui est nécessaire
- Minimiser la paperasserie
- Flexibilité dans la gestion des besoins, plans, utilisation des ressources, etc...
- Apprendre de ses erreurs précédentes
- Réévaluer les risques régulièrement
- Établir des critères de progrès objectifs et mesurables
- Automatiser autant que possible les activités

Le RUP est architecturé selon 2 structures orthogonales (voir figure 7.6) :

- Structure statique, où l'on trouve les notions de Workers, artifacts, activities, workflows ainsi que authoring et configuration du processus
- Structure dynamique, qui comprend la structure du cycle de vie : (phases, itérations) ainsi que la mise en oeuvre du processus : planification, exécution, gestion des activités, suivi de projet

La structure dynamique est présentée aux figures 7.7 et 7.8.

Bien-sûr, la durée des itérations peut-être très variable en fonction du contexte : l'organisation d'un petit projet de commerce électronique n'a rien a voir avec celle d'un grand projet d'infrastructure nécessitant un Gros travail d'architecture (voir figure 7.9).

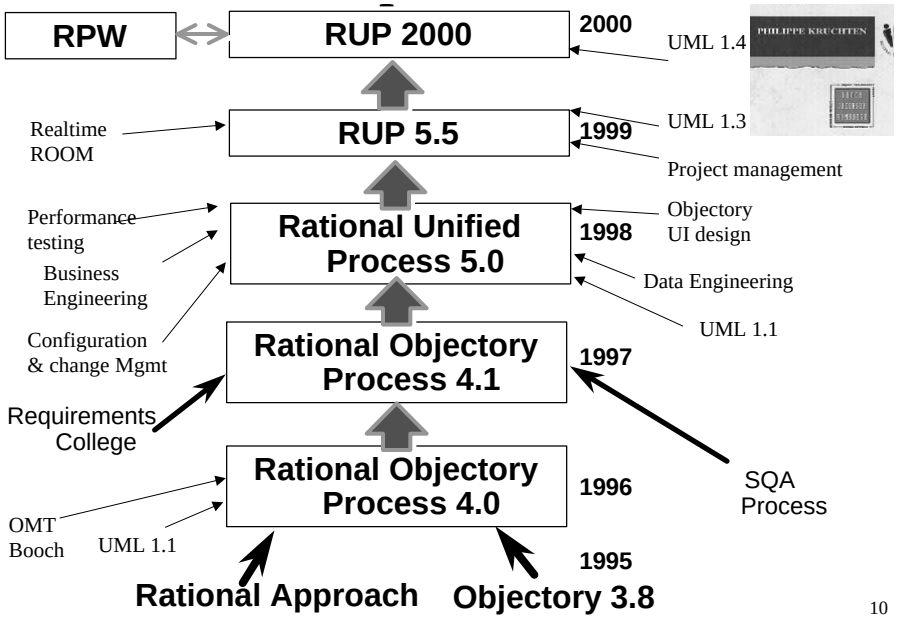


FIG. 7.5 – *Exemple du RUP*

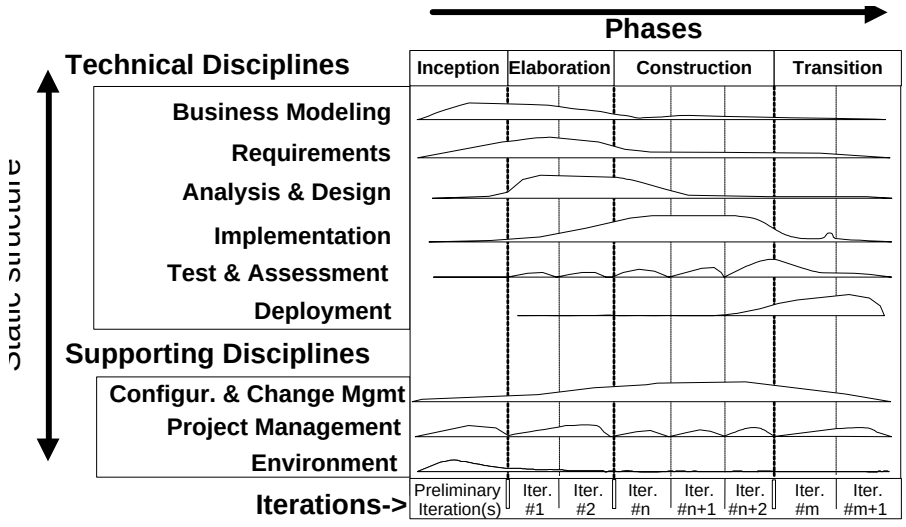
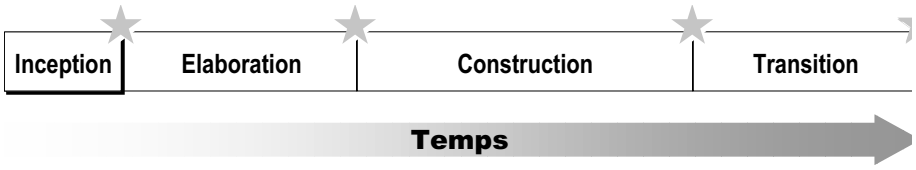


FIG. 7.6 – *Les 2 dimensions du processus*



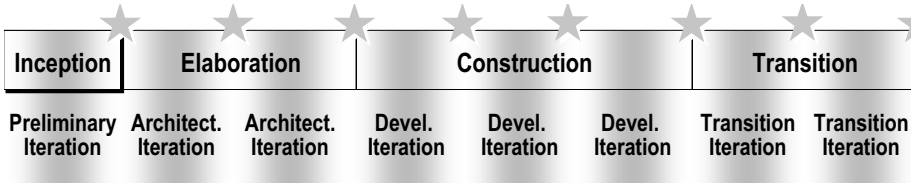
Inception: définition de la porté du projet

Elaboration: planification du projet, spécification des fonctionnalités, architecture de base

Construction: réalisation du produit

Transition: transfert du produit vers les utilisateurs

FIG. 7.7 – *Phases du développement itératif*



Une itération est une séquence d'activités avec un plan bien établi et un critère d'évaluation, résultant en la livraison d'un logiciel *exécutable*.

FIG. 7.8 – *Itérations*

- Petit projet de commerce électronique
  - Intégration à un mainframe
  - 5 personnes
- Grand projet d'infrastructure
  - Gros travail d'architecture nécessaire
  - 20 personnes

	No. of Iterations				Project Length	Iteration Length
	Inception	Elaboration	Construction	Transition		
e-business	0.2	1	3	1	3-4 months	2-3 weeks
infrastructure	1	3	3	2	9-12 months	5-7 weeks

FIG. 7.9 – *Phases et itérations : 2 exemples*

7.1.5 Utilisation d'UML dans les différentes activités du cycle de vie

UML offre une variété de diagrammes qui permet aux ingénieurs de communiquer leurs idées vis-à-vis des différentes activités du cycle de vie du logiciel. Tous diagrammes ne sont pas sont utilisés de la même façon au même moment. La figure 7.10 donne la «température» de chacun de ces diagrammes pour chacune des activités. L'image est ici qu'un diagramme sur lequel on travaille pas ou peu est relativement froid (gelé, couleur bleue ou verte) tandis qu'un diagramme sur lequel on est en train de travailler bouge beaucoup, et est donc représenté avec des couleurs chaudes. Cette figure pourra servir de guide si l'on se pose la question de quels types de diagramme utiliser pour telle ou telle activité du développement de logiciels.

7.2 Construction d'un modèle d'analyse

7.2.1 Principe

La construction d'un modèle d'analyse consiste à élaborer un modèle unique mais quadri-dimentionnel :

- Aspects utilisation du système
- Aspects statiques du système
- Aspects dynamiques du système (comportemental)
- Aspects déploiement et implantation

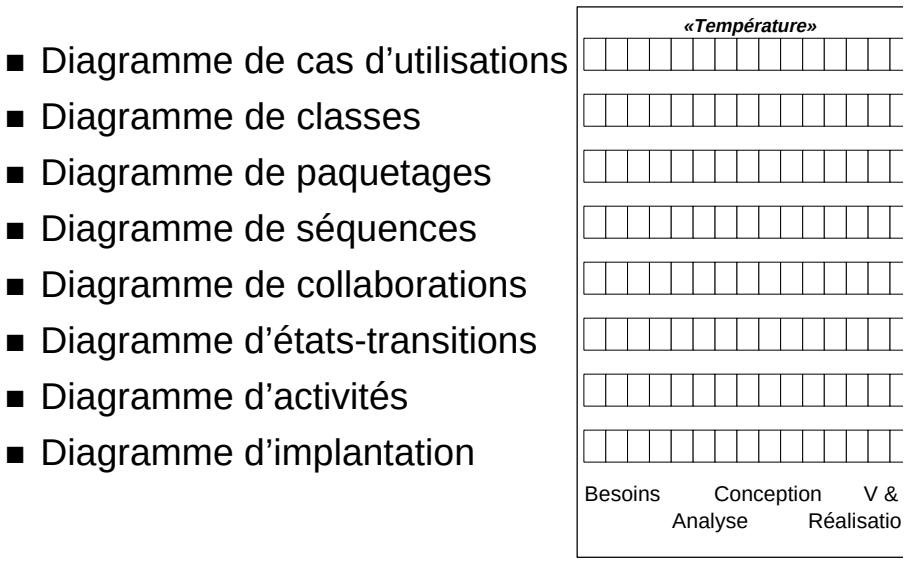


FIG. 7.10 – *Température des diagrammes UML*

La construction d'un modèle d'analyse fait appel à l'hypothèse d'un monde idéal, c'est-à-dire qu'on imagine un système virtuel doté d'un processeur suffisamment rapide, d'une mémoire suffisamment grande et rapide, de communications rapides et fiables, et n'ayant pas de défaillances (l'une des activités de la tache de conception consistera bien sûr à passer de ce monde idéal au monde réel).

Il s'agit donc de faire une sélection des aspects cruciaux d'un problème. Mais bien sûr, il n'y a pas de solution unique : une modélisation est tout d'abord une interprétation du monde réel selon la sensibilité et la compréhension du modélisateur. Ceci n'est toutefois pas quelque chose de spécifiques à l'informatique, comme l'avait noté Gaston Bachelard :

> *Quoiqu'on en dise, dans la vie scientifique, les problèmes ne se posent pas d'eux-mêmes. Rien ne va de soi. Rien n'est donné. Tout est construit.*

Pour donner un ordre de grandeur de l'importance relative des différents aspects d'une démarche typique de modélisation avec UML, on pourra considérer les proportions suivantes :

Construction du diagramme de cas d'utilisations, grande découpe fonctionnelle (10% de l'effort)

Construction du diagramme de classes, à partir des noms des données du problème (30% de l'effort)

Construction de diagrammes de séquences et de collaborations, instances des

cas d 'utilisation (25% de l'effort)

Généralisation à l'aide de diagrammes d'états-transitions, à partir des diagrammes de séquences (15% de l'effort)

Affiner et préciser la solution (20% de l'effort)

7.2.2 Construction des cas d'utilisation

Les cas d'utilisation donnent une grande découpe fonctionnelle du système et servent de fil conducteur, tout au long du projet. Initialement, ils sont utilisés pour capturer les besoins de l'utilisateur, sous la forme d'un diagramme global et d'un petit texte (une dizaine de ligne au plus) décrivant les séquences d'événements (éventuellement précisés par des diagrammes de séquence) dues aux interactions entre l'acteur et le système.

Par la suite, les cas d'utilisation offrent un cadre pour déterminer l'ordre de réalisation des différents prototypes qui caractérisent un développement itératif : ils permettent de piloter les incréments dans la spirale. Chaque prototype réalise tout ou partie des fonctionnalités décrites par un ou plusieurs cas d'utilisation, choisis selon des critères qui visent à réduire les risques. Les premiers prototypes ont pour objectif de valider et stabiliser les choix d'architecture, les prototypes suivants étendent la couverture fonctionnelle, jusqu'à obtention du système complet.

La modélisation par les cas d'utilisation suit un critère de décomposition fonctionnelle. Si ce critère de décomposition est conservé lors du passage vers la définition de l'architecture, la forme du système sera le reflet de la fonction et, de fait, le système sera difficilement extensible.

Afin d'échapper à ce travers, il convient de négocier la transition vers les objets lors du passage à l'analyse du domaine, puis de montrer comment des sociétés d'objets collaborant (initialement issus du domaine, et complétés plus tard par des objets techniques de conception) viennent réaliser les interactions décrites dans les diagrammes de séquence qui documentent les cas d'utilisation.

7.2.3 Processus de construction du diagramme de classes

La décomposition par objets consiste à effectuer un zoom à l'intérieur de la boite qui symbolisait l'ensemble du système dans les descriptions des cas d'utilisation.

Une modélisation par objets est constituée de descriptions dynamiques et statiques. En modélisation objet, les deux aspects ne sont pas disjoints, et l'élaboration des modèles repose sur la consolidation mutuelle des deux points de vue. La dynamique est supportée par la statique, qui n'existe que pour permettre la dynamique, et la structure statique émerge des besoins d'interactions entre objets. Les aspects dynamiques étant modélisés en UML à l'aide de diagrammes spécifiques (diagrammes de séquence, diagrammes de collaboration, diagrammes d'états et transitions et marginalement par les diagrammes

d'activités), il ne faut pas commettre l'erreur de vouloir modéliser dans un diagramme de classe la dynamique du système.

La représentation des classes vise l'abstraction et se concentre sur l'essentiel, sur l'expression du potentiel, des généralités et de la structure statique. Il est généralement impossible de construire un tel diagramme des classes correct du premier coup, à froid, sans l'aide des autres points de vue. Ce processus d'élaboration ressemble plus à un processus de cristallisation, ou à partir de petits éléments bien compris on va progressivement figer une surface de plus en plus grande.

En pratique, n'importe quelle démarche de construction du diagramme de classe est possible a priori. Seul le résultat compte. Mais dans le cas où l'on ne voit vraiment pas comment démarrer, on peut toujours revenir à la bonne vieille technique de l'identification des noms de domaine du problème. Il s'agit dans un premier temps d'identifier des classes potentielles en listant tous les noms et phrases nominales du cahier des charges (ou si celui-ci n'est pas rédigé, en utilisant un processus de remue-méninges aussi appelé brainstorming) ; puis dans un second temps d'éliminer les candidats qui ne semblent pas appropriés. On procédera ensuite de même pour identifier les relations entre classes puis les attributs.

Une fois une liste de classes initiales obtenues, on cherchera à éliminer les classes qui sont :

- redondantes (noms synonymes)
- non pertinentes (vs. le modèle)
- trop vagues (non réifiable facilement)
- des attributs, des opérations, ou des rôles de relations
- des constructions liées à l'implantation

De même, une fois l'identification des relations entre classes effectuée (par exemple par recherche des phrases verbales), on cherchera à éliminer les relations :

- entre classes éliminées
- non pertinentes ou liées à l'implantation
- qui sont en fait des actions
- pouvant être dérivées d'autres relations

Comme un ensemble de boites reliées par de simple traits n'apporte pas beaucoup d'information, il faut aussi autant que possible raffiner la sémantique des relations en :

- ajoutant les rôles
- qualifiant les relations (sélecteur)
- spécifiant la multiplicité

On procède de la même manière à l'identification des attributs. Ce sont les propriétés d'objets individuels de type primitif (booléens, entiers, réels, chaînes, sommes d'argent,

etc.) à rechercher à l'aide des adjectifs (couleur, poids...) ou propositions substantives du problème. On cherchera aussi à éliminer les attributs non nécessaires ou incorrects :

- s'ils sont en fait des objets
- des sélecteurs de relations
- des identificateurs (clef de BD)
- des attributs de relations
- des valeurs internes ou détails d'implantation

À chaque fois que c'est possible, on utilisera le mécanisme de l'héritage pour :

- factoriser des descriptions : pour procéder à des généralisations à l'aide de superclasses, on recherche des classes avec des attributs, relations ou opérations similaires ;
- mettre en évidence de variantes (repérées par exemple par des phrases "soit ... soit" dans le cahier des charges) avec spécialisation à l'aide de sous-classes (menu fixe, menu pop-up, menu déroulant...).

Comme on l'a dit plus haut, il est pratiquement impossible de construire un diagramme de classe d'un seul coup. Il faut donc itérer la modélisation, à la recherche de classes manquantes ou en trop, scinder les classes disparates en classes plus élémentaires, se demander à quoi sert une classe si elle n'a pas d'attributs ou d'opérations, voir s'il n'y a pas de relations manquantes, en trop (si aucune opération ne traverse une relation) ou mal placées, remplacer certains attributs utilisés pour distinguer des objets dans une collection par des relations qualifiées. Mais souvent il faut attendre d'avoir fait les autres vues pour pouvoir itérer efficacement.

7.2.4 Diagrammes dynamiques

L'interaction entre les acteurs et le système (au sein des cas d'utilisation) est documentée à la fois sous forme textuelle, en utilisant le vocabulaire des utilisateurs, et sous forme graphique, au moyen de diagrammes de séquence.

Pour chaque cas d 'utilisation on essaiera de donner un scénario nominal (diagramme de séquence) qui permet d'illustrer ce qui se passe quand tout se passe bien, et quelques scénarios exceptionnels qui montrent des variations sur le scénario optimal.

Selon P.-A. Muller [17], les utilisateurs ont généralement beaucoup de mal à expliquer de manière précise et claire ce qu'ils attendent du système, en l'absence d'une application tangible à critiquer. L'objectif de la démarche par cas d'utilisation est de forcer les utilisateurs à imaginer comment ils entendent utiliser le système, dans le cadre de la satisfaction d'un de leur besoin. Cette démarche, basée sur l'interaction, est très efficace pour obliger les utilisateurs à articuler leurs désirs, parce qu'elle les force à exprimer la manière dont ils entendent utiliser le système.

Chaque cas d'utilisation apparaît alors comme une classe de scénarios, décrits chacun par un ou plusieurs diagramme de séquence. Lorsque le comportement est riche, c'est-à-dire lorsqu'il existe de nombreuses variantes au sein d'un cas d'utilisation, la représentation exhaustive à partir des scénarios n'est plus adaptée et il est alors préférable d'opter pour une modélisation plus abstraite au moyen d'automates à états, visualisés par les diagrammes d'états et transitions.

Alternativement, la méthode Catalysis [10] propose une description par pre/post-conditions de l'état du système avant/après chaque occurrence d'événement.

7.2.5 Construction des diagrammes d'états

Il s'agit de généraliser pour chaque classe qui a un comportement dynamique modal, l'ensemble des scénarios qui mettent en jeu ses instances. En suivant les transition de l'automate, on doit pouvoir retrouver tous les scénarios.

7.2.6 Conseils pratiques

Tout d'abord il faut réfléchir au problème avant de commencer à tracer un diagramme de classes. On répète qu'on a jamais vu personne réussi à produire un diagramme de classes satisfaisant dès le premier coup.

Il est aussi très important de soigner le nommage, et particulièrement sur le nommage des relations et des rôles. On doit être capable de paraphraser un diagramme UML en navigant les relations.

Il est par dessus tout vital de *faire simple*.

> *Things must be as simple as possible, but no simpler.* A. Einstein.

Il faut éviter toute complication nuisible, et se dégager de l'implémentation : raisonner objets, classes, messages, relations, attributs, opérations, etc. et ne pas s'inquiéter si les possibilités de la notation ne sont pas toutes exploitées. Bien-sûr, un modèle très compliqué aura tendance à valoriser la virilité de son auteur (désolé mesdemoiselles), et pourra impressionner ses collègues ou son peu expérimenté chef [1] qui auront peut-être du mal à tout comprendre sans qu'on leur explique 10 fois, mais se révèlera complètement contre-productif pour exactement la même raison. En fait, l'expérience montre qu'*il est simple de faire de modèles complexes, mais complexe de faire des modèles simples.*

Dans une approche incrémentale, il faut itérer et confronter ses modèles aux autres. Mais il est nécessaire de savoir s'arrêter avant d'atteindre la perfection. On répète qu'il ne s'agit donc pas de produire un «bon» système dans l'absolu, mais bien de produire, en suffisamment peu de temps, un système suffisamment bon et suffisamment bon-marché compte tenu du contexte dans lequel il sera utilisé. Il faut donc asservir la construction

1. Mais en aucun cas son professeur!

de la modélisation au processus de développement, en prenant en compte qualité (niveau de précision), coûts, délais...

Répétons pour finir le point essentiel : FAIRE SIMPLE!

Un bon modèle n'est pas un modèle où l'on ne peut plus rien ajouter, mais un modèle où on ne peut plus rien enlever. (d'après A. de St-Exupéry)

7.2.7 Critères de qualité d'un bon modèle d'analyse avec UML

Voici quelques critères de qualité d'un bon modèle d'analyse avec UML, utiles lors de la phase de relecture du modèle[2].

Cas d'utilisation :

- se limiter à environ 6 cas
- nommage correct des acteurs (noms) et des cas (verbes)
- pour chaque cas, présence d'un court texte d'explication
- complétude vs. Cahier des charges

Diagramme de classes :

- nommage correct des classes (noms)
- définitions présentes dans un dictionnaire
- nommage de toutes les relations, présence des cardinalités
- attributs seulement de types simples
- non duplication attributs/relations (utilisation héritage)

Diagrammes de séquence :

- attachés à un cas d 'utilisation
- pour chaque cas d 'utilisation,
- présence d 'un scénario nominal
- quelques scénarios exceptionnels
- chaque opération (message reçu par un objet) définie dans la classe correspondante du diagramme statique

Diagrammes d 'états :

- chaque automate est attaché à une classe
- chaque attribut/opération utilisée sur l 'automate doit être défini dans la classe englobante
- en suivant les transitions de l'automate, on doit pouvoir retrouver tous les scénarios

2. Ces critères sont en particulier utilisés lors de la correction des copies d'examens

Chapitre 8

Introduction aux Design patterns

8.1 Introduction

La définition de la nature de l'activité de conception de logiciel est restée longtemps assez floue. La généralisation dans l'industrie d'une approche par objets de la conception de logiciel a permis de focaliser la tâche de conception sur la description de schémas (ou motifs) d'interactions entre objets, et d'assigner des responsabilités à des objets individuels de telle sorte que le système résultant de leur composition ait les propriétés voulues de flexibilité, maintenabilité, efficacité, etc. L'aspect récurrent de certains de ces schémas d'interactions entre objets a conduit à vouloir les cataloguer : on les appelle alors *design patterns*. On choisit ici de traduire cette notion par "patron de conception", d'une part pour sa sonorité proche de l'anglais mais aussi pour mettre en évidence le parallèle avec la notion de forme prédéfinie existant dans un patron en couture qui laisse en pratique de nombreux degrés de libertés dans son utilisation. Perçus depuis les dix dernières années comme des techniques essentielles à la conception objet, les patrons de conception ont fait l'objet d'un véritable engouement, tant dans l'industrie que dans le monde académique, et ont donné lieu à de multiples publications, tant de catalogues que d'articles scientifiques les analysant. Ce chapitre a pour objectif de familiariser le lecteur avec cette notion de "patron de conception", ainsi que de montrer son utilisation dans un processus de conception de logiciels à objets

8.1.1 Définition simple

La notion de patron de conception a été popularisée au milieu des années 90 et a depuis profondément marqué la pratique de la conception de logiciels orientés objets. Qu'est-ce qu'un patron de conception? Une définition simple et communément admise en fait "une solution à un problème de conception dans un contexte". Ceci mérite bien

sûr quelques explications complémentaires.

- La notion de contexte se rapporte à un ensemble de situations récurrentes dans lesquelles le patron de conception peut s'appliquer.
- Le problème fait référence à un ensemble de forces, à la fois objectifs et contraintes, qui se produisent dans ce contexte.
- La solution fait référence à une forme ou une règle de conception canonique qui peut être appliquée pour résoudre ces forces.

8.1.2 Historique et Motivations

L'idée d'identifier et de documenter systématiquement les patrons de conception en tant qu'entités autonomes clés pour la conception de logiciels date de la fin des années 80. Elle a été popularisée par des gens comme Beck, Ward, Coplien, Booch, Kerth, Johnson, etc. (connus sous le nom de "Hillside Group"), mais il est clair que la contribution séminale à ce domaine a été la publication en 1995 du livre *Design Patterns: Elements of Reusable Object Oriented Software* par la désormais célèbre "bande des quatre" (en anglais *Gang of Four*, ou *GoF*) : Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides [11]. Ce livre, l'un des plus vendus de toute l'histoire de l'informatique, a constitué le premier catalogue de patrons de conception applicable à une grande variété d'applications. Dans ce premier catalogue, la plupart des patrons de conception se focalisent sur un aspect particulier de la conception de logiciel, à savoir sa flexibilité. Il s'agit donc pour l'essentiel de solutions de conception visant à rendre le logiciel conçu plus évolutif, que ce soit dynamiquement ou tout simplement pour faciliter de futures opérations de maintenances. L'idée clé des patrons de conception du GoF est de permettre de faire varier certains aspects de la structure du système indépendamment des autres aspects, afin de rendre le système robuste à certains types de changements qui peuvent alors être effectués sans re-conception. Depuis ce premier opus, de nombreux autres catalogues sont apparus, par exemple les quatre volumes de PloPD (Pattern Languages of Program Design, voir [8, 22, 20, 18]) ou encore le catalogue en ligne `http://wwwipd.ira.uka.de/ tichy/publications/Catalogue.doc`). Certains de ces catalogues contiennent des patrons de conception qui prolongent la tradition du GoF, parfois en se concentrant sur des patrons de conception spécifique d'un métier particulier (voir l'excellent chapitre sur les patrons de conception pour les télécommunications dans [22]) ; mais d'autres en élargissent le spectre pour aborder une plus grande variété d'aspects extra-fonctionnels de la conception, comme la fiabilité, la prise en compte de la répartition des données ou des calculs, la ponctualité, la sécurité, la mise à l'échelle, les performances etc. D'autres élargissent encore ce spectre en abordant des patrons qui ne sont plus de conception, mais par exemple d'analyse ou de rétro conception [21] ou même encore hors du domaine du logiciel, avec des patrons d'organisation [9]. Aujourd'hui, il est indéniable que les patrons de conception sont largement acceptés comme

des outils particulièrement utiles pour guider et documenter la conception de logiciels à objets. Mais au-delà de cet aspect de base, les patrons de conception jouent plusieurs autres rôles dans le processus de développement.

- Ils fournissent tout d'abord un vocabulaire partagé pour communiquer à propos de conception logicielle.
- Grâce à ce vocabulaire, des constructions complexes peuvent être décrites avec un meilleur niveau d'abstraction, ce qui permet de maîtriser la complexité globale d'un système.
- Finalement, les patrons de conception constituent une base d'expérience pour la construction de logiciels sous forme de briques de base de conception à partir desquels des conceptions plus complexes peuvent être élaborées.

8.1.3 Points clés

Partage de savoir faire de conception

Les patrons de conception représentent des solutions à des problèmes qui se produisent de manière récurrente lorsqu'on développe du logiciel dans un contexte particulier. Ils peuvent être considérés comme des micros architectures réutilisables qui participent de l'architecture globale du système : ils capturent les structures statiques et dynamiques des collaborations entre les éléments clés d'une conception par objets, à savoir classes, objets, méthodes, messages etc. Parce que les patrons de conception identifient et documentent l'essence de solutions pertinentes à des problèmes de conception, ils sont d'une certaine façon assez proches de la notion d'idiomes, ou d'astuces de codage, qui existent pour la plupart des langages de programmation (voir par exemple [7] pour des idiomes C++). La principale différence est que les patrons de conception fonctionnent au niveau de la conception plutôt qu'au niveau de l'implantation, mais il est clair qu'ils capturent de manière similaire un certain savoir-faire, qui par définition distingue les concepteurs expérimentés des débutants. Même si des brillants concepteurs de logiciels n'ont pas attendu la popularisation des patrons de conception pour produire des éléments de conception élégants, leur isolement face à l'ampleur de la tâche de conception des logiciels modernes a souvent rendu fragile leur contribution. Ce qui est singulier et si intéressant dans l'idée des patrons de conception est de surmonter cet isolement par un effort concerté au plan international pour identifier, documenter et classifier les meilleures pratiques de la conception orientée objet. A cet égard, il convient de remarquer que la notion de patrons de conception n'est pas intrinsèquement liée aux technologies à objets. Mais force est de constater que, poussé par l'adoption massive par l'industrie à partir du milieu des années 90, le principal effort de catalogage a été effectué dans ce contexte, ce qui a permis d'obtenir effectivement un très grand retour sur investissement vis-à-vis de la qualité de conception des logiciels orientés objets

modernes. La principale contribution des patrons de conception est donc de capturer explicitement le savoir-faire des experts ainsi que les compromis de conception, et de cette façon permettre le partage d'un savoir-faire architectural entre développeurs. Ainsi la conception dans un projet n'est plus dépendante d'un seul gourou qui aurait tout dans la tête : le savoir-faire de conception d'une organisation peut-être documenté à l'aide de patrons de conception. Ceci permet en particulier la répétition de succès antérieurs sur de nouveaux projets ayant le même type de problèmes de conception.

Documentation de conception

Les patrons de conception permettent aussi de documenter de manière concise l'architecture d'un système logiciel par la définition d'un vocabulaire approprié. Les patrons de conception constituent donc les briques de base d'une architecture, et permettent ainsi sa compréhension à un plus haut degré d'abstraction, ce qui en réduit la complexité apparente. Plutôt que de penser la conception en termes de classes individuelles et de leurs comportements, il est maintenant possible de penser la conception en termes de groupes de classes collaborant selon un certain schéma prédéfini. Par exemple, pour expliquer une idée de conception sans faire référence aux patrons de conception, on devra dire à ses collègues que :

> *Cette conception fait collaborer cet ensemble de classes ayant telles et telles relations entre elles de manière à ce que lorsque un événement se produit dans un objet instance de cette classe-ci, une certaine méthode est appelée, qui devrait déclencher telle et telle autre méthode sur les instances des sous-classes de cette autre classe ; lesquelles méthodes doivent rappeler l'objet source de l'événement pour obtenir la formation voulue.*

A l'inverse, un concepteur de logiciels familiarisé avec les patrons de conception les plus courants pourra simplement dire :

> *Dans cette conception, cet ensemble de classes collaborent selon le patron de conception observateur.*

Si ses collègues sont comme lui familiarisés avec ce patron de conception (qui sera d'ailleurs décrit plus en détail ci-dessous), ils comprendront immédiatement les structures statiques et dynamiques mises en jeu dans la collaboration entre classes de cette partie de l'architecture logicielle. De manière au moins aussi importante, les compromis de conception seront immédiatement explicités entre les principes de simplicité, d'efficacité, et la prise en compte d'autres forces extra fonctionnelles comme la réutilisabilité, la portabilité, l'extensibilité, etc. Dans les phases initiales de la conception il est d'ailleurs souvent suffisant de savoir qu'on va utiliser un certain patron de conception à un certain point de l'architecture. Les détails concernant la manière dont ce patron de conception peut-être finalement implanté pourront être abordés plus tard.

Intérêt pédagogique

Finalement, les patrons de conception présentent un intérêt pédagogique certain. Il est souvent affirmé que le but des patrons de conception est d'apprendre par l'expérience, de préférence celle d'un autre. Codifier les meilleures pratiques de conception aide à disséminer cette expérience et aide à éviter certains des pièges du développement de logiciels. Parce qu'ils s'appuient sur les meilleures pratiques des technologies objets, les patrons de conception servent aussi à montrer comment utiliser élégamment et efficacement ces technologies, ce qui en a clairement permis une bien meilleure diffusion vers la fin des années 90.

8.2 Comprendre la notion de patron de conception

8.2.1 Connexion avec d'autres domaines

L'origine des patrons de conception se situe dans le domaine de l'architecture (celle du génie civil) avec les travaux de Christopher Alexander [1] :

> *Chaque patron décrit un problème qui se produit encore et encore dans notre environnement, et ensuite décrit le coeur d'une solution à ce problème de manière à ce qu'on puisse utiliser cette solution plus d'un million de fois, sans jamais le faire deux fois exactement de la même manière.*

On peut remarquer la connexion entre la notion de patron de conception et celle de symétrie en mathématiques, qui est une opération sur un objet mathématique qui laisse cette objet inchangé (par exemple, une rotation ou une translation sont des exemples de symétries). Comme un patron de conception, une symétrie a la propriété essentielle de permettre le changement tout en préservant certains aspects inchangés.

Mais la notion sous-jacente aux patrons de conception est aussi connexe à la notion de savoir-faire caractéristique des métiers d'artisanat traditionnel. Elle englobe en particulier l'idée d'un plus haut niveau de compréhension élaboré à partir d'interactions entre composants primaires. On a ainsi pu tracer un parallèle intéressant entre l'apprentissage de la conception de logiciels et l'apprentissage du jeu d'échecs et, où trois niveaux de maîtrise sont bien identifiés.

Au premier niveau il s'agit d'apprendre les règles de base : pour les échecs le nom des pièces, les mouvements légaux, l'orientation et la géométrie de l'échiquier etc. ; et pour le logiciel la connaissance des algorithmes de base, des structures de données et des langages de programmation. Au second niveau, il s'agit d'apprendre les principes : aux échecs la valeur relative de certaines pièces (comme la reine), la valeur stratégique des cases centrales de l'échiquier, pour le logiciel la connaissance des principes d'organisation du logiciel (modularité, orientation objet, généricité, etc.). Au troisième niveau il s'agit d'apprendre les motifs récurrents. Pour devenir un maître d'échecs, il faut en effet

étudier le jeu des autres maîtres : les parties d'échecs disputées par des grands maîtres contiennent en effet des patrons qui devraient être compris, mémorisés, et appliqués de manière répétitive. De même, l'idée des patrons de conception est que l'expérience de conception logicielle s'acquiert en étudiant des conceptions maîtresses.

8.2.2 Exemple : le patron observateur

Bien qu' il y ait probablement une gamme infinie de patrons spécifiques de domaines particuliers, il est en revanche probable que la plupart des patrons à usage général qu'un ingénieur logiciel devrait connaître ont déjà été identifiés. L'un des plus connu parmi ceux-ci est probablement l'observateur, que nous rappelons ici pour illustrer notre propos. L'intention du patron observateur est de définir une dépendance entre un objet et ses multiples observateurs de manière à ce que lorsque cet objet change d'état, tous les observateurs qui en dépendent sont automatiquement notifiés. Prenons l'exemple d'un serveur de fichiers auquel est connecté un ensemble de PC clients (notion de connexion d'un lecteur réseau dans le monde Windows ou de montage d'un système de fichiers dans le monde Unix, voir figure 8.1).

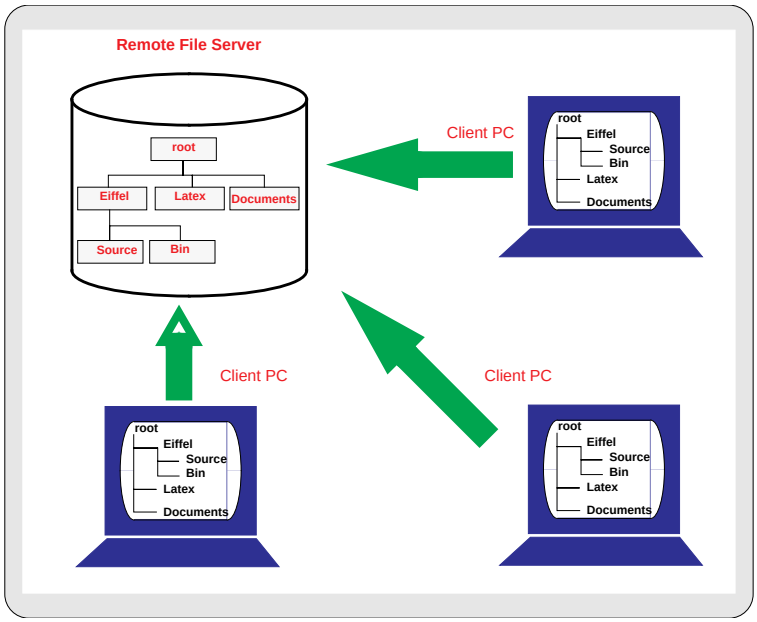


FIG. 8.1 – *Un système de fichier réparti*

À chaque fois qu'un changement se produit sur le serveur (renomage de fichiers ou de répertoires, ajout ou suppression de fichiers), celui-ci doit être reflété sur chaque PC

client de manière à lui permettre de mettre à jour sa vue du système. Une solution à ce problème doit prendre en compte un certain nombre de contraintes qu'on appelle le contexte :

- les identités et le nombre des clients ne sont pas déterminés à l'avance
- de nouvelles sortes de clients peuvent être ajoutées au système dans le futur
- l'interrogation active n'est pas appropriée, car par exemple trop coûteux quand le nombre d'observateurs est grand.

Le patron observateur résout le problème dans ce contexte en faisant en sorte que le serveur (qu'on appelle le sujet) informe les PC clients (les observateurs) à chaque fois qu'un changement intéressant se produit. En utilisant un diagramme de classes UML, la figure 8.2 illustre les relations existant entre un sujet et ses observateurs : il décrit un exemple de structure statique du patron.

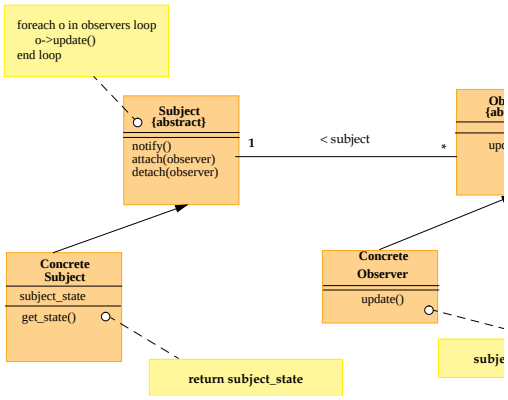


FIG. 8.2 – *Structure statique du patron Observateur*

De même la figure 8.3 illustre l'aspect dynamique d'une collaboration entre un sujet et ses observateurs : c'est un exemple de structure dynamique du patron.

Si vous avez déjà un peu d'expérience en conception de systèmes informatiques, vous devez vous demander pourquoi une solution de conception aussi simple mérite qu'on s'y attarde. Pour répondre à cette question, commençons par prendre un second exemple.

Considérons un outil graphique qui permet à un utilisateur de gérer un ensemble de valeurs de données dans un tableur, un histogramme et un camembert (voir figure 8.4). N'importe laquelle de ces vues peut être ouverte à n'importe quel moment (y compris plusieurs fois la même vue ouverte simultanément) et visible à l'écran. Chaque fois que l'utilisateur modifie l'une des vues, toutes les autres doivent bien sûr être modifiées en adéquation.

Remarquons qu'à un certain niveau d'abstraction, nous avons exactement le même

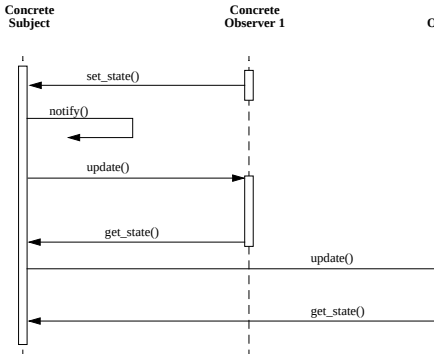


FIG. 8.3 – *Structure dynamique du patron Observateur*

	1er trim.	2e trim.	3e trim.	4e trim.
Est	20,4	27,4	90	20,4
Ouest	30,6	38,6	34,6	31,6
Nord	45,9	46,9	45	43,9

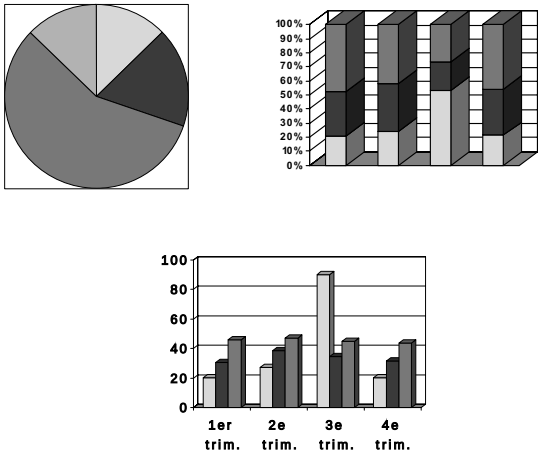


FIG. 8.4 – *Différentes vues sur un ensemble de données*

problème de conception que pour le serveur de fichiers, avec en particulier les mêmes contraintes :

- les identités et le nombre des vues ne sont pas déterminés à l'avance ;
- des nouvelles sortes de vues peuvent être ajoutées au système dans le futur ;
- le polling n'est pas approprié, car par exemple trop coûteux.

Ainsi la solution peut donc être la même : l'ensemble de valeurs de données joue le rôle du sujet alors que les vues jouent celui d'observateurs ; ceux-ci interagissant comme illustré aux figures 8.2 et 8.3.

Le patron Observateur peut alors être formulé de la manière suivante :

- Objectif : Définir une dépendance de un vers plusieurs entre les objets de façon à ce que, quand un objet change d'état, tous ses dépendants sont informés et mis à jour automatiquement
- Contraintes clés
  - Il peut y avoir plusieurs observateurs
  - Chaque observateur peut réagir différemment à la même information
  - Le sujet devrait être aussi découplé que possible des observateurs (ajout et suppression dynamique d'observateurs)
- Structure de la solution
  - Voir figures 8.2 et 8.3.

Ce qui fait de l'observateur un patron de conception est qu'il abstrait les architectures statiques et dynamiques d'une solution donnée pour résoudre un problème particulier dans un contexte lui aussi particulier, et qu'il est donc applicable à beaucoup de situations différentes.

8.2.3 Décrire des patrons de conception

Les patrons de conception sont avant tout des concepts du domaine des idées. Comme toute idée, un patron de conception n'a de valeur que s'il peut être communiqué. Il est donc très important d'avoir un format plus ou moins standard pour pouvoir les décrire. Comme souvent en matière de standards, il en existe plusieurs en pratique. Les plus utilisés sont sans doute le format d'Alexander et celui du GoF. Les patrons

d'Alexander sont décrits sous la forme suivante :

si vous vous trouvez dans un *{contexte}*
par exemple *{exemples}*,
avec *{problème}*,
contraint par *{forces}*
alors pour ces *{raisons}*,
appliquer le *{patron de conception}* et/ou *{règle}*
pour construire une *{solution}*
amenant à un *{nouveau contexte}* et *{d'autres patrons}*

Il y a en pratique de nombreuses variations stylistiques à ce format. Celui du GoF inverse en quelque sorte l'ordre de la présentation en commençant par la forme de la conception puis en décrivant le problème, le contexte et les exemples auxquels elle s'applique. Le canevas de présentation est donc structuré comme suit :

Nom du patron : le nom du patron est choisi de manière à véhiculer succinctement l'essence du patron. Un nom bien choisi est absolument vital, car il devient ensuite un élément clé du vocabulaire de conception. Exemple : observateur.

Intention : un court texte ayant pour objectif de répandre la question suivante : que cherche à faire le patron de conception? Quelle est l'intention sous-jacente ? À quel problème particulier de conception s'attaque-t-on ? Exemple : Définir une dépendance de un vers plusieurs entre les objets de façon à ce que, quand un objet change d'état, tous ses dépendants sont informés et mis à jour automatiquement.

Alias : autres noms souvent donnés à ce patron de conception (s'il y en a).

Motivations : la motivation ou le contexte dans lequel ce patron s'applique. Ceci peut être concrétisé par un scénario qui illustre un problème de conception et comment une structure de classe et d'objets aide à résoudre le problème. Le scénario a pour objectif d'aider à comprendre les descriptions plus abstraites qui suivront.

Applicabilité : les pré-requis qui doivent être satisfaits pour permettre l'utilisation de ce patron. Quelles sont les situations dans lesquelles ce patron de conception peut être appliqué ? Quels exemples de conception peuvent être résolus par ce patron de conception ? Comment peut-on reconnaître ces situations ?

Structure : une description structurelle (en termes de classes et de leurs relations exprimés par exemple en UML) d'un exemple du type de solution proposée par ce patron. Exemple : voir figure 8.2. A cet égard, il faut insister sur le fait que le schéma UML décrivant la structure n'est pas le patron de conception, mais, au coté des autres rubriques, contribue simplement à véhiculer l'idée constituée par ce patron de conception.

Participants : la liste des participants typiquement impliqués dans l'application de ce patron.

Collaborations : comment les participants collaborent pour mener à bien leurs responsabilités.

Conséquences : les conséquences de l'utilisation de ce patron, tant positives que négatives. Comment le patron de conception supporte-t-il son intention ? Quels sont les compromis et les résultats de l'application du patron ? Quels aspects de la structure du système ne permet-il pas de faire varier de manière indépendante ?

Implantation : quels pièges, astuces, ou techniques concernent l'implantation de ce patron de conception ? Y a-t-il des aspects qui sont spécifiques d'un langage de programmation particulier ?

Exemple de code et utilisation : des fragments de code qui illustrent comment on peut implanter ce patron de conception dans des langages de programmation à objets classiques.

Utilisations connues : exemples documentés d'application de ce patron de conception dans des systèmes réels. L'effort de documentation d'une solution de conception n'est en effet valable que si cette solution peut être appliquée de manière récurrente pour résoudre le problème dans un contexte donné. Il est donc d'usage de considérer qu'une solution de conception n'est en réalité un patron de conception que si elle a été identifiée dans au moins trois applications totalement différentes. C'est bien sûr le cas du patron observateur qui se retrouve d'une manière ou d'une autre au coeur de toutes les interfaces graphiques conçues depuis plus de 25 ans.

Patrons connexes : quels patrons de conception ont un rapport avec celui qu'on est en train de décrire ? Quels sont les différences importantes ? Avec quels autres patrons celui-ci peut-il être utilisé en synergie ?

8.2.4 Patrons de conception et notions connexes

On trouve de manière connexe à la notion de patron de conception d'un côté la notion de patron d'architecture, et de l'autre la notion de patron de codage aussi connu sous le nom d'idiome.

Un patron d'architecture exprime un schéma d'organisation structurelle fondamental pour des systèmes logiciels [6]. Il fournit un ensemble de sous-systèmes prédéfinis en spécifiant leurs responsabilités, et inclut des règles et des guides méthodologiques pour organiser les relations entre ces sous-systèmes. On parlera par exemple du patron d'architecture trois tiers pour parler des systèmes constitués par d'une part un serveur d'application effectuant la majorité des traitements, d'autre part d'un serveur de bases de données et un troisième lieu d'une interface graphique déportée sur une station cliente (voir figure 8.5).

À l'autre extrémité du spectre, on trouve la notion d'idiome. Un idiome peut-être vu comme un patron de bas niveau spécifique à un langage de programmation donné. Un

FIG. 8.5 – *Patron d'architecture 3-tiers*

idiome décrit comment implanter des aspects particuliers d'un composant en utilisant les caractéristiques d'un langage de programmation donné [7]. Par exemple il peut s'agir de la manière de gérer l'allocation mémoire dans un langage comme C++.

Sur l'échelle de l'abstraction et du niveau de détail, les patrons de conception se situent à un niveau intermédiaire entre ces idiomes (ou patrons de bas niveau) et les patrons d'architecture. Un patron de conception typique se situe au niveau d'un groupe de classes (une demi-douzaine tout au plus) et de leurs relations. Les patrons d'architecture sont des patrons stratégiques de haut niveau qui s'intéressent à des composants large échelle et aux propriétés globales de systèmes logiciels. Les patrons de conception sont des patrons tactiques à échelle moyenne. Ils sont indépendants d'un langage de programmation particulier, même si les patrons les plus utilisés sont fortement biaisés par l'hypothèse que l'on dispose pour l'implantation d'un langage de programmation par objet. En revanche, les idiomes sont des techniques spécifiques d'un langage pour résoudre des problèmes d'implantation de bas niveau.

8.2.5 Patron de conception et canevas d'application

La notion de canevas d'application (*framework* en anglais) est aussi fortement connexe de celle de patron de conception. En effet un canevas d'application orienté objet est constitué d'un ensemble cohérent de classes interagissant de manière prédéfinie qui peuvent être spécialisées ou instanciées pour réaliser une application. C'est une architecture logicielle réutilisable qui fournit une structure et un comportement générique pour une famille d'applications logicielles dans un domaine particulier.

Un canevas d'application doit être complété par des fonctions spécifiques d'une application donnée afin de former une application complète. Il peut donc être vu comme une application à trous, ou plus précisément comme une structure préfabriquée dans laquelle un ensemble de pièces situées à des endroits spécifiques (appelés points d'accrochage ou encore points chauds) ne sont pas implantés ou bien ont des implantations par défaut qui doivent être redéfinis. Pour obtenir une application complète à partir d'un canevas d'application orientée objet, on doit fournir ces pièces manquantes sous la forme de sous-classes définissant ou redéfinissant les parties manquantes ou existant par défaut, le mécanisme de liaison dynamique (caractéristiques des langages à objet) étant utilisé pour faire en sorte que les éléments prédéfinis du canevas puissent appeler

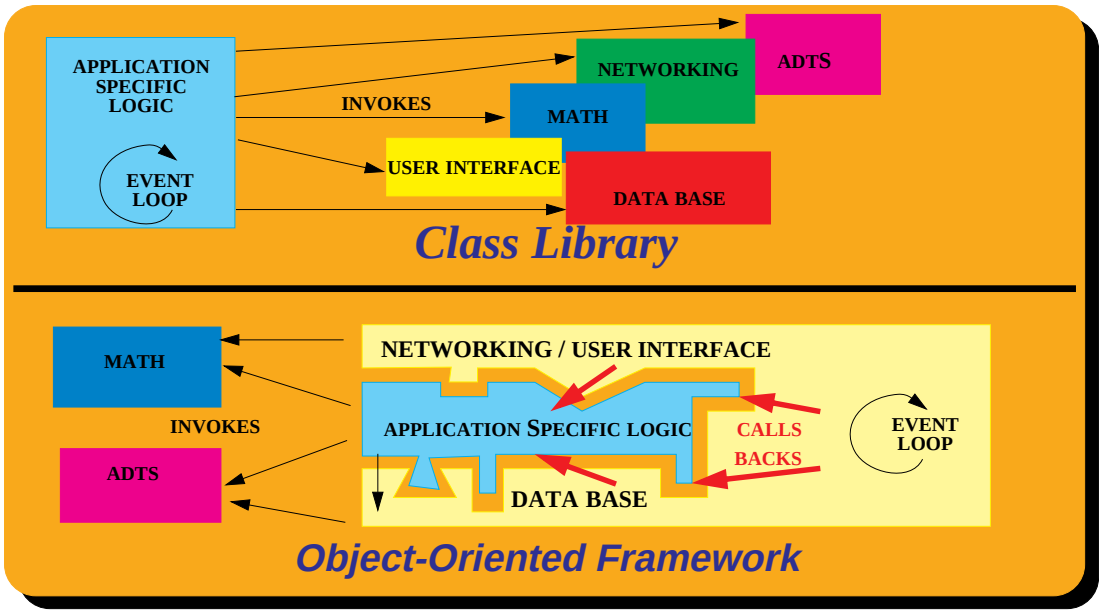


FIG. 8.6 – *Difference entre bibliothèque et canevas d'application (*frameworks*)*

les éléments complémentaires.

Un canevas d'application est ainsi assez différent d'une bibliothèque de classes classiques dans le sens où le flot de contrôle est usuellement bidirectionnel entre les classes spécifiques de l'application et le canevas d'application (voir figure 8.6). Le canevas d'application a fréquemment la charge de gérer la plus grosse part de l'application, le programmeur n'ayant plus idéalement qu'à fournir quelques éléments ici ou là. Mais il arrive souvent que la logique globale de l'application soit assez complexe et que la manière dont doivent collaborer les éléments préexistants et ceux introduits par le programmeur soit non triviale. Les patrons de conception peuvent alors être utilisés pour documenter les collaborations entre les différentes classes du canevas. Symétriquement, un canevas d'application peut mettre en oeuvre plusieurs patrons de conception, certains d'entre eux d'usage général d'autres plus spécifiques. Les patrons de conception et les canevas application sont donc ainsi fortement couplés, mais il n'opèrent pas au même niveau d'abstraction : un canevas d'application est constitué de logiciel concret, alors que les patrons de conception sont de la connaissance, de l'information et de l'expérience à propos de logiciels. On peut dire que les canevas d'application sont de nature physique alors que les patrons de conception sont d'une nature purement logique (ils sont du domaine des idées) : les canevas d'application sont des réalisations physiques d'une ou plusieurs solutions proposées par des patrons de conception ; les patrons de conception

forment en quelque sorte le mode d'emploi de comment réaliser ces solutions.

8.2.6 Un bref aperçu des patrons de conception du GoF

Le catalogue de patrons de conception [11] a organisé les patrons les plus souvent rencontrés en 3 catégories :

Creational patterns Les patrons de création ; ou comment découpler la création d'un objet de la connaissance de son type précis ; voir figure 8.7.

Structural patterns Les patrons structurels ; ou comment découpler interfaces et implantations de classes et d'objets ; voir figure 8.8.

Behavioral patterns Les patrons comportementaux ; ou comment organiser des interactions dynamiques entre groupes d'objets; voir figure 8.9.

Abstract Factory Interface for creating families of objects without specifying their concrete classes

Builder Factory for building complex objects incrementally

Factory Method Lets a class defer instantiation to subclasses.

Prototype Factory for cloning new instances from a prototype

Singleton Access to the unique instance of class

FIG. 8.7 – *Les patrons de création*

Adapter Convert the interface of a class into another interface clients expect.

Bridge Decouple an abstraction from its implementations

Composite Recursive aggregations letting clients treat individual objects and compositions of objects uniformly

Decorator Extends an object functionalities dynamically.

Facade Simple interface for a subsystem

Flyweight Efficiently sharing many Fine-Grained Objects

Proxy Provide a surrogate or placeholder for another object to control access to it.

FIG. 8.8 – *Les patrons structurels*

On trouvera dans [14] le détail des ces patrons accompagné d'un exemple d'implantation complet en Eiffel pour chacun d'entre eux.

Chain of Responsibility Uncouple request sender from precise receiver on a chain.
Command Request reified as first-class object
Interpreter Language interpreter for a grammar
Iterator Sequential access to elements of any aggregate
Mediator Manages interactions between objects
Memento Captures and restores object states (snapshot)
Observer Update observers automatically when a subject changes
State State reified as first-class object
Strategy Flexibly choose among interchangeable algorithms
Template Method Skeleton algo. with steps supplied in subclass
Visitor Add operations to a set of classes without modifying them each time.

FIG. 8.9 – *Les patrons comportementaux*

8.3 Les patrons de conception dans le cycle de vie du logiciel

8.3.1 Représentation de l'occurrence de patrons de conception en UML

Pour faire face à la complexité toujours croissante des systèmes logiciels, on a de plus en plus souvent recours en informatique comme dans les autres sciences à des techniques de modélisation. La modélisation est en effet l'utilisation d'une représentation en lieu et place d'une chose du monde réel dans un but cognitif. Un modèle remplace un système réel dans un contexte donné, à moindre coût, plus simplement, plus rapidement et sans les risques ou dangers inhérents à une manipulation du monde réel. Une fois admise l'idée que la modélisation joue un rôle crucial dans le développement de logiciels, il faut encore disposer d'un langage suffisamment abstrait pour être indépendant de telle ou telle plate-forme matérielle ou logicielle, tout en étant suffisamment précis pour permettre de capturer aussi précisément que nécessaire les aspects fondamentaux d'un problème. Depuis les années soixante-dix, où est apparue l'idée qu'il était nécessaire de modéliser avant de programmer, de très nombreux langages et méthodes ont été proposés pour répondre à ce problème. Le standard actuel du domaine est sans aucun doute le langage de modélisation unifiée UML (Unified Modeling Language), qui ne fait finalement qu'une synthèse (ou du moins tente de le faire) des meilleures pratiques existantes pour le développement par objets de logiciels. UML propose des éléments de modélisation et de visualisation pour représenter les différents aspects des systèmes logiciels. Les éléments de modélisation sont généralement manipulés au travers de représenta-

tions graphiques (les diagrammes UML), qui contiennent les éléments de visualisation qui correspondent aux éléments de modélisation. UML, dans sa version actuelle 2.0, définit treize types de diagrammes, et permet en outre mélanger différents types de diagrammes, par exemple pour combiner des aspects structurels et comportementaux au sein d'une même représentation. UML prend en compte en particulier la notion de patron de conception au travers de celle de collaboration paramétrée. Une collaboration est un contexte pour décrire des interactions entre objets. Une collaboration peut donc être utilisée pour spécifier la réalisation de principes de conception. Il est aussi possible de voir une collaboration comme une entité de conception autonome et donc réutilisable. Ceci peut par exemple être utilisé pour identifier l'occurrence de patron de conception dans un modèle de conception. L'application d'un patron de conception peut alors être vue comme une collaboration paramétrée où à chaque utilisation du patron des éléments de modèles effectifs (comme classes, relations, méthodes, attributs, états, etc.) sont substitués aux paramètres de la définition du patron de conception.

L'utilisation d'un patron de conception est représentée comme une ellipse en pointillé contenant le nom du patron de conception. Une ligne pointillée est tirée entre cette ellipse et chacune des classes (ou chacun des objets s'il s'agit d'un diagramme objet) participant à cette collaboration. Chacune de ces lignes est étiquetée avec le nom du rôle joué par le participant dans la collaboration. Le rôle correspond au nom des éléments décrits dans le contexte de la collaboration (par exemple pour le patron observateur, on trouve les rôles sujet et observateur qui doivent être joués par des classes, mais aussi les rôles notify et update qui doit être joués par des méthodes définies dans les classes jouant les rôles sujet et d'observateurs). Ces noms de rôle peuvent être vus comme des paramètres formels, alors que les éléments du modèle de l'utilisateur sont des paramètres effectifs, les liaisons entre paramètres formels et paramètres effectifs se faisant graphiquement comme illustré dans la figure 8.10. Cette notation à l'avantage de rendre explicite l'utilisation d'un patron de conception donné dans un modèle de conception sans avoir à entrer dans les détails de son implantation.

8.3.2 Des problèmes aux solutions avec les patrons de conception

L'activité de conception par objets de logiciels a pour objectif fondamental de prendre en compte les forces non fonctionnelles (telles que fiabilité, sécurité, performance, ponctualité, consommation d'énergie, qualifié de service, flexibilité, évolutivité, maintenabilité, etc.) implicites dans un modèle d'analyse pour aller vers un modèle qui sera directement implantable dans un langage à objets donné. L'objectif d'une analyse par objet est de modéliser le domaine du problème de façon à ce qu'il puisse être parfaitement compris et servir de base stable à la conception. Le processus de conception peut être vu comme une transformation progressive du modèle d'analyse en un modèle d'implantation par l'application successive d'un ensemble de règles de conceptions.

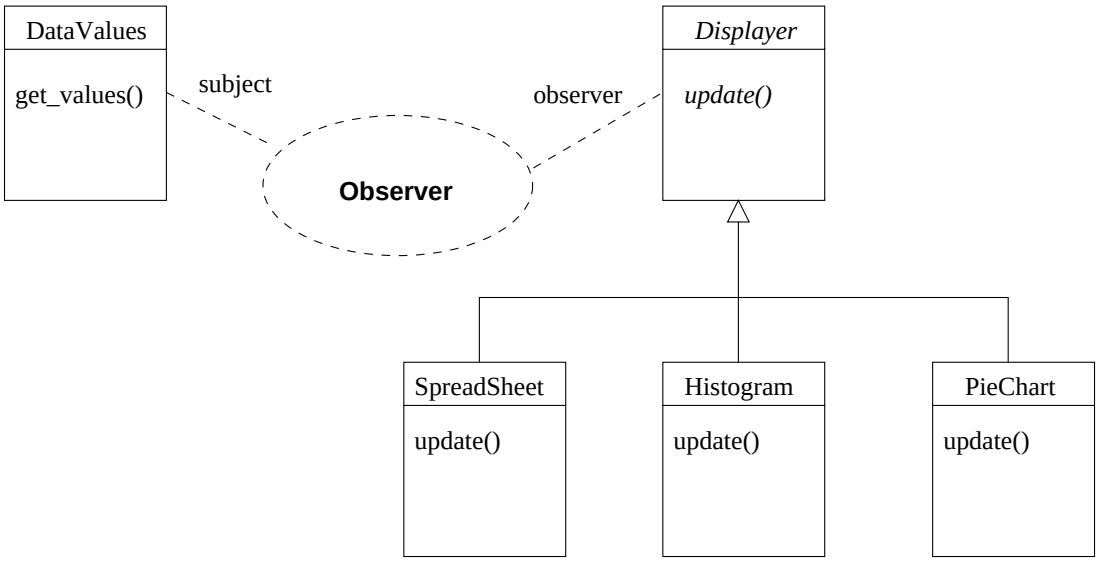


FIG. 8.10 – *Utilisation du patron observateur en UML*

Idéalement, ces règles de conceptions devraient toutes pouvoir être exprimées comme des patrons de conception. On parle alors de langage de patrons (*pattern language* en anglais), dans le sens où chaque patron de conception individuel joue le rôle d'un mot dans un vocabulaire, et où leur articulation cohérente forme des phrases faisant sens et permettant de résoudre des problèmes globaux de conception. Vlissides a par exemple montré comment l'articulation de 5 des patrons de conception du GoF permettait de concevoir un système de fichier :

Composite : pour fournir la structure arborescente du système de fichiers ;

Proxy : pour supporter la notion de lien symbolique (ou de raccourci)

Template Method : pour fournir une protection d'accès aux données ;

Singleton : pour forunir deux niveaux de protection ;

Mediator : pour supporter la notion de groupe d'utilisateur pour l'accès aux fichiers.

Dans ce contexte de transformation progressive du modèle d'analyse en un modèle d'implantation par l'application successive d'un ensemble de règles de conceptions, UML peut être utilisé comme l'épine dorsale supportant ce processus en permettant au concepteur de progressivement entrer dans les détails de son implantation sans changer de langage de modélisation.

8.4 Conclusion

La généralisation dans l'industrie d'une approche par objets de la conception de logiciel a permis de focaliser cette tâche de conception sur la description des schémas (ou motifs) d'interactions entre objets, et d'assigner des responsabilités à des objets individuels de telle sorte que le système résultant de leur composition ait les propriétés extra-fonctionnelles voulues. L'aspect récurrent de certains de ces schémas d'interactions entre objets a conduit à vouloir les cataloguer sous la forme de *design patterns* ou patrons de conception Au cours de dernière décennie, on est ainsi passé progressivement d'une approche artisanale et approximative de la conception à une application rationnelle de solutions ayant fait l'objet d'un catalogage systématique parce que issues des meilleures pratiques du domaine. Aujourd'hui, on aborde un nouveau défi concernant l'automatisation de l'application de ces solutions de conception à l'aide des notions de transformation de modèle ou de tissage d'aspects. D'abord cantonnée au niveau de la programmation, la notion de tissage aspects remonte au niveau des modèles de conception et rejoint celle de transformation de modèles en s'appuyant sur le fait que dans le domaine du logiciel, un modèle a la particularité d'avoir la même nature que la chose qu'il modélise, ce qui ouvre la possibilité de dériver automatiquement, depuis un modèle, une conception logicielle, ainsi d'ailleurs que d'autres artefacts comme le code, les cas de test, des profils de performances, ou de la documentation. Le défi reste bien sûr toujours de concilier automatisation et flexibilité afin de ne point restreindre la créativité des concepteurs.

Chapitre 9

Conception de Logiciel avec les Design Patterns et UML

9.1 Introduction

9.1.1 Place de la conception dans le processus de développement

> *There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.* (C. A. R. Hoare)

Le processus de développement avec UML est fondé sur une approche itérative, incrémentale, dirigée par les cas d'utilisation. Les activités d'expression des besoins et d'analyse ont permis l'élaboration d'un modèle « idéal ».

Le rôle principal de l'activité de conception consiste à passer de ce monde idéal au monde réel, en s'appuyant sur des catalogues de design patterns pour ne pas à avoir à ré-inventer la roue à chaque fois (voir figure 9.1).

Le rôle du Designer est alors crucial pour architecturer et construire un système viable tenant compte de toutes les contraintes exprimées dans l'analyse. Selon Jasper Morrison (un célèbre designer industriel),

> *Le designer est celui qui doit simplifier, donner une personnalité forte et invisible à ce qu'il crée, élaguer, épurer, désencombrer, créer les produits adaptés.*

Cela s'applique avec autant de force en informatique.

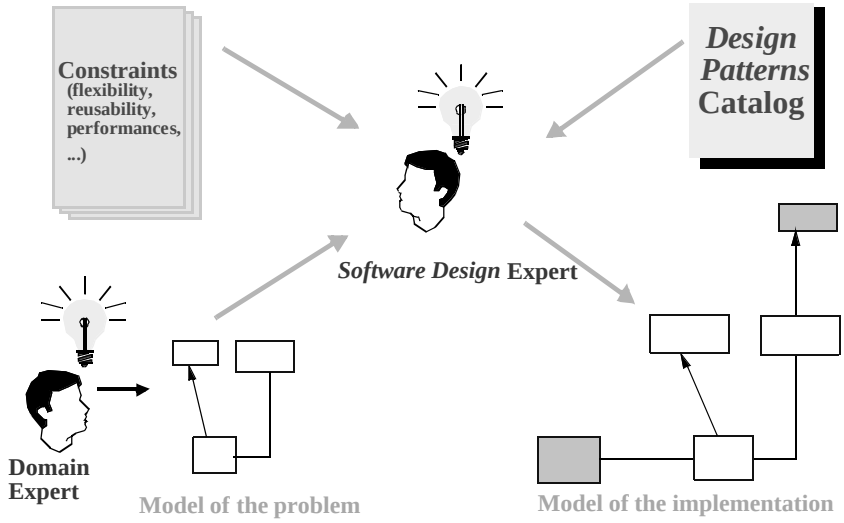


FIG. 9.1 – *La conception dans le processus de développement avec UML*

9.1.2 Aperçu global d'une démarche de conception

La démarche de conception que nous proposons ici est inspirée de méthodes comme OMT, Catalysis, Rapshody, en mettant l'accent sur l'utilisation de design patterns. Elle se découpe en deux grands activités :

Conception Systémique (Architecture) : Il s'agit de faire les choix *stratégiques* de réalisation du système. On va trouver dans cette activité beaucoup d'ingéniérie, et finalement assez peu d'informatique. Le type de design patterns utilisés ici sera appelé «patrons d'architecture».

Conception Objet (détaillée) : Il s'agit de préparer la projection vers une plateforme logicielle et matérielle (un ou plusieurs langages de programmation à objets et leur environnement), et donc de projeter les différents concepts présents dans un modèle UML (aspects dynamiques, fonctionnels, relations...) vers l'axe statique (classes, méthodes et attributs), le seul disponible dans les langages de programmation. Le type de design patterns utilisés ici sera appelé «patrons de conception» détaillée, et ceux-ci seront en général issus du GoF book [11].

Il faut bien comprendre que cette découpe est liée plus à une différence *d'intention* qu'à une différence dans la pratique. En effet dans les deux cas, chaque décision va se traduire par une transformation d'une modèle UML initial où le pattern n'est pas appliqué en un nouveau modèle UML montrant le résultat de l'application du pattern (cf. chapitre précédent). De plus même si dans la suite de ce chapitre nous allons détailler le contenu de ces activités de manière séquentielle, il faut garder à l'esprit que comme

on se trouve dans un cycle de vie en spirale, ces décisions de conceptions (et donc les transformations de modèle afférentes) se trouvent entrelacées.

9.2 Conception Systémique (Architecture)

La conception systémique permet de considérer le système logiciel dans son ensemble et de décider en connaissance de cause les grands choix d'architecture, en fonction des contraintes de type : coût du système produit, performances (vitesse, utilisation mémoire, consommation d'énergie, poids), fiabilité, etc. Nous listons dans cette section un certain nombre de questions à se poser pour mener à bien cette activité. En fonction du type de système qu'on développe, toutes n'auront pas la même pertinence et le même intérêt. Il est clair que ces contraintes n'ont rien à voir entre, part exemple, un logiciel de gestion, et le logiciel de bord d'un avion (accent mis sur la fiabilité), ou un jeu grand public (accent mis sur le prix) et le logiciel d'un téléphone portable (accent mis sur utilisation mémoire, consommation d'énergie, poids du système résultant).

Le plus simple est de se servir de cette section comme d'une *check-list*, qu'on déroule point par point pour être sûr de ne rien oublier.

9.2.1 Organisation en sous-systèmes

Dès qu'un système a une taille significative, il est impossible de la traiter en un seul bloc. On le décompose donc en sous-systèmes, qui seront développés indépendament. Cette découpe peut être horizontale ou verticale.

Dans le premier cas, on obtient une architecture en couches (telle qu'utilisée par ex. dans les protocoles de communication) : une couche n utilise les services de la (ou des) couche(s) de niveau inférieur, et fournit des services aux couches supérieures.

Dans le second cas, on obtient une architecture partitionée fonctionnellement (par ex. bibliothèques Unix). Le plus souvent, bien-sûr, on a une combinaison des deux.

On encapsulera chacun des sous-systèmes obtenus dans son propre package UML (voir figure 9.2).

Pour les sous-systèmes complexes, il peut être utile d'utiliser le pattern *Facade* pour équiper le sous-système d'une interface simplifiée évitant à l'utilisateur de devoir comprendre l'interaction entre les classes le composant.

9.2.2 Choix de l'implantation du mode de contrôle du logiciel

Les différents choix possibles pour l'implantation du mode de contrôle du logiciel sont :

- Procédural, traitement par lots (batch), comme utilisé par exemple dans un compilateur,

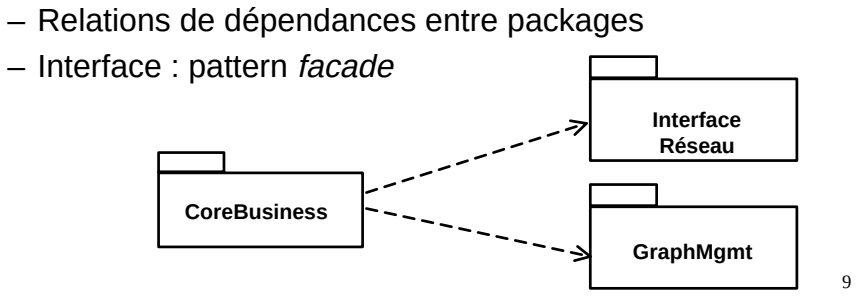


FIG. 9.2 – *Organisation en sous-systèmes*

- Dirigé par les événements, on parle alors de système réactif.
- Concurrent, co-routines
- Autres (programmation logique)

Il faut noter qu'aujourd'hui la plus grande part des systèmes qui sont développés sont des systèmes réactifs. En effet, soit il s'agit de logiciels embarqués (téléphones portable, controle de véhicules, etc.) et ils le sont par nature. Soit il s'agit de logiciels s'exécutant sur stations de travail, qui sont alors le plus souvent pilotés par des interfaces graphiques, qui sont elles-mêmes réactives.

9.2.3 Gestion de la concurrence

Pour pouvoir gérer la concurrence, il faut commencer par clairement en identifier la source. Les deux sources de concurrence sont :

la concurrence intrinsèque, c'est-à-dire celle imposée par l'environnement au système. Par exemple, si le système a plusieurs interfaces, plusieurs événements peuvent se produire simultanément aux frontières du système, et il faut s'organiser pour gérer cette concurrence.

la concurrence comme choix d'ingéniérie, choix qui peut être fait pour deux types de raisons :

- Performance : traitements en parallèle
- Résistance aux défaillances : redondance

9.2.4 Allocation et gestion des ressources

Estimation des ressources nécessaires

- puissance CPU
- mémoire (ROM, principale, secondaire)
- réseaux

– consommation énergie

Compromis logiciels-matériels en fonctions du nombre d'unités fabriquées et du coût unitaire.

9.2.5 Conception du modèle des tâches

Mapping direct vers des Objets Actifs

Une approche classique consiste à utiliser un "real-time operating system' ' et allouer un processus à chaque objet actif.

Objets actifs : activités concurrentes communiquant par envois de messages, souvent munies de statecharts. On se ramene ainsi à un problème de scheduling classique.

Quelques inconvénients de cette approche :

– «context switching overhead»: bcp. de petits processus

– ressources système gaspillées

– processus idle en attente d'événements

– message buffering

– gestion mémoire + disruption du flot d'exécution

Utilisation du pattern Réacteur (Reactor)

Les objets étant par définition réactifs, il semble naturel de construire un système réactif en utilisant directement cette caractéristique. C'est applicable quand le temps de traitement des événements est plus petit que la *deadline* la plus stricte.

L'idée est alors que la plupart des événements peuvent être traités atomiquement, ce qui évite le scheduling pre-emptif (et donc simplifie gestion des ressources, par exemple les verrous). Les messages montent et descendent le long des couches de protocoles (implementées comme des objets) en suivant une chaîne d'appels de méthodes. Et un bon compilateur peut lier statiquement (voire même *inliner*) la plupart des appels de méthodes (par exemple au moins 90% pour GNU Eiffel), ce qui entraine beaucoup moins de défauts de cache instructions.

L'idée du Design Pattern *Reactor* (Schmidt95) est d'implanter au coeur du système un *dispatcher* (aiguilleur) d'événements qui dans une boucle de contrôle sans fin, reçoit les événements et les aiguille vers les objets chargés de les traiter:

```
run is
      -- initialise et démarre l'application réactive
      do
         from initialize
         until is_shutdown_required
         loop
```

```
            process_next_event
      end -- loop
end -- run
```

Réacteur « réaliste »

Pour rendre le réacteur réaliste, il faut gérer les points suivants :

- Interruptions asynchrones (e.g., déclenchement de timers), pour ne pas briser conceptuellement l'atomicité du traitement, on se contentera de lever un drapeau de type «timer sonnant» qui sera consulté à la prochaine itération dans la boucle du réacteur.
- Prise en compte d'événements «longs» à traiter. Ceci ne se produit que si on a des boucles. Dans les cas simples, il suffit alors de rappeler (call back) `process_next_event` depuis la boucle (co-routines). Pour les cas plus complexes, on utilisera un second (voire troisième) thread.
- Un ramasse-miette (ou Garbage Collector (GC)) est très utile dans un environnement de programmation par objets pour des raisons de simplicité et de correction (pour éviter notamment les problèmes de fuites mémoire (memory leaks). Mais il faut utiliser un GC incrémental, activé à volonté, avec un temps d'activation borné dans un intervalle acceptable. Et comme la gestion mémoire se trouve décallée dans le temps vers des moments où le système est moins chargé, celà permet dans certains cas d'améliorer l'efficacité globale de l'application (en débit crête).

9.2.6 Organisation et accès aux données

L'utilisation d'une base de données présente de nombreux avantages pour une vaste gamme d'applications :

- Prévues pour gérer de grandes quantités de données
- crash recovery, partage, distribution, intégrité
- interface standardisée (SQL)
- Modèle d'architecture standard, dit « 3 tiers » (voir figure 9.3).

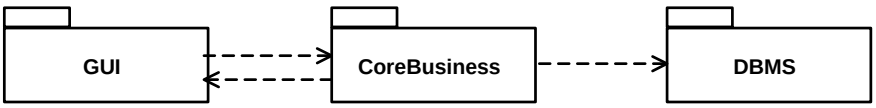


FIG. 9.3 – *Modèle d'architecture « 3 tiers »*

Cependant, pour d'autres applications, l'utilisation de bases de données se révèlera impossible ou inutile. En effet les inconvénients d'une base de données sont :

- surcoût en performances
- difficiles à manipuler si traitements complexes
- interface médiocre avec les langages de programmation

Il faudra alors soi-même décider comment gérer ses données, en particulier vis-à-vis des problèmes de persistance et d'accès concurrents.

9.2.7 Prévisions des conditions aux limites

Jusqu'ici on s'est intéressé au système fonctionnant en régime permanent. Il faut aussi se poser la question des régimes transitoires, à savoir comment initialise-t'on le système, comment est-ce qu'on l'arrête, et que faire en cas d'erreur.

Initialisations

- Lire les informations de configuration (EPROM, disque)
- Créer et initialiser les objets « principaux » (déploiement)
- Etablir les liens, entrer dans la boucle de polling

Terminaisons

sortir de la boucle de polling, nettoyer et sortir

Erreurs et reprises sur erreurs

Les erreurs non fatales sont reportées à l'opérateur. Sinon, si l'erreur est *transiente*, il faut nettoyer et relancer.

Pour savoir si une erreur est transiente, on peut utiliser le *Leaky bucket pattern*, c'est-à-dire le principe du saut percé (ou de la baignoire qui fuit, c'est-à-dire une mesure de débits) dont voici un exemple d'implantation :

```
run is
 -- initialize and run this application
    do
         from initialize
         until is_shutdown_required
         loop
            process_next_event
         end -- loop
         clean_up
```

```
    rescue -- Warm reboot only if transient failure
         clean_up; crash_rate := crash_rate + 1
         if crash_rate <= crash_rate_threshold then
            retry
         end -- if
end -- run
```

9.3 Conception Objet (détaillée)

La conception objet (ou conception détaillée) a pour objectif de préparer l'implantation dans un langage à objets tel que C++, Java, Eiffel ou C#. Comme ces langages ne disposent que de concepts statiques (notion de classes, de méthodes, d'attributs, d'héritage, etc.), il va falloir réaliser une projection des aspects dynamiques (par exemple des diagrammes états transitions) et des autres aspects non directement disponibles dans le langage cible (par exemple les relations) sur ces aspects descriptibles statiquement. On obtiendra ainsi, exprimé en UML, un modèle de l'implantation.

9.3.1 Projection des aspects dynamiques

On commence par s'assurer de la présence d'une méthode par opération possible dans chaque classe. Par exemple, les opérations qu'on trouve sur les diagrammes de séquences seront traduits en termes de méthodes et constructeurs dans la classe cible.

Le cas des diagrammes états transition (statecharts) est en général plus complexe, car il s'agit de trouver un mécanisme permettant de choisir quelle méthode appelée en fonction à la fois de l'état dans lequel se trouve l'objet, et de l'événement à traiter.

$$m = f(state,event)$$

9.3.2 Exemples d'implantation de StateChart

Nous allons maintenant présenter 4 moyens classiques d'implanter un diagramme état-transition, d'abord avec une solution simple, puis avec l'utilisation des design patterns suivants :

Command : réification des événements

State : réification des états

ou les deux simultanément afin de réaliser un double dispatch

Considérons l'exemple d'un diagramme d'états-transitions (ou StateChart) associé à une classe *Book* représentant un livre dans la gestion d'une bibliothèque (voir figure 9.4). Une personne peut emprunter un nombre quelconque de livres, et si un livre n'est pas disponible pour le prêt, celui-ci peut être réservé, puis quand il devient disponible, il est

mis de coté jusqu'à ce que la personne en tête de la liste des réservations pour ce livre vienne le chercher, ou qu'un délai se soit écoulé (auquel cas on prend le second sur la liste et ainsi de suite).

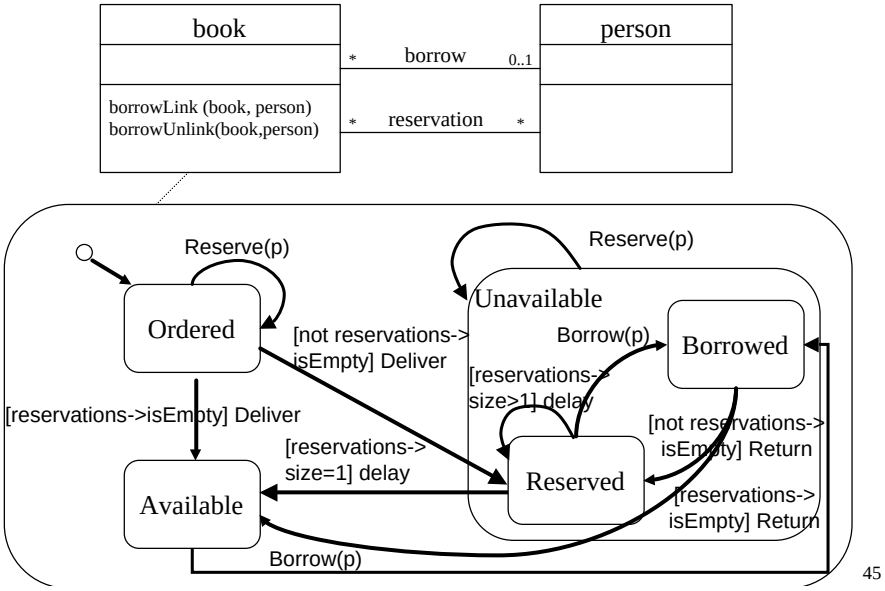


FIG. 9.4 – *Exemple pour l'implantation de StateCharts*

Cas le plus simple

Une première solution simple, utile lorsque le diagramme d'états possède assez peu d'états (deux ou trois) consiste à introduire une méthode `process_e (paramètres optionnels)` pour chaque événement `e` (signal, condition, expiration de délai) du diagramme d'état, puis d'effectuer un traitement par cas selon l'état de l'objet (voir figure 9.5).

Gestion des délais sur les transitions

Dans le cas où on a architecturé son système autour d'un réacteur, on peut facilement gérer une multitude de délais sur les transitions à l'aide du Pattern SoftTimer (voir figure 9.6).

Réification d'événements

Il s'agit de l'utilisation du pattern « command », c'est à dire de définir une classe par événement, avec la possibilité d'utiliser l'héritage pour factoriser des traitements

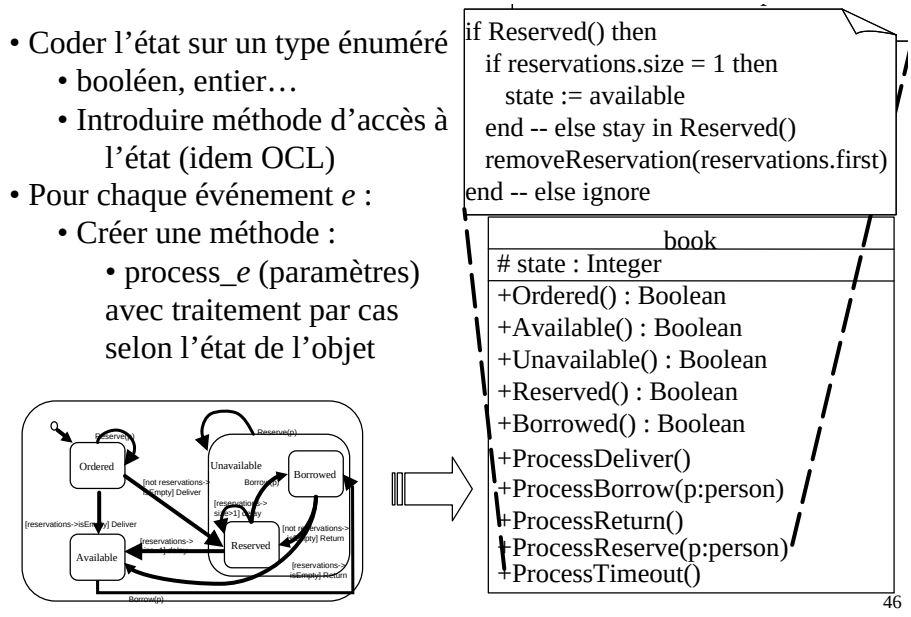
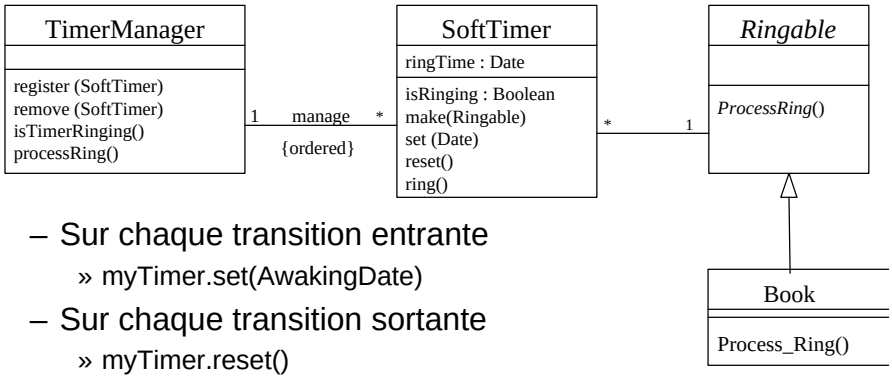


FIG. 9.5 – *Cas le plus simple*



- Sur chaque transition entrante
  - » myTimer.set(AwakingDate)
- Sur chaque transition sortante
  - » myTimer.reset()
- *TimerManager* : regarde périodiquement *t.isRinging* en tête de sa file, et si oui appelle *t.ring()*

FIG. 9.6 – *Gestion des délais sur les transitions*

équivalents, ou la gestion des erreurs(voir figure 9.7). L'utilisation de ce pattern est particulièrement approprié lorsqu'on a à traiter de nombreux événement organisés en une hiérarchie d'héritage complexe. Ceci permet aussi de prendre en compte de nouveaux événements sans avoir à modifier le code existant.

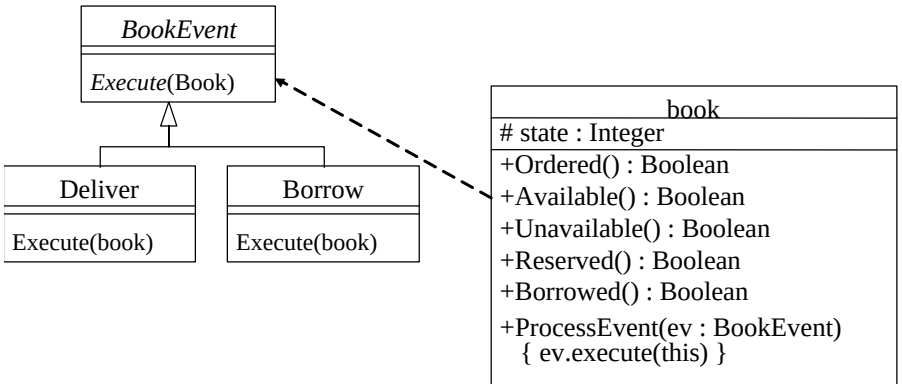


FIG. 9.7 – *Réification d événements*

Réification d'état

Il s'agit de l'utilisation du pattern « state », c'est à dire de définir une sous-classe par sous-état. La hiérarchie d'états correspond à la hiérarchie de classes, avec priorité aux transitions les plus internes comme le veut la sémantique d'UML (voir figure 9.8). C'est le pendant de la réification d'événements. L'utilisation de ce pattern est particulièrement approprié lorsqu'on a à traiter des diagrammes d'états-transitions complexes, en particulier avec de nombreux états imbriqués.

Réification d'état et d'événement

Il s'agit de l'utilisation des patterns State et Command simultanément (voir figure 9.9). C'est bien sûr le choix d'implantation qui offre le plus de souplesse vis-à-vis des modifications ultérieures, mais c'est aussi le plus complexe, qui introduit de nombreuses classes annexes.

9.3.3 Conception des algorithmes

L'activité d'analyse a mis en évidence ce que le logiciel devait faire, mais pas comment il devait le faire. Il s'agit maintenant de concevoir le comment, c'est-à-dire de trouver des algorithmes les plus appropriés pour résoudre les problèmes mentionnés. La

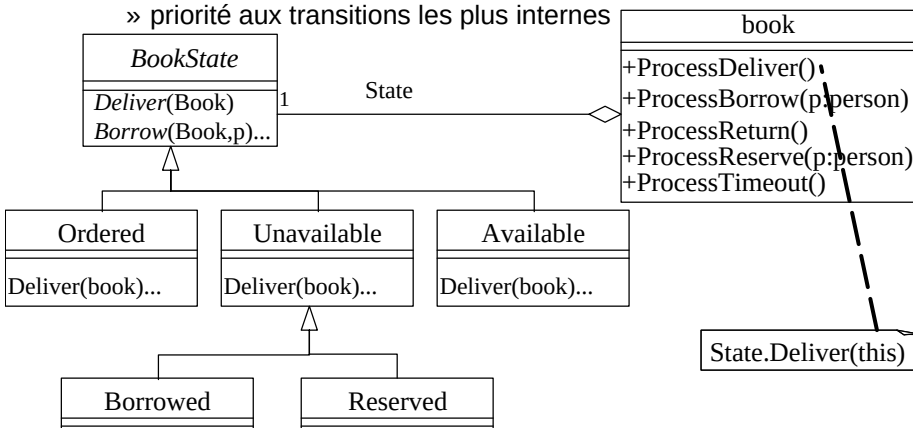


FIG. 9.8 – *Réification d état*

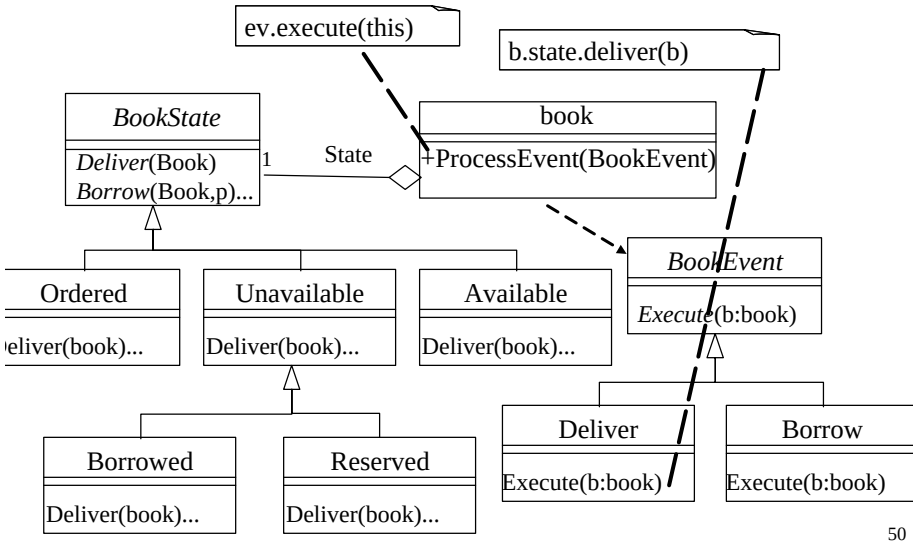


FIG. 9.9 – *Réification d état et d événement*

règle est ici de ne jamais commencer par chercher une solution soi-même, mais de rechercher dans la littérature scientifique des solutions générales, et de les adapter au problème particulier qu'on a à traiter. Dans le cas où plus d'un algorithme existe pour résoudre son problème, on pourra choisir les algorithmes en fonction des critères suivants :

- leur complexité «computationelle» tris $O(n^2)$, $O(n.log(n))$,...
- leur facilité d'implantation et de compréhension
- leur flexibilité

Si le choix doit être fait dynamiquement (c'est-à-dire qu'il existe pas de meilleurs algorithmes dans toutes les situations pour un problème donné), on pourra s'appuyer sur le pattern Strategy dont le principe est d'encapsuler les différentes solutions possibles chacune dans une classe héritant d'un ancêtre commun, et grâce à la liaison dynamique, de sélectionner le plus approprié à un moment donné compte-tenu d'un critère de décision quelconque.

Le fait de concevoir ces algorithmes faits usuellement apparaîtraient le besoin d'utiliser dès structures de données annexes comme des listes, tables, arbres etc. ce qui conduit à un enrichissement du modèle de classes.

9.3.4 Optimisations

Rappelons que le modèle d'analyse avait pour objectif d'être sémantiquement correct, mais a priori n'a aucune raison de proposer des solutions efficaces. Or à cette étape de la conception, on doit avoir une bonne connaissance des points critiques (s'il y en a) vis-à-vis de l'efficacité du système. On est donc bien placé pour faire des modifications nécessaires afin d'optimiser au niveau de la conception ces points critiques. Par exemple, on pourra envisager l'ajout (redondant) de relations pour accélérer l'accès à certaines informations (c'est-à-dire pour éviter d'avoir à traverser plusieurs relations successivement).

On peut aussi envisager la mémorisation d'attributs dérivés pour éviter de les recalculer à chaque accès, ce qui revient à maintenirt à jour un « cache » contenant le résultat du calcul (avec tous les problèmes de maintien de cohérence qui vont avec).

9.3.5 Ajustement de l'héritage

Comme les décisions de conception détaillée ont un impact important sur le diagramme de classes, il ne faut pas hésiter à procéder à des réarrangements des classes et des opérations pour en particulier rechercher la réutilisation. Pour cela on cherchera à rendre semblable ce qui l'est presque (travail sur le type et le nombre d'arguments des opérations, sur leurs noms...). On cherchera aussi à mettre en évidence des abstractions de comportements communs.

Si l'on veut pouvoir faire du partage d'implantation, et que les interfaces des classes existantes sont différentes, on pourra se servir du design pattern adapter :

- Utilisation de l'héritage multiple si le langage le permet proprement (Eiffel, pas C++) : Une pile est une sorte de liste moins certaines opérations (pattern adapter au niveau de la classe).
- Utilisation de la délégation : Une pile possède une liste pour stocker ses données (pattern adapter au niveau de l'objet).

9.3.6 Conception des relations

Finalement, sur le diagramme statique lui-même il existe encore un concept qui n'a pas de correspondance directe dans les langages à objets classiques : il s'agit du concept de relations. Il faut donc trouver un moyen d'implanter ses relations.

Pour cela on procède à une analyse des sens de traversée des relations, en distinguant les cas suivants :

A sens unique, selon la multiplicité :

- vers 1 : attribut (référence)
- vers n : liste ou dictionnaire (hachage)

Bi-directionnels, selon la fréquence :

- un sens peu fréquent : attribut (référence) + recherche dans l'autre sens
- si mise à jour peu fréquentes : attributs dans les 2 sens, mais structure complexe à mettre à jour
- sinon, à l'aide d'un objet matérialisant le graphe de cette relation (la liste des couples des objets qui sont en relations) : c'est la réification de la relation (voir figure 9.10). On pourra bien sûr utiliser à la place d'une liste de couples des structures de données plus sophistiquées, comme par exemple des tables de hachage à double entrée...

9.3.7 Revue de la conception détaillée

De même que pour la construction du modèle analyse, le processus de conception détaillée qui permet d'obtenir le modèle d'implantation n'est pas un processus linéaire en une seule passe. Il nécessite d'être relu et amélioré autant qu'il est possible contenu des contraintes projets (coûts et délais). Une fois de plus, un des points cruciaux sera d'essayer de simplifier au maximum le modèle (Einstein « as simple as possible, but no simpler »). Rappelons ici la citation de St Exupéry, cette fois exactement telle qu'il l'exprima :

> *Un bon design n'est pas un design auquel on ne peut rien ajouter, mais un design auquel on ne peut rien enlever.*

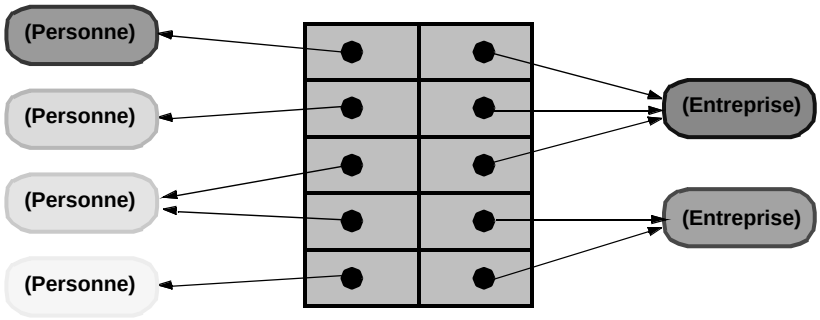


FIG. 9.10 – *Réification d'une relation*

Pour chaque classe, on vérifiera qu'elle en capsule un concept de manière cohérente et modulaire. On devra avoir un concept par classe et une classe par concept. En se débrouillera aussi pour regrouper les classes afin que les paquetages soient le plus faiblement couplés possible (cibles des tests unitaires). Finalement, on s'attachera à documenter toutes les décisions de conception, tâche qui peut être largement soulagée en précisant sur le modèle conception l'occurrence des pattern utilisés, qui sert en quelque sorte de liens vers la documentation correspondante dans les catalogues publiés.

Chapitre 10

Validation et Vérification

10.1 Introduction

Le domaine de la Validation et Vérification (V&V) de logiciels est défini comme suit [4] :

Vérification La vérification est l'ensemble des actions de revue, inspection, test, preuve automatique, ou autres techniques appropriées permettant d'établir et de documenter la conformité des artéfacts du développement vis-à-vis de critères préétablis. La vérification répond à la question «*is the system being built right?*».

Validation La validation consiste à évaluer l'adéquation du système développé vis-à-vis des besoins exprimés par ses futurs utilisateurs. La validation répond à la question «*is the right system being built?*».

Si l'on était capable, dans un cadre général, de prouver qu'un logiciel est conforme à son cahier des charges, le test du logiciel n'existerait pas. Il n'existerait pas et, qui plus est, on aurait la certitude que le logiciel est correct. A l'heure actuelle, il faut renoncer au domaine des certitudes et accepter de n'avoir qu'une certaine confiance (ou méfiance ?) dans les produits logiciels. Le test est nécessaire et s'inscrit dans une démarche empirique où l'on vise à se rassurer sur le caractère *fiable* du produit logiciel plus qu'à garantir sa correction : *Testing can prove the presence of bugs, but never their absence* (Dijkstra). Aucune technique de test, si élégante et si formelle fut-elle, ne peut se targuer de garantir qu'un logiciel n'a plus aucun défaut.

Mais en pratique, le test constitue le moyen principal de validation d'un logiciel. L'introduction au test que nous proposons ici se décompose en deux parties, d'abord un exposé des grandes familles de test du logiciel classique puis les travaux portant sur le test des logiciels orientés objet : pour ceux-ci nous nous concentrerons sur les approches existantes pour le test d'intégration.

10.2 Rappels généraux

10.2.1 Le test de logiciels

Le test de logiciel a pour objectif d'examiner ou d'exécuter un programme dans le but d'y trouver des erreurs. Il est souvent défini comme le moyen par lequel on s'assure qu'une implantation est conforme à ce qui a été spécifié.

Dans le cas où le moyen de détection de l'erreur n'implique pas l'exécution du programme sous test, on parle de test statique. Le test statique s'applique dans toutes les phases du cycle de vie du logiciel, et s'avère particulièrement efficace pour éviter les erreurs de conception. Les techniques de revue ou de relecture de code sont les plus répandues : lorsqu'elles sont systématiquement mises en place dans une entreprise, on estime qu'en moyenne 70% de l'ensemble des erreurs détectées avant la livraison du logiciel sont mises en évidence lors de ces étapes de revue de code.

Dans le cas ou le programme sous test est exécuté, on parle de test dynamique. Comme il est illusoire d'espérer tester un logiciel de manière exhaustive sur tout son domaine possible de valeurs d'entrée (même une simple addition réclamerait 2^64 tests si les opérandes sont codées sur 32 bits), la première difficulté du test dynamique réside dans la sélection (on parle de génération des cas de test) d'un sous-ensemble fini de ces valeurs, que l'on espère représentative de l'ensemble du fonctionnement du logiciel. Le test dynamique prend place à la fin du processus de développement et s'effectue —traditionnellement— en étapes successives correspondants aux étapes du cycle en V :

- le test unitaire : une unité est la plus petite partie testable d'un programme, c'est souvent une procédure ou une classe dans les programmes à objet.
- le test d'intégration : consiste à assembler plusieurs unités et à tester les erreurs liées aux interactions entre ces unités.
- le test système : teste le système dans sa totalité en intégrant tous les sous-groupes d'unités testés lors du test d'intégration.

Dans la suite, on ne considérera que le test dynamique. Pour simplifier, on parlera indifféremment de défaut, de faute ou d'erreur pour désigner ce qui dans le code ou la conception n'est pas correct.

Si le test vise d'abord à détecter des défauts fonctionnels, il inclut aussi la mise à l'épreuve d'aspects non directement fonctionnels, tels que la robustesse (test hors bornes/domaine) ou les performances (test en charge).

Le test dynamique se décompose en grandes étapes logiques : il s'agit de générer des jeux d'essais ou cas de test, de les exécuter, d'analyser la validité des résultats produits, et finalement de localiser et de corriger les fautes. L'étape de localisation et de correction ne relève pas —au sens strict— du test, mais du diagnostic. En effet, si le test vise à la détection des erreurs, le diagnostic vise à leur localisation et à leur correction. Dans la pratique, il s'avère souvent que ces deux tâches soient effectuées par des équipes

différentes (par exemple, très souvent les testeurs " listent " les erreurs détectées et laissent le soin aux développeurs de les corriger).

Le test joue un rôle à la fois dans l'obtention et dans l'évaluation de la qualité d'un logiciel. Lors des premières phases de test (mise au point et tests unitaires) l'objectif du test est d'améliorer la correction de l'implémentation. Pour cela le logiciel est exécuté avec l'intention de révéler des fautes c'est-à-dire de provoquer des défaillances. Chaque fois qu'il se produit une défaillance, les fautes à l'origine de celle-ci sont recherchées et supprimées. L'efficacité de la technique dépend de l'aptitude des ingénieurs de test à mettre au point des jeux d'essais ayant une forte probabilité de révéler, par une défaillance, la présence de fautes.

Lors des dernières phases de test (tests d'intégration et tests du système) l'objectif des tests tend à évaluer la qualité (fiabilité, performances, etc.) de la réalisation.

Quelle que soit la technique de test, tester un logiciel consiste à l'exécuter avec des entrées prédéterminées —les jeux d'essai— et à comparer les résultats obtenus à ceux attendus. Il faut donc dans un premier temps avoir construit ou généré ces jeux d'essai.

Une différence entre les résultats obtenus et ceux attendus pour les entrées sélectionnées révèle une défaillance. Pour déterminer qu'il y a différence, il faut avoir au préalable déterminé un *oracle*, c'est à dire un prédicat spécifiant le domaine des valeurs attendues comme correctes à l'exécution du jeu d'essai.

Une défaillance apporte la preuve que l'implémentation testée est incorrecte. Autrement dit, une défaillance témoigne de la présence dans le texte du logiciel de fautes qui ont été introduites lors du codage ou de la conception. Il s'agit alors de corriger la faute, après l'avoir localisée.

Lorsqu'un logiciel est soumis au test, il n'y a aucun moyen (à moins de pouvoir le tester exhaustivement sur tout son domaine d'entrée) de s'assurer qu'il ne contient absolument aucune faute. Il faut donc décider à un certain moment d'arrêter les tests en espérant avoir obtenu une confiance satisfaisante dans le logiciel sous test. C'est le problème du critère d'arrêt des tests.

En résumé, toute technique de test doit répondre à trois problèmes principaux, et peut être caractérisée en fonction de ceux-ci :

- problème de la génération des jeux d'essai,
- problème de l'oracle,
- problème du critère d'arrêt.

La Figure 10.1 résume le schéma global de l'activité de test.

- Génération de cas de tests : cette étape consiste à trouver des données en entrée pour le programme à tester.
- Exécution : le programme est exécuté avec les données d'un cas de test en entrée

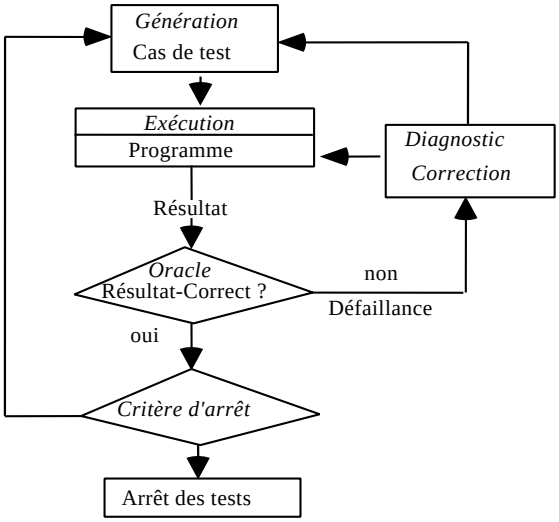


FIG. 10.1 – *Schéma global de l'activité de test*

- Oracle : l'oracle est une fonction qui doit permettre de distinguer une sortie correcte du logiciel.
- Si l'oracle a détecté une défaillance, il faut corriger et ré-exécuter le programme avec le même cas de test pour vérifier que la correction a effectivement éliminé la faute. L'étape de diagnostic —différente du test— vise la localisation et la correction des erreurs.
- Si l'oracle n'a pas détecté de défaillance, soit le critère d'arrêt est vérifié et la phase de test du logiciel est terminée, soit le programme est exécuté avec un nouveau cas de test.

10.2.2 Les techniques de test

Il existe deux grandes catégories complémentaires de techniques de test : le test fonctionnel et le test structurel. Le test fonctionnel est basé uniquement sur la connaissance d'une spécification la plus formelle possible du programme et des services attendus. Cette technique est intéressante car si l'implémentation change, on peut réutiliser les cas de test, et développer les programmes de test en parallèle avec l'implémentation. Cependant, cette technique n'assure pas que toutes les parties du code aient été testées et, à l'inverse, ne permet pas d'éviter la redondance de certains tests (en testant deux fonctionnalités différentes, des parties du code peuvent être testées plusieurs fois).

Le test structurel est basé sur la connaissance de la structure interne du logiciel. Les techniques de test structurel travaillent sur une abstraction du code ou de l'architecture

du logiciel, un modèle structurel du logiciel qui est en général un graphe (graphe de contrôle, de flot de données). L'abstraction permet de définir des critères de couverture de la structure indépendants du programme particulier : couverture de toutes les nœuds d'un graphe, couverture de tous ses arcs (cf. exemple de la Figure 10.2). L'avantage du test structurel est de renforcer la validité des tests puisqu'on sait quelle partie du code a été testée.

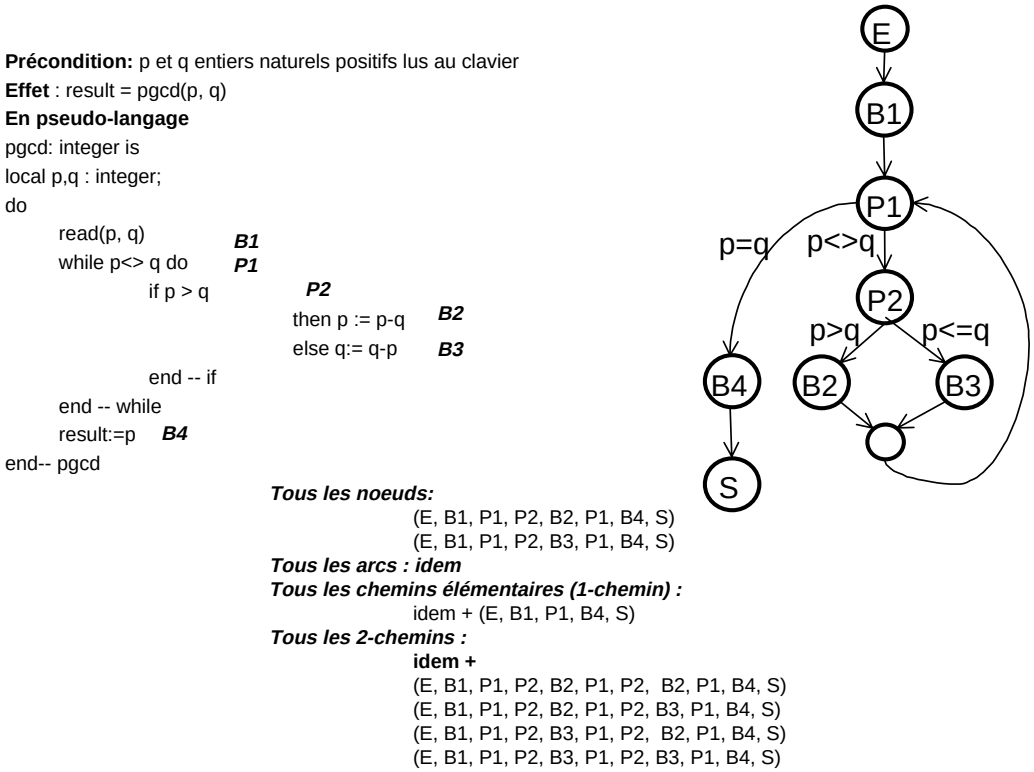


FIG. 10.2 – *Exemple pour le test structurel*

L'exemple de la Figure 10.2 illustre la possibilité de construire un graphe de contrôle à partir du code d'un programme procédural. Un graphe de contrôle a pour but de représenter tous les chemins d'exécution potentiels d'une procédure. C'est un graphe orienté comportant un nœud d'entrée E et un nœud de sortie S (un nœud fictif est ajouté si le programme a plusieurs sorties). Les nœuds correspondent aux blocs élémentaires du programme, ou bien aux prédicats des conditionnelles/boucles, ou enfin à des nœuds de jonction "vides" associés à un nœud prédicat. Un graphe de contrôle correspond à l'or-

ganigramme du corps d'une procédure. Un bloc élémentaire est une séquence maximale d'instructions séquentielles et un arc exprime l'enchaînement d'exécution possible entre deux nœuds.

Différents critères de sélection des tests fondés sur le flot de contrôle, plus ou moins difficiles à obtenir, existent :

- Couverture des instructions : chaque bloc doit être exécuté au moins une fois
- Couverture des nœuds (garantit la couverture des instructions)
- Couverture des arcs ou enchaînements
- tous les chemins élémentaires ou 1-chemins
- tous les i-chemins : de 0 à i passages dans les boucles
- tous les chemins : si boucles, potentiellement infinis

A partir du graphe de la Figure 10.2, on peut écrire des tests en fonction du critère de couverture choisi. Pour couvrir toutes les instructions, il faudra un cas de test (e.g. pgcd(12, 8)), alors que pour couvrir tous les chemins élémentaires, il en faut au minimum trois (e.g. pgcd(1, 1) , pgcd(4, 3) et pgcd(3, 4)).

Dans la pratique, test fonctionnel et test structurel sont complémentaires. Dès que l'on vise la couverture de chemins, le problème des chemins infaisables apparaît : certains chemins du graphe peuvent ne correspondre à aucune séquence d'exécution dans le programme réel. Le testeur doit déterminer ces chemins infaisables et justifier pourquoi ils ne peuvent être couverts. Le test structurel est donc une activité qui serait trop fastidieuse si elle n'était couplée avec le test fonctionnel et supportée par des outils (tel Logiscope pour le plus connu) : les tests fonctionnels sont réappliqués comme test structurels et l'outil de couverture indique les nœuds/arcs/chemins non encore couverts. Le testeur complète alors les tests manquants pour assurer le niveau de couverture souhaité.

D'autres modèles existent, plus complets que le graphe de contrôle, afin de capturer plus d'informations utiles au test. Par exemple, le graphe de flot de données permet de suivre la *ligne de vie* des variables et de concentrer les tests sur des variables d'intérêt. La complexité des prédicats de contrôle peut aussi être prise en compte pour qu'ils soient " couverts " par les tests.

Une technique structurelle particulière pour la génération de tests est l'analyse de mutation. Cette méthode permet, par injection de fautes simples dans le logiciel, de construire des tests capables de détecter ces fautes. La qualité du test est alors reliée à la proportion des programmes erronés que les cas de test sont capables de détecter.

10.2.3 La réutilisation des tests / le test de non-régression

On appelle test de non-régression le fait de s'assurer, lors de l'évolution du logiciel (ajout de fonctions, refactorings), que les modifications ajoutées n'ont pas altéré les

fonctionnalités existantes. La clef du test de non-régression réside dans la mémorisation des cas de tests et des oracles de la version antérieure : il suffit alors de rejouer les tests et de vérifier que les résultats sont inchangés. Dans le cas contraire, il faut disposer d'un moyen de déterminer l'impact des modifications sur les parties existantes du logiciel pour les restester complètement.

10.3 Le test des logiciels à objets

Du point de vue du test, les logiciels à objets diffèrent des logiciels procéduraux classiques par la petite taille des procédures/méthodes (en général inférieures à six instructions). Ceci s'explique par un découpage conceptuel des architectures en vue de la factorisation/réutilisation de code et par l'utilisation intensive du polymorphisme et de la liaison dynamique qui permettent répartir/déléguer les traitements (et donc le contrôle). En conséquence, la difficulté du test porte moins sur le test des méthodes prises individuellement (de faible complexité) que dans le test des interactions entre les objets du système. L'intégration est donc l'étape de test la plus cruciale pour le test de logiciel à objets, car le test du code des méthodes ne couvre pas les aspects traitements distribués propres aux applications conçues par objet.

Il est difficile d'appliquer la décomposition classique test unitaire/test d'intégration/test système pour des logiciels qui ne sont a priori pas conçus à partir d'un cycle de vie en V. Toutefois, on peut considérer que le test d'une classe et de ses méthodes correspond au test unitaire, que le test d'intégration consiste à valider le système vu comme un ensemble de composants communicants, chaque composant ayant été testé isolément au niveau unitaire. Mais comme pour tester unitairement une classe, il faut déjà disposer des classes dont elle est cliente, la différence entre test unitaire et test d'intégration est plus une différence de point de vue qu'une réelle différence de démarche. En fait, on commence à faire du test unitaire qui se transforme progressivement en test d'intégration et se termine en test système.

10.3.1 Le test unitaire/ réutilisation de composants

Le succès de l'approche objet tient beaucoup de la notion de composant réutilisable. La réutilisation de composants logiciels devient un enjeu crucial tant pour éviter de réécrire inutilement du logiciel, que pour réduire les coûts de développement. On cherche alors à développer la notion de composant objet "sur étagère", et il faut pour cela pouvoir mesurer la confiance que l'on peut avoir en un composant. Une solution consiste à considérer chaque composant comme un tout possédant une spécification (ses contrats), une implémentation et des tests qui lui permettent de se tester.

La Figure 10.3 illustre cette vue d'un composant en triangle. Le sommet " spécification " comporte non seulement la documentation mais aussi les contrats, c'est-à-dire

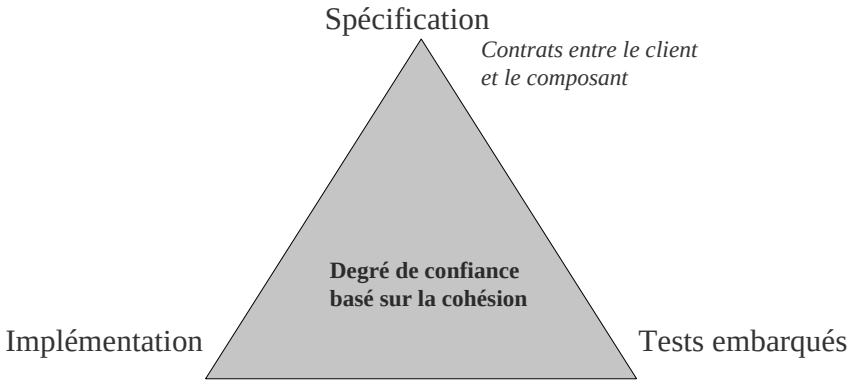


FIG. 10.3 – *Composant autotestable et la vue en triangle*

les pré/post conditions de méthodes et invariants de classe. Ces contrats pouvant être exécutés, ils servent d'assertions qui vérifient dynamiquement la cohérence de l'état du programme à chaque appel de méthode. Ils peuvent donc, dans une certaine mesure, servir " d' oracles embarqués ". Si les contrats sont suffisamment précis, ils ont une probabilité importante de détecter un état erroné du programme. Un composant bien spécifié devrait avoir une bonne capacité à détecter des anomalies internes ou des utilisations non conformes. La plupart des langages à objets supportent les contrats soit de manière native (Eiffel) soit à travers une extension (icontract et JContract pour Java).

Le sommet " test embarqués " du triangle comporte les cas des test (sous forme de méthodes de test exécutables) et les oracles correspondants (sous forme d'assertions explicites pour chaque cas de test). L'important dans le fait d'embarquer les tests est de se donner la possibilité des les exécuter lorsqu'on le souhaite : divers frameworks objets permettent d'embarquer des tests unitaires dans des classes et de les stucturer en cas de test et suite de test. La famille de frameworks *Unit proposée par Gamma et Beck [2] permet de réaliser facilement des classes autotestables (Junit pour Java, Nunit pour C#).

Les tests sont donc considérés comme faisant partie du composant. Ceci permet de fournir une première solution au problème de la réutilisation et de l'évolution du

composant puisqu'il porte la capacité à se tester (autotest) qui peut être appelée depuis le système.

Divers cas de figures peuvent être pris en compte pour un composant C dans un système S :

- modification de l'implémentation de C : l'autotest permet de tester la nouvelle implémentation, la spécification ne change pas,
- ajout de nouvelles fonctionnalités à C : l'autotest doit être lancé pour assurer le test de non-régression puis complété pour tester les nouvelles fonctionnalités. La spécification doit être complétée.
- réutilisation du composant dans un système S : quand on modifie C, tous les composants clients de C doivent s'autotester pour garantir la non-régression.
- intégration : lors de l'ajout d'un composant C dans un système, l'autotest de C doit être lancé depuis le système pour vérifier sa bonne intégration (héritage et clientèle).

L'un des aspects de la conception de composants autotestables et réutilisables est d'assurer la cohésion entre l'implémentation, les tests et les contrats. Avant d'embarquer les tests dans le composant, il faut s'assurer que les tests ont un qualité minimale (soit en terme de couverture du code, soit à l'aide d'une autre technique de qualification des tests, comme l'analyse de mutation).

10.3.2 Le test d'intégration classique et le problème de l'objet

Le test d'intégration consiste à valider le système vu comme un ensemble de composants communicants, chaque composant ayant été testé isolément au niveau unitaire. Il s'agit donc de vérifier le bon fonctionnement d'un système complexe, en général en intégrant les composants par étapes au système.

Le test d'intégration est intermédiaire entre le test unitaire et le test système : il s'agit moins de tester la validité des fonctions principales du système et les propriétés non-fonctionnelles spécifiées dans le cahier des charges que d'assembler de manière fiable les composants. A terme, le test système est possible précisément parce que l'intégration des composants s'est passée de manière satisfaisante et que toutes les anomalies détectées sont corrigées. Les principales anomalies devraient porter théoriquement sur les interfaces des composants, mais la pratique montre que de nombreuses bogues rémanentes sont révélées à cette étape, ainsi que des erreurs portant sur la compréhension des fonctions qui devaient être réalisées et dont l'incohérence apparaît lors des tests d'intégration (ce sont les "fautes sémantiques" portant sur une ambiguïté des dossiers de conception ou une incompréhension de ceux-ci). A l'issue du test d'intégration, on est censé disposer d'un système s'exécutant correctement et dont toutes les fautes portant sur la cohérence de la structure ont été révélées et corrigées.

Le test d'intégration (sauf dans le cas du "Big Bang" qui consiste à tout tester ensemble sans aucune étape intermédiaire et qui est le contraire d'une intégration) suppose la connaissance d'une structure représentant la manière dont sont dépendants les composants du système, ce que l'on nomme l'architecture. Il s'agit donc de test structurel, basé sur l'architecture du système, par exemple un diagramme de classes UML.

Le fait d'intégrer les composants par étapes peut nécessiter la création et l'emploi de *stubs* ou *bouchons de test*.

Un *Stub* est un composant fictif, souvent simplifié, se substituant ou simulant le comportement d'un vrai composant. Le test d'un composant A qui dépend d'un composant C qui n'est pas encore testé implique en effet le remplacement de C par ce composant de substitution, ou *stub*. Un *stub* est spécifique s'il simule le comportement de C uniquement pour son utilisation par A. Un *stub* réaliste correspond à un simulateur de C fonctionnant pour tous ses utilisateurs. Dans l'exemple de la Figure 10.4, on compte soit 2 *stubs* spécifiques de C (l'un vis-à-vis de A, l'autre de B) ou bien un stub réaliste. Les *stubs* réalistes peuvent être d'anciennes implémentations fiables du composant *stubbé*.

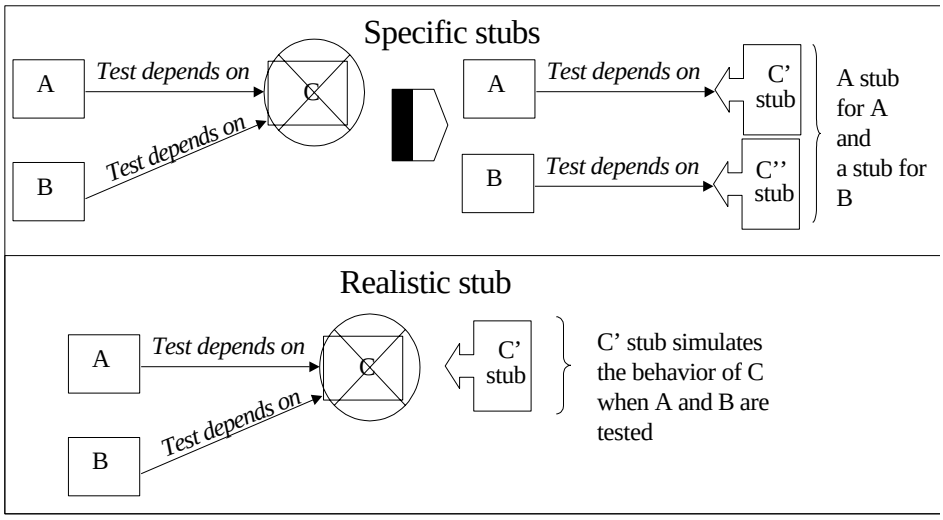


FIG. 10.4 – *Stubs réalistes et spécifiques*

L'effort de test sera lié au nombre de *stubs* nécessaires à la stratégie d'intégration. Le nombre de *stubs* sera calculé en fonction du type de *stubs* que l'on peut développer.

Le développement d'un stub est coûteux, et de plus risque de créer des erreurs à l'intégration du fait du caractère simplifié de tels composants de substitution.

Les problèmes principaux liés à l'intégration des systèmes à objet sont donc les suivants :

1. Minimisation du nombre de *stubs* devant être créés
2. Minimisation du temps d'intégration en fonction des ressources de test (nombre de personnes pouvant travailler à l'intégration du système).

La question de l'intégration est donc une question de planification des étapes de test : il s'agit de définir dans quel ordre les composants doivent être intégrés. La manière d'intégrer s'appelle stratégie d'intégration. Un exemple de stratégie d'intégration est donné dans la Figure 10.5, la stratégie "Bottom-up", qui part des feuilles de l'architecture pour intégrer. Les composants appelés "driver" sont les composants de test qui possèdent les données avec lesquelles le composant sous test doit être exécuté avec la partie du système déjà intégrée.

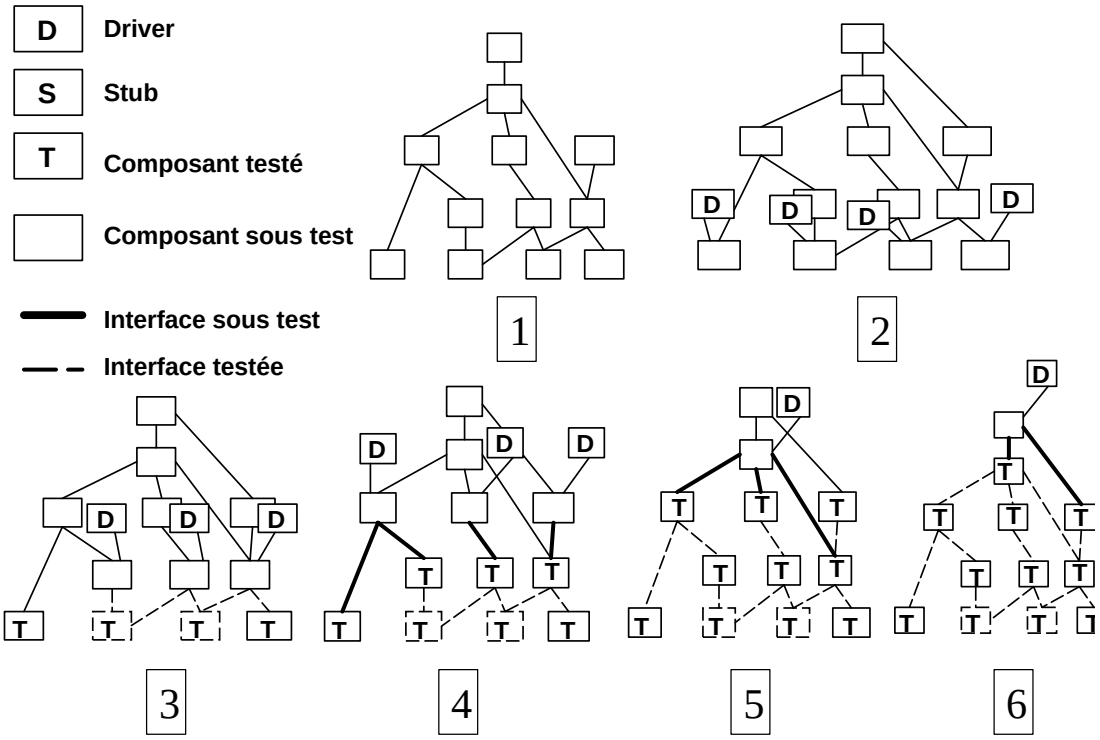


FIG. 10.5 – *La stratégie «Bottom-up»*

Une autre stratégie —qui correspond à intégrer au fur et à mesure que les composants sont implantés dans une méthodologie flot de données (SADT etc.)— consiste à intégrer

depuis le composant racine, en ne construisant qu'un seul driver de test pour tout le système : c'est la stratégie "top-down" dont nous présentons les deux premières étapes dans la Figure 10.6

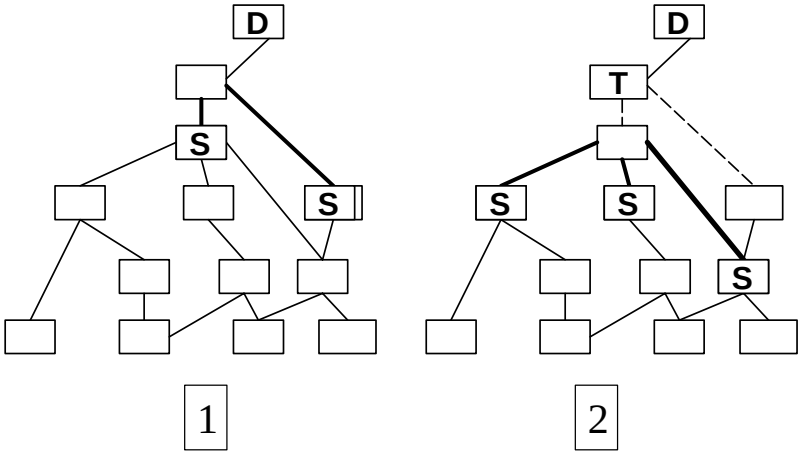


FIG. 10.6 – *La stratégie « Top-down »*

Le défaut de cette approche vient de ce qu'elle nécessite à la fois la construction d'autant de stubs que de composants (moins le composant racine) ainsi que la création d'un driver de test global capable d'exercer les composants les plus enfouis (les feuilles). L'avantage vient de ce que le driver de test résultant peut servir immédiatement pour l'étape de test système. Cette approche reste cependant la plus coûteuse et la moins sûre dans la mesure où il n'existe pas de moyen d'estimer la qualité du driver global.

Ces exemples représentent le cas idéal, pour lequel il n'existe pas d'interdépendance (directe ou induite par un cycle de dépendances) entre les composants. Dans ce dernier cas, fréquent dans le domaine objet à l'intérieur d'un package de classes, il faudra nécessairement réaliser un *stub* ou "bouchon de test". En effet, un composant A utilisant un composant B ne peut fonctionner sans ce composant. En l'absence de B (ou si B n'a pas été déjà intégré au système), on devra utiliser un composant fictif en substitut de B, permettant à A de fonctionner (de s'exécuter en envoyant des réponses suffisamment réalistes). Un *stub* correspond donc dans le meilleur cas un "simulateur" d'un composant, qui a le comportement simplifié du composant complet. Dans le cas d'une interdépendance, B dépend aussi de A, il est clair qu'un *stub* au moins est nécessaire.

D'ores et déjà, il faut préciser (du fait de l'existence de cycles) que ce problème est un problème NP-complet, et que l'algorithme permettant d'atteindre l'optimum est en

coût exponentiel avec le nombre de composants du système, ce qui revient à dire qu'il faut déterminer une heuristique —une stratégie d'intégration— la plus efficace possible.

La Figure 10.5 décrit une intégration Bottom-up, dans une architecture sans cycle, et donc pour laquelle on n'a pas besoin de *stubs*. En ce qui concerne le test orienté objet, une solution consiste à décomposer le programme en parties hiérarchiques fonctionnellement cohérentes. Dans de telles approches, la décomposition offre un canevas pour le test unitaire, d'intégration et système. Ces modèles contraignent la conception à diviser le système en petites parties respectant la complexité du comportement.

10.3.3 Le test de systèmes à objets

Il existe peu de techniques de test système dédiées à l'orienté-objet. Un système peut-être considéré comme un composant, au même titre qu'une classe peut-être vue comme un composant unitaire. De ce point de vue, l'approche "composant" présentée précédemment s'applique aussi au niveau d'un système.

Toutefois, lorsqu'on parle de système, on considère en général un logiciel complet et complexe décrit à l'aide d'un cahier des charges précis. Dans le monde objet, le dossier d'analyse décrit avec la notation UML sert de référence. Les tests système devront couvrir chaque cas d'utilisation (use-case), et pour chacun d'eux couvrir les diagrammes de séquence associés (ou scénarios).

Des méthodes plus élaborées imposent au dossier d'analyse de spécifier les dépendances entre cas d'utilisation à l'aide de diagrammes d'activités (activity diagram). Les tests système devront alors couvrir les combinaisons de cas d'utilisation correspondant à des chemins dans le diagramme d'activité, de manière analogue à la couverture de chemins dans un graphe de contrôle. Dans ces méthodes, on distingue les scénarios nominaux correspondant à une utilisation normale de la fonction, des scénarios exceptionnels qui correspondent à des utilisations rares ou même à des échecs de la fonction.

Par exemple, on peut considérer un système comportant trois cas d'utilisation CU1, CU2 et CU3, chacun ayant 3 scénarios associés (2 nominaux et 1 exceptionnel correspondant au cas d'échec). Si les cas d'utilisation s'appliquent nécessairement dans l'ordre :

$$CU1 \rightarrow CU2 \rightarrow CU3$$

il faudra tester les 3 scénarios de chaque cas d'utilisation à travers tous les scénarios nominaux y menant, ce qui fera $(3 + 2 * 3 + 2 * 2 * 3) = 21$ cas de test (en fait 17 si l'on supprime les scénarios qui se recouvrent).

Bibliographie

[1] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language*. Oxford University Press, New York, 1977.

[2] Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, 1998.

[3] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *IEEE Computer*, 13(7), July 1999.

[4] B. W. Boehm. The high cost of software. In Ellis Horowitz, editor, *Practical Strategies for Developing Large Software Systems*. Addison-Wesley, 1975.

[5] B. W. Boehm. Improving software productivity. *Computer*, 20(9):43–57, September 1987.

[6] F. Buschmann, R. Meunier, P. Sommerland, and M. Stal. *Pattern Oriented Software Architecture, A System of Patterns*. Wiley & Sons, British Library, 1996.

[7] J. Coplien. *Advanced C++: Programming Styles and Idioms*. Addison-Wesley, 1991.

[8] J. Coplien and D. Schmidt, editors. *Pattern Languages of Program Design*, volume 1. Addison Wesley, 1995.

[9] J.O. Coplien. *Organizational Patterns: Beyond Technology to People*. MA: Addison-Wesley, 2005.

[10] Desmond D'Souza and Alan Wills. *Objects, Components and Frameworks With UML: The Catalysis Approach*. Addison-Wesley, 1998.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[12] M.A. Jackson. *System Development*. Prentice-Hall International, Series in Computer Science, 1985.

[13] J.-M. Jézéquel and B. Meyer. Design by contract: The lessons of Ariane. *Computer*, 30(1):129–130, January 1997.

[14] J.-M. Jézéquel, M. Train, and C. Mingins. *Design Patterns and Contracts*. Addison-Wesley, October 1999. ISBN 1-201-30959-9.

[15] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 1988.

[16] Marvin Minsky. *La société de l'esprit (The Society of Mind).* InterEditions, 1987.

[17] Pierre-Alain Muller. *Modélisation objet avec UML.* Editions Eyrolles, 1997. ISBN : 2-212-08966-x.

[18] H. Rohnert N. Harrison, B. Foote, editor. *Pattern Languages of Program Design*, volume 4. MA: Addison-Wesley, 2000.

[19] D.L. Parnas. Software aspects of strategic defence systems. *Comm. of the ACM*, 28(12), December 1985.

[20] F. Buschmann R. Martin, D. Riehle, editor. *Pattern Languages of Program Design*, volume 3. MA: Addison-Wesley, 2000.

[21] O. Nierstrasz S. Demeyer, S. Ducasse. *Object-oriented reengineering patterns.* Morgan Kaufmann, 2003.

[22] Vlissides, Coplien, and Kerth, editors. *Pattern Languages of Program Design*, volume 2. Addison Wesley, 1996.

[23] John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors. *Pattern Languages of Program Design 2.* Addison-Wesley, Reading, MA, 1996.

[24] Jos Warmer and Anneke Kleppe. *The Object Constraint Language.* Addison-Wesley, 1998.