Programmation impérative et objets

Support de cours

Jacques Malenfant, professeur des universités
avec la participation de
Daniel Deveaux, maître de conférences

UEF 3.3 — Informatique
DEUG MIAS/MASS
2e année

UBS
UNIVERSITÉ DE BRETAGNE-SUD

Table des matières

1 La métaphore des objets **7**

1.1 Ordinateurs et programmation 7

1.2 L'objet pour décomposer et abstraire : une métaphore 13

1.2.1 La puissance de la métaphore objet 13

1.2.2 Exemple : l'agence de location d'automobiles 14

1.2.3 Discussion 21

1.2.4 Un second exemple : le problème des n corps 22

1.3 Le chemin à parcourir 24

1.4 Éléments de savoir-faire : premiers pas en Java 25

1.4.1 Utilisation de Java 25

1.4.2 Écriture, compilation et exécution d'un programme 27

1.4.3 Environnement de développement JDE 28

1.5 Exercices 28

2 Objet = procédures + données **31**

2.1 Introduction aux procédures 31

2.1.1 Des expressions aux fonctions 31

2.1.2 De la variable mathématique à la variable impérative 36

2.1.3 La procédure 43

2.2 Données, encodage et représentation 47

2.3 Objet comme abstraction de la mémoire et des opérations 54

2.4 Éléments de savoir-faire : types primitifs, expressions et méthodes 56

2.4.1 Types de données primitifs et littéraux de Java 56

2.4.2 Expressions en Java 58

2.4.3 Méthodes en Java (première version simplifiée) 59

2.4.4 Assertions en Java 62

2.5 Exercices 63

3 Classes et objets **65**

3.1 L'objet . . . 65

3.1.1 Les principes et leur illustration . . . 65

3.1.2 Création et utilisation des objets . . . 68

3.1.3 Architectures logicielles et coopération entre objets . . . 72

3.2 La classe . . . 74

3.2.1 Classe comme descripteur d'objets . . . 74

3.2.2 Instantiation . . . 77

3.2.3 La méthode . . . 78

3.2.4 Partage d'informations via la classe . . . 80

3.2.5 Héritage . . . 80

3.3 Programmation avec des classes et des objets . . . 85

3.3.1 Analyse des problèmes et identification des classes . . . 86

3.3.2 Développement des classes . . . 88

3.4 Type abstrait de données et classe . . . 93

3.5 Éléments de savoir-faire : utilisation des objets en Java . . . 93

3.5.1 Les objets polynômes de degré 2 . . . 93

3.5.2 Déclaration des variables de l'objet . . . 94

3.5.3 Initialiseurs, modificateurs et accesseurs . . . 97

3.5.4 Création, référence, accès et envoi de message . . . 98

3.5.5 L'héritage en Java . . . 100

3.5.6 Documentation embarquée avec *Javadoc* . . . 102

3.6 Exercices . . . 106

4 Flôt de contrôle **109**

4.1 La séquence . . . 109

4.2 Rupture de séquence et flôt de contrôle dirigé par la syntaxe . . . 111

4.3 Énoncés conditionnels . . . 113

4.4 Énoncés de répétition . . . 121

4.5 L'appel de méthode ou envoi de message . . . 127

4.6 Élément de savoir-faire : énoncés de flôt de contrôle en Java . . . 135

4.6.1 Énoncés conditionnels en Java . . . 135

4.6.2 Énoncés de répétition en Java . . . 137

4.6.3 Maîtrise de la répétition . . . 138

4.7 Exercices . . . 141

5 Les tableaux **143**

5.1 Agrégation de données : la notion de tableau . . . 143
5.2 Quelques algorithmes classiques sur les tableaux . . . 147
5.2.1 Traversée d'un tableau . . . 147
5.2.2 Recherche séquentielle . . . 148
5.2.3 Plus grand élément d'un tableau . . . 149
5.2.4 Recherche dichotomique . . . 151
5.3 Tableaux comme collections de données à taille variable . . . 155
5.3.1 Collections non-ordonnées . . . 155
5.3.2 Collections ordonnées . . . 157
5.3.3 Parcours standardisé des éléments d'une collection . . . 160
5.4 Tableaux et notion de réference . . . 161
5.5 Élément de savoir-faire : tableaux en Java . . . 163
5.5.1 Tableaux versus objets . . . 163
5.5.2 Exemple d'utilisation des tableaux dans les objets . . . 164
5.6 Exercices . . . 170

6 Développement systématique de classes **173**

6.1 Méthodologie de programmation . . . 173
6.2 Spécification fonctionnelle de la classe . . . 177
6.3 Spécification de l'interface et programmation contractuelle . . . 181
6.3.1 Programmation contractuelle . . . 181
6.3.2 Spécification de l'interface de la classe . . . 184
6.4 Définition des tests et classes de test . . . 189
6.4.1 Le test du logiciel . . . 189
6.4.2 Définition des tests sur la classe . . . 193
6.5 Mise en œuvre . . . 193
6.5.1 Définition de la structure et invariants . . . 194
6.5.2 Mise en œuvre des méthodes . . . 197
6.6 Vérification . . . 203
6.6.1 Vérification par les tests . . . 204
6.6.2 Vérification des contrats et des tests . . . 204
6.7 Exercices . . . 205

7 Entrées/sorties et interface graphique **207**

7.1 Élément de savoir-faire : . . . 207

7.2 Exercices 207

A La classe `Ensemble` **209**

B La classe `RelationBinaire` **221**

Bibliographie **229**

Chapitre 1

La métaphore des objets

1.1 Ordinateurs et programmation

Un ordinateur est une entité capable de suivre des ordres, ou plus spécifiquement, d'exécuter des *instructions*, comme des opérations arithmétiques. Les humains ont probablement été les premiers «ordinateurs», mais de nos jours le terme ordinateur désigne plutôt une machine électronique qui peut exécuter des instructions de manière autonome :

> **ordinateur :** *nom masc.* — [...] — 2 - Calculateur électronique doté de mémoires à grandes capacités et de moyens de calculs ultra-rapides, pouvant adapter son programme aux circonstances et prendre des décisions complexes.

L'ordinateur tel que nous le connaissons est maintenant vieux d'une cinquantaine d'années. Nous n'avons pas fini de nous émerveiller devant cette invention humaine qui, pour la première fois sans doute, prolonge non pas un de ses membres mais bien cette faculté centrale qu'est la mémorisation et le traitement de l'information au sens large. L'ordinateur traite donc bien de l'information, mais selon une suite d'opérations bien précises que l'on appelle *programme*.

> **programme :** *nom masc.* — Ensemble ordonné (et formalisé) des opérations nécessaires et suffisantes pour obtenir un résultat ; dispositif permettant à un mécanisme d'effectuer ces opérations.

Les opérations de base qu'un ordinateur sait aujourd'hui effectuer réellement, ou physiquement, demeurent relativement simplistes : copier une donnée d'un endroit à un autre, d'additionner deux données, etc. Un programme exécutable est un assemblage de telles opérations élémentaires qui tire sa puissance en grande partie de la quantité des opérations élémentaires qu'il met en œuvre. Cette taille des programmes est à la fois la force et la faiblesse de l'ordinateur : généralement les problèmes plus complexes ne peuvent être résolus que par des programmes de grande taille. Mais en même temps le développement de tels programmes est extrêmement complexe, en fait peut-être l'activité la plus complexe à laquelle nous nous soyions jamais attaqués.

Tout calcul réalisé sur un ordinateur sur des informations préalablement mémorisées est, de fait, le résultat de l'application d'un programme. Le processus par lequel un ordinateur applique un programme est complexe. Nous n'allons pas le traiter en détail ici. En gros, l'ordinateur va charger le programme en mémoire, puis en lire les instructions et les évaluer les unes après les autres jusqu'à ce qu'il arrive à la fin du programme. Un ordinateur ne sert donc à rien s'il n'a pas d'abord été *programmé*.

programmer : *verbe* — Élaborer un programme.

Programmer est bien l'élaboration d'un programme, mais il a acquis par extension la signification d'en charger la mémoire d'un ordinateur alors prêt à l'exécuter. Programmer, dans son sens premier, consiste à définir les données permettant de représenter les informations du problème au sein de l'ordinateur, puis à exprimer les traitements ou calculs à réaliser sur ces données.

L'élaboration de programmes s'est révélée l'une des activités intellectuelles les plus complexes jamais entreprises par l'homme. Les questions fondamentales posées à chaque fois sont :

Comment passer de l'expression d'un problème à l'expression de sa solution sous la forme d'un programme ?
Comment structurer les informations complexes de manière à faciliter l'expression de la solution au problème ?
Comment exprimer de manière correcte et complète un certain traitement sur des informations données ?

Les premiers traitements envisagés étaient déjà complexes : calculs de statistiques descriptives (moyennes, etc.) sur des échantillons de grande taille (recensement), simulations numériques pour la mise au point de la bombe atomique, etc. Aujourd'hui, les traitements réalisés par les ordinateurs sont devenus d'un très grande complexité.

L'ordinateur est effectivement utilisé dans la plupart des systèmes de gestion au sens large, qui impliquent la saisie, la mémorisation, la tenue à jour et le croisement d'immenses banques de données. À l'autre extrémité du spectre en quelque sorte, il est utilisé dans l'industrie du film et de la télévision pour la synthèse d'images qui demande à simuler les lois physiques de la propagation de la lumière. Ces dernières applications sont centrées sur les calculs plutôt que sur le traitement de l'information.

L'ordinateur est aussi de plus en plus utilisé pour piloter des équipements très complexes comme les téléphones portables, les réseaux de télécommunications, les avions, les centrales nucléaires, etc. Cette informatique en prise directe avec des équipements qui doivent être pilotés en temps réel est appelée *informatique embarquée*.

Tout programme est exprimé dans un certain langage dit de programmation. Un ordinateur est construit autour de son processeur, organe central qui est capable d'interpréter et d'exécuter des instructions. L'ensemble des instructions «comprises» par un processeur s'appelle son *jeu d'instructions*, forcément limité en taille. Le langage de programmation constitué des instructions d'un processeur est appelé *langage machine*.

Les langages machines sont conçus pour être exécutés de manière efficace par les processeurs physiques (ou réels). Pour cela, ils sont volontairement constitués d'instructions simples, comme l'addition de deux opérandes numériques. Construire un programme complexe à partir d'instructions machines demandent d'assembler un très grand nombre d'instructions. Se crée alors un fossé entre l'expression d'une solution par un être humain, en termes d'opérations de très haut niveau proches de l'expression même du problème, et les instructions de très bas niveau orientées vers les capacités de la machine.

Pour combler ce fossé, des langages de programmation plus proche de l'expression humaine des choses ont été développés. Ils sont génériquement appelés *langages de haut niveau*, dont fait partie le langage *Java* que nous allons étudier dans ce document. Ces langages exigent toujours une expression très précise et non-ambigüe des programmes, de manière à être exécutés par une machine, mais permettent d'utiliser et de définir des opérations de haut niveau. Un programme en langage de haut niveau n'est pas directement exécutable par un processeur. Il doit être

traduit en langage machine, selon un processus appelé *compilation*, ce qui est réalisé par des outils logiciels : les *compilateurs*.

Programmer, c'est gérer la complexité

La complexité exacerbée des traitements ou des calculs à réaliser aujourd'hui entraîne une complexité correspondante des programmes. En une cinquantaine d'années, nous sommes passés de logiciels de quelques milliers d'instructions machine à des systèmes qui dépassent aujourd'hui les dix millions de lignes de code écrites dans des langages de haut niveau. L'informatique en tant que science et technique est née du besoin de gérer cette complexité.

> **informatique :** *nom masc.* — Science de l'information ; ensemble des techniques de collecte, du tri, de la mise en mémoire, de la transmission et de l'utilisation des informations traitées automatiquement à l'aide de programmes (logiciels) mis en œuvre sur ordinateurs.

L'informatique en tant que science étudie la complexité intrinsèque des problèmes de traitement de l'information et des modèles de calcul nécessaire à leur résolution. Le genre de questions qu'elle pose est par exemple : "Quel est le nombre minimal d'échanges de valeurs qu'il faut faire pour trier n valeurs ?" C'est ce que l'on appelle la complexité de calcul. Ou encore : "Existe-t-il des problèmes pour lesquels aucun programme ne pourra jamais être construit qui pourra traiter n données en exécutant un nombre d'instructions polynomial en n ?" C'est ce qu'on appelle aussi la calculabilité. La calculabilité dépend directement du modèle de calcul (quelles instructions ?) utilisé ; l'informatique s'intéresse donc aussi aux modèles de calcul.

Mais l'informatique est tout autant une technique, celle de la construction d'applications correctes et complètes qui réalisent certains calculs ou qui remplissent certaines tâches. Cette dualité science/technique de l'informatique en fait un domaine riche et varié qui tient aujourd'hui une place prépondérante dans l'économie. Cette place est en partie due à l'activité économique générée par l'industrie de l'informatique elle-même, mais bien sûr aussi par le fait que l'informatique est au service de l'ensemble des industries (commande et contrôle, simulation numérique, ou images de synthèse pour les médias, par exemple) et des administrations (traitement de l'information) pour lesquelles elle est devenue un outil indispensable.

La science et la technique informatiques se sont intéressées depuis longtemps à rendre l'expression des programmes plus simple et plus sûre. Construire des applications de plusieurs millions de lignes n'est pas une mince affaire. L'informatique s'est donc particulièrement intéressée à proposer des langages de programmation, des processus et des méthodes de développement aptes à aider les programmeurs dans leur tâche.

L'art de la résolution de problèmes

Dans cette quête, comme toutes les autres activités intellectuelles, l'informatique se heurte aux limites fondamentales de l'être humain et en particulier sa capacité à traiter relativement peu d'information à la fois. On dit que le cerveau humain ne peut traiter beaucoup plus de sept informations «élémentaires» différentes à la fois ; ce serait la capacité de sa mémoire à court terme qui est directement impliquée dans le raisonnement conscient.

On sait depuis longtemps que cette limite d'environ sept éléments n'implique pas de «taille» particulière de l'information à traiter. On peut raisonner sur sept petits nombres entiers (écolier résolvant un problème) ou encore sur la navigation de sept avions de ligne (par un contrôleur aérien). Il suffit que chacune de ces informations soit *élémentaire du point de vue du raisonnement en cours*, c'est-à-dire manipulable directement, sans autre information consciente. Sept éléments d'informations peuvent donc être, selon le raisonnement à tenir, sept mots, sept rela-

tions, sept théorèmes, sept individus, sept pièces de machines, sept morceaux de programmes, sept organisations, peu importe pourvu que le cerveau puisse les appréhender chacun comme un tout dont les propriétés utiles au raisonnement en cours lui sont suffisamment connues pour être manipulés directement. Ainsi, le raisonnement s'*abstrait* des (ou néglige simplement les) détails ne relevant pas des propriétés qui lui sont utiles.

Programmer, comme toute activité du raisonnement humain en général, doit intrinsèquement tenir compte de ces limites. Lorsqu'il s'agit de construire un programme, l'informaticien est vite confronté à la manipulation d'instructions qu'il va assembler pour réaliser un certain calcul. Une fois qu'un bout de programme est écrit, il peut servir de base à la construction d'autres bouts de programmes, et ainsi de suite. Le programme entier exprime un calcul qui résoud un problème du monde réel. Pour mener à bien cette tâche, il faut que le programmeur puisse raisonner sur les bouts de programmes en ne retenant à tout instant que les propriétés essentielles à la tâche en cours de réalisation.

L'un des objectifs fondamentaux de l'informatique a donc été de fournir au programmeur un support à son raisonnement en lui permettant de décomposer sa tâche en sous-tâches, de se concentrer sur une tâche à accomplir, puis à en réutiliser le résultat pour accomplir d'autres tâches. Ces supports, qu'ils soient langages de programmation ou processus et méthodes de développement, n'ont qu'un seul objectif : délimiter la portée des informations à manipuler de façon à rester dans les limites de ce que peut absorber le cerveau humain. Délimiter implique de placer des frontières opaques au-delà desquelles une information n'est plus visible. Les mécanismes permettant de placer de telles frontières dans les langages de programmation sont les mécanismes de *décomposition*, d'*abstraction*, d'*encapsulation* et de *composition*.

Décomposer, c'est diviser pour règner

Comment se réalise le découpage de tâches complexes en sous-tâches plus simples ? Au plus bas niveau, une tâche peut être de réaliser quelques traitements simples sur quelques éléments d'information : calculer la racine carré d'un nombre, ou encore le discriminant d'une équation du second degré. Au niveau immédiatement supérieur, on peut se servir de la séquence d'instructions calculant le discriminant d'une équation du second degré pour en déduire le nombre de racines de l'équation et, éventuellement, les calculer. À un plus haut niveau encore, on peut utiliser un tableur pour produire une figure en pointes de tarte qui sera ensuite utilisée par un traitement de texte pour être insérée dans un document. L'important à chaque niveau est qu'une tâche soit composée d'un petit nombre de sous-tâches sur lesquelles seul un petit nombre d'informations et d'éléments sont nécessaires à leur compréhension. Ils doivent être assez simplement caractérisés pour être manipulés comme des informations élémentaires par le cerveau.

Exemple 1 Considérons la tâche consistant à se rendre en automobile de Vannes à la tour Eiffel. Un premier découpage en sous-tâches peut prendre la forme suivante :

1. Rejoindre la route de Rennes.

2. Se rendre jusqu'au périphérique de Paris.

3. Aller du périphérique sud jusqu'à la tour Eiffel.

Chaque point peut ensuite être redécomposé en sous-sous-tâches. Si on considère la deuxième sous-tâche, consistant à se rendre au périphérique parisien après s'être engagé sur la route de Rennes près de Vannes, on peut la redécouper en :

2.1 Suivre la quatre-voies jusqu'au périphérique de Rennes.

2.2 Contourner Rennes par son périphérique nord.

2.3 Suivre la quatre-voies jusqu'à l'entrée de l'autoroute A83.

2.4 Suivre l'A83 jusqu'au Mans où elle se «jette» dans l'A11.

2.5 Suivre l'A11 jusqu'au périphérique de Paris.

□

Le principe de la décomposition en sous-tâches consiste à redécouper récursivement chaque point jusqu'à ce que les tâches obtenues soient élémentaires du point de vue de l'agent qui les exécutera. Pour un conducteur français, on peut supposer que les points 2.1 à 2.5 sont élémentaires. Par contre, si on doit détailler le point 3, on n'obtiendra pas la même chose selon que l'on s'adresse à un conducteur parisien chevronné ou à un conducteur vannetais n'étant jamais allé à Paris. Si le point 3 paraît suffisamment détaillé au premier, cela sera insuffisant pour le second. L'objectif d'un langage de programmation de haut niveau est de définir des actions qui soient proches de ce que le programmeur considère comme élémentaire.

En tant que véhicule permettant d'exprimer une solution à un problème, tout langage de programmation fournit essentiellement, outre ses actions élémentaires, deux autres types de mécanismes : des mécanismes d'*abstraction* (et donc d'encapsulation) et des mécanismes de *composition*.

Abstraire, pour ne pas se noyer dans les détails

De manière générale, l'*abstraction* est le processus mental qui permet à l'esprit de se concentrer sur les détails pertinents à un raisonnement tout en ignorant ceux qui sont momentanément insignifiants. Pour raisonner sur la manière de se rendre de Vannes à la tour Eiffel en voiture, il est parfaitement inutile de se préoccuper du fonctionnement du moteur à explosion. Dans ce raisonnement, vous vous contenterez d'une vision abstraite de la voiture en tant que véhicule dans lequel on monte, que l'on démarre puis que l'on conduit avec un volant, et un certains nombre de pédales, de manches et de boutons. Vous allez vous abstraire des détails insignifiants pour atteindre votre objectif, tels que le cycle du moteur à 4 temps. De même, vous n'allez pas raisonner sur une carte au 25000^{e}, mais plutôt sur une carte routière ne présentant que les grands axes routiers. Vous choisissez donc un niveau d'abstraction répondant au problème posé.

L'abstraction est un outil très puissant et essentiel à tout mécanisme de raisonnement ou de résolution de problèmes complexes. Sa mise en œuvre est indissociable des mécanismes d'encapsulation évoqués ci-haut, qui permettent de masquer effectivement l'information momentanément non-pertinente dont on cherche à s'abstraire. Encapsuler, c'est construire une frontière autour d'une entité qui soit opaque à tout ce qui n'est pas nécessaire au raisonnement en cours. Par exemple, on peut raisonner sur une fonction mathématique à partir de son comportement en termes des résultats obtenus à partir des entrées, sans savoir comment la fonction sera effectivement calculée.

Composer, ou résoudre un gros problème à partir de petites solutions

La *composition*, elle, est liée à un autre aspect très important des mécanismes de résolution de problème : l'approche diviser pour règner vise la décomposition de problèmes en sous-problèmes, et ce récursivement jusqu'à ce que les sous-problèmes soient «faciles» à résoudre. Lorsqu'un problème est trop complexe pour être résolu directement, il est divisé en sous-problèmes plus simples. Si les sous-problèmes sont encore trop complexes, ils sont subdivisés à leur tour, et ainsi de suite jusqu'à obtenir des sous-problèmes élémentaires résolvables directement. En terme de programmation, un sous-problème est résolvable directement si on peut écrire un énoncé, ou une ligne de programme, qui en donne la solution.

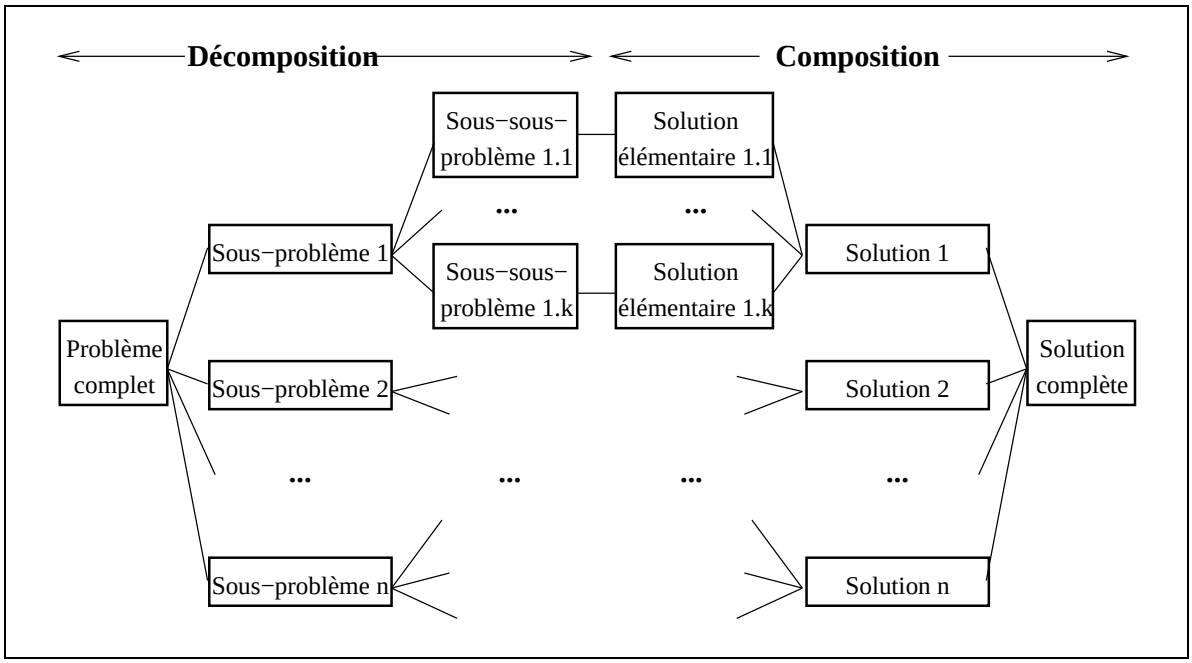


FIG. 1.1 – Développement par décomposition des problèmes puis composition des solutions partielles.

Cette approche de décomposition en sous-problèmes nécessite des mécanismes de décomposition produisant des solutions partielles hiérarchisées, mais également des mécanismes de composition des solutions des sous-problèmes pour produire la solution au problème global à partir des solutions aux sous-problèmes. Les solutions partielles sont généralement liées à des abstractions d'une partie du problème, et découlent donc de la faculté d'abstraction du langage. Les mécanismes de composition permettent la recomposition hiérarchique des solutions partielles pour obtenir la solution complète.

Dans le domaine des traitements, le mécanisme de composition le plus simple et le plus courant est la composition séquentielle : faire d'abord ceci, puis cela (comme dans les recettes de cuisine). La composition fonctionnelle est aussi bien connue en mathématique : calculer f puis appliquer g sur le résultat obtenu. Les données aussi peuvent être décomposées et composées : une adresse se compose d'un numéro civique et d'un nom de rue ; pour l'imprimer, imprimer d'abord le numéro, puis le nom de la rue.

La figure 1.1 résume l'approche de développement par décomposition récursive des problèmes en sous-problèmes puis composition des solutions partielles vers la solution globale. Chaque solution partielle forme une abstraction pour la solution immédiatement englobante, et la frontière autour de cette solution partielle doit être comprise comme une frontière d'encapsulation qui ne laisse filtrer que les éléments strictement nécessaire à sa composition avec les autres solutions partielles pour arriver à la solution immédiatement englobante.

Dans le présent document, nous allons nous intéresser à l'évolution de la notion de programmation depuis les «instructions» types de l'ordinateur de von Neumann à la programmation impérative structurée et nous allons étudier plus profondément la programmation par objets comme moyen ultime, aujourd'hui, de structurer la pensée du programmeur autour d'entités qui peuvent être manipulées comme des informations élémentaires bien définies.

En effet, en cinquante ans de progrès, l'informatique a utilisé plusieurs concepts et mécanismes permettant de décomposer, d'abstraire et de recomposer. La décomposition procédurale

(en procédures) puis fonctionnelle (en fonctions) en offrent deux exemples éminents. Sans vouloir entrer trop vite dans les détails, le principal reproche fait à ces deux approches est de ne se concentrer que sur les calculs ou traitements à faire et de négliger (ou de traiter séparément) les données. Données et traitements étant souvent intimement liés, ces deux approches se sont révélées incomplètes.

1.2 L'objet pour décomposer et abstraire : une métaphore

Pour appréhender le monde, l'humain a toujours eu recours à l'analyse comparative. Cette méthode permet de s'appuyer sur une connaissance bien maîtrisée, qui sert de point de comparaison, pour essayer de comprendre un domaine moins bien maîtrisé en trouvant des similitudes qui permettront ensuite d'appliquer au domaine partiellement connu un raisonnement acquis dans le domaine connu. Le domaine bien connu et ses théories sous-jacentes forment alors ce que l'on peut appeler une métaphore explicative du domaine partiellement connu.

Prenons l'exemple de la théorie de l'évolution. Au 19^{e} siècle, Darwin a parcouru le monde et observé les écosystèmes riches des forêts équatoriales. De ses observations, il en a tiré la théorie de l'évolution par sélection naturelle, selon laquelle le monde vivant évolue par un processus assimilable à de l'essai et erreur. De nouveaux êtres vivants apparaissent par mutation d'êtres existants. Ces êtres sont ensuite mis en compétition pour leur survie avec les autres êtres vivants dans l'écosystème. S'ils survivent, la mutation est transmise à leurs générations suivantes et prolifère. S'ils s'éteignent, la mutation disparaît aussi vite qu'elle est apparue. On dit qu'il y a sélection naturelle.

Extirpée du monde du vivant et des écosystèmes naturels, la théorie de l'évolution a peu à peu acquis la forme d'une métaphore générique, applicable à d'autres situations. En effet, l'évolution à la Darwin a fait l'objet de tentative d'application à divers domaines des sciences sociales et à l'économie des entreprises, par exemple. Il est courant aujourd'hui d'entendre dire que les entreprises naissent et meurent selon un processus de sélection naturelle, les mieux adaptées survivant aux autres. De même, le darwinisme social a nourri certaines pensées libérales extrêmes. À chaque fois, il s'agit d'appliquer l'idée des mutations suivies de sélection pour expliquer l'évolution de systèmes complexes.

1.2.1 La puissance de la métaphore objet

Par rapport à la programmation, la puissance d'une métaphore vient du fait qu'elle permet de récupérer un vaste corpus de connaissances sur l'organisation et la réalisation d'une production de type intellectuel pour l'appliquer au domaine de la programmation des ordinateurs. Ce faisant, la métaphore permet d'arnacher et de vaincre la complexité inhérente à la programmation.

La programmation par objets procède de cette approche métaphorique. Elle propose de voir la solution à un problème de programmation selon la métaphore du *calcul comme simulation de la réalité*. La solution se présente ainsi sous la forme d'une collection d'entités indépendantes, les *objets*, qui coopèrent pour réaliser un calcul en échangeant des requêtes, les *messages*, portant sur du travail à accomplir. Un objet modélise une entité réelle du domaine duquel est tiré le problème. L'objet possède un état interne dont les valeurs successives reflètent son évolution en cours de calcul, de la même façon qu'une entité du monde réel possède aussi un état. La métaphore consiste donc à voir la programmation comme la construction d'un scénario où chacun joue son rôle en utilisant son savoir-faire propre.

Les messages qui sont adressés aux objets forment autant de requêtes pour l'exécution d'un bout de programme, faisant partie du savoir-faire de l'objet, que l'on va appeler *méthode*. Un message indique le nom de la méthode à exécuter et comporte aussi des informations additionnelles (*arguments*) nécessaires à la réalisation de la tâche demandée. Le *receveur* est l'objet auquel le message est envoyé. Si le receveur accepte le message, il accepte également la responsabilité de réaliser l'action demandée. Le comportement des objets est donc défini en termes de responsabilités qu'ils assument au sein de l'ensemble. L'interprétation d'un message (c'est-à-dire la méthode qui sera exécutée) dépend du receveur du message et peut donc varier de receveur en receveur.

Muni de ce vocabulaire, la métaphore objet se résume par une vision du calcul comme une simulation des entités du monde réel et de leurs actions. Les deux maîtres-mots de la conception de programmes en approche objet sont *comportement* et *responsabilité*.[1] Tout objet possède un comportement, défini par ses méthodes, que l'on peut activer par envoi de message. L'activation d'un objet (par envoi de message) vise à lui faire réaliser une tâche précise dont le résultat est attendu par l'objet qui a expédié le message. Il est de la responsabilité de l'objet activé de réaliser la tâche demandée, quitte à ce qu'il active à son tour d'autres objets pour l'aider dans cette tâche.

1.2.2 Exemple : l'agence de location d'automobiles

Considérons un exemple, celui d'une l'agence de location d'automobiles. L'application que l'on souhaite développer est une agence de location automatique où le client se présente à une borne informatique, entre ses requêtes (réservation, prise ou retour de l'automobile) et paie sa location en utilisant sa carte bleue. Nonobstant le problème de l'accès aux clefs de la voiture et celui de la vérification de l'exactitude des informations (kilométrages parcouru, plein fait au retour, etc.), cette application se rapproche de ce que l'on voit aujourd'hui dans l'industrie hôtelière. Plus précisément, le problème s'énonce comme suit.

Énoncé du problème

L'agence de location possède un parc d'automobiles qu'elle loue à des clients moyennant paiement. L'agence propose trois catégories d'automobiles, des petites, des intermédiaires et des luxueuses, pour lesquelles ils existent différents tarifs : partie fixe à la journée plus une partie au kilomètre parcouru ou encore forfait à durée fice mais à kilométrage illimité, etc. Une agence reçoit de la part des clients les demandes de réservation précisant chacune la catégorie d'automobiles choisie et une plage de dates définie par une date de prise de l'automobile et une date de retour prévue. Une réservation est donc constituée d'une date de prise, d'une date de retour prévue, d'un tarif de location et du numéro de réservation délivré au client.

L'agence répond à une demande de réservation soit en retournant un numéro de réservation, soit en signifiant qu'il n'y a pas de voiture de la catégorie demandée disponible pour la plage de dates requise. La réservation faite, l'agence la mémorise dans une liste de réservations. Cette liste est conçue de telle façon qu'on puisse retrouver ou retirer une réservation en fonction de son numéro. L'agence doit aussi pouvoir consulter la liste des réservations prises pour trouver une voiture libre lorsqu'un client demande une réservation.

Si la réservation a été faite, l'agence reçoit le client le jour convenu pour qu'il prenne sa voiture. Pour obtenir son automobile, le client donne son numéro de réservation, et l'agence

[1] En anglais, on dit que l'approche objet est «*behavior-centered*» et «*responsibility-driven*».

lui répond en lui notifiant le numéro d'immatriculation de la voiture retenue. Clairement, une automobile doit pouvoir accepter plusieurs réservations si les plages de dates ne se recouvrent pas. Lors d'une demande de réservation, l'agence doit prendre les dates de prises et de retour proposées et trouver parmi son parc une automobile de la catégorie demandée qui soit libre pour cette plage de dates.

Une automobile possède une immatriculation, une catégorie, et un kilométrage qui augmente lorsque l'automobile parcourt des kilomètres. Elle est aussi dotée d'une mémoire permettant de mémoriser le kilométrage au moment de la prise, de manière à calculer le kilométrage total parcouru par le client. Au retour, le client rend l'automobile en donnant son numéro de réservation. L'agence note alors le kilométrage parcouru par le client avec l'automobile et établit le prix de la location qui est communiqué au client.

Le tarif de la location varie en fonction de la catégorie de l'automobile louée ; ce tarif se compose généralement d'un tarif de base par journée et d'un tarif au kilomètre parcouru. D'autres politiques tarifaires sont susceptibles de s'ajouter par la suite. On calcule le prix d'une location en additionnant le tarif de base journalier multiplié par le nombre de jours de location et la multiplication du tarif kilométrique par le nombre de kilomètres parcourus.

Le client joue le rôle actif dans l'application. C'est à son initiative que les opérations sont réalisées, que ce soit la réservation, la prise, le parcours de kilomètres ou le retour de l'automobile. Toutes ces opérations seront réalisées via l'interface de l'application. Lorsqu'un client souhaite faire une location, il s'adresse à l'agence en demandant la réservation d'une automobile de catégorie donnée (petite, intermédiaire ou luxueuse) pour une plage de dates donnée. À une requête de réservation, l'agence répond au client en lui donnant un numéro de réservation (si une voiture est disponible). Le client prend la voiture de la catégorie choisie le jour de sa réservation auprès de l'agence en fournissant son numéro de réservation. L'agence lui fournit une automobile.

Lorsque le client a récupéré l'automobile, il lui fait parcourir des kilomètres, puis il la rend à l'agence. Lors du retour, l'agence calcule le prix de la location et le retourne au client pour paiement. Si la date de retour effective précède la date de retour prévue, la location des jours retenus doit être payée entièrement. Si la date de retour effective dépasse la date de retour prévue, la totalité de jours effectifs d'utilisation de l'automobile seront facturés au client. Pour simplifier, on ne considèrera pas les opérations de paiement.

Première décomposition

En gros, une première décomposition objet fait apparaître des entités réelles, comme l'agence, les tarifs, les automobiles, les dates, les réservations et la borne ou guichet automatique. Ces entités sont représentés informatiquement par des objets (fig. 1.2). La simulation procède par analogie avec le comportement des entités réelles. L'objet `client` cherche à louer une voiture. Dans un premier temps, il faut trouver un objet `agenceLocation` auquel il pourra transmettre sa requête sous la forme d'un *message*. L'objet `agenceLocation` pourra répondre à ce message en retournant, comme résultat de l'exécution de la *méthode* appropriée, un objet `contrat` proposant à l'objet `client` les termes de la location. Ce dernier pourra ensuite faire une réservation par l'envoi d'un message à l'objet `agenceLocation` comprenant l'objet `contrat` approuvé en paramètre. La réservation, comme la proposition initiale, sont des opérations qui sont déléguées par l'objet `client` à l'objet `agenceLocation`. Ce sera la *responsabilité* de l'objet `agenceLocation` de répondre à ces requêtes. L'objet `client` n'a pas besoin et ne désire pas connaître le détail du traitement de ces requêtes ; cette information lui est *cachée*.

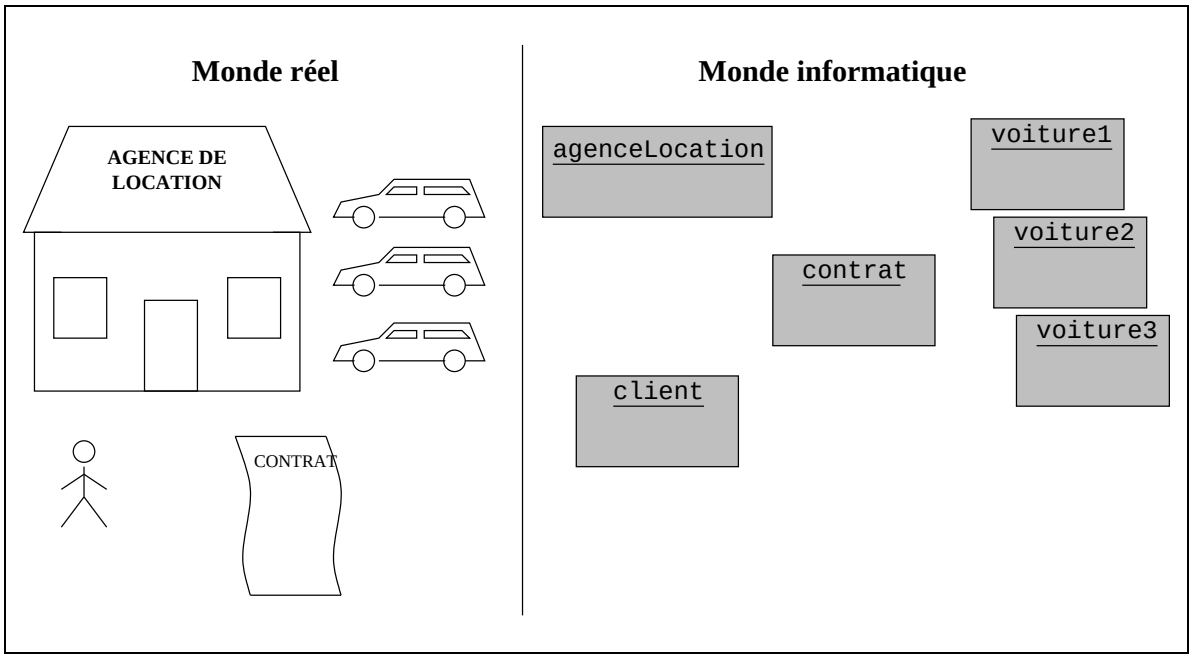


FIG. 1.2 – Calcul comme simulation

En effet, il importe peu à l'objet `client`, tout comme au client réel d'ailleurs, de savoir que lorsqu'il émet sa requête pour la location d'une voiture, l'agence qu'il a choisi lui réservera une voiture qu'elle possède en propre, ou si cette agence va simplement envoyer un message à une autre agence de son réseau pour obtenir une voiture qui correspond à la demande du client. Le fait pour l'objet `client` de s'abstraire de ce genre de détails simplifie grandement son propre comportement. Pour chaque requête qu'il a à émettre, un objet n'a comme seul objectif que de trouver un autre objet capable de satisfaire cette requête et de la lui expédier sous la forme d'un message.

Intéressons-nous maintenant à la borne d'interaction graphique. Les figures 1.3 1.4 1.5 et 1.6 donnent une illustration de ce qu'il est concevable d'obtenir assez facilement en Java. La figure 1.3 montre le menu d'interaction principal où le client choisit une opération parmi les cinq possibles.[2] Si le client choisit l'opération «Réserver une automobile», une boite de dialogue illustrée à la figure 1.4 apparaît. Cette boite de dialogue demande de préciser par des menus la date de prise, la date de retour et le type d'automobile. Cela fait, comme illustré à la figure 1.5, il termine son opération en appuyant sur le bouton «OK» s'il souhaite poursuivre, ou sur le bouton «Abandon» s'il change d'avis. Dans le cas où il a appuyé sur le bouton «OK», une nouvelle boite de dialogue apparaît pour lui redonner les détails de sa réservation et lui demande de confirmer par «OK» ou d'infirmer par «Abandon».

Cette description de scénario d'utilisation est généralement obtenue à la suite d'échanges entre informaticiens et leurs clients (c'est-à-dire les gens qui ont commandés le développement de l'application).

De la même manière que les entités définies dans l'énoncé du problème, les entités de l'interaction entre l'utilisateur et l'application via la borne graphique vont être simulés informatiquement par des objets. Sur la base du scénario précédent, on voit poindre un objet borne principale pour le menu général de l'application, les objets dialogues pour les principaux items du menu général, dont celui pour la réservation, et les dialogues de confirmation, dont celui

[2] Le choix «Changer la date du jour» est à négliger pour l'instant.

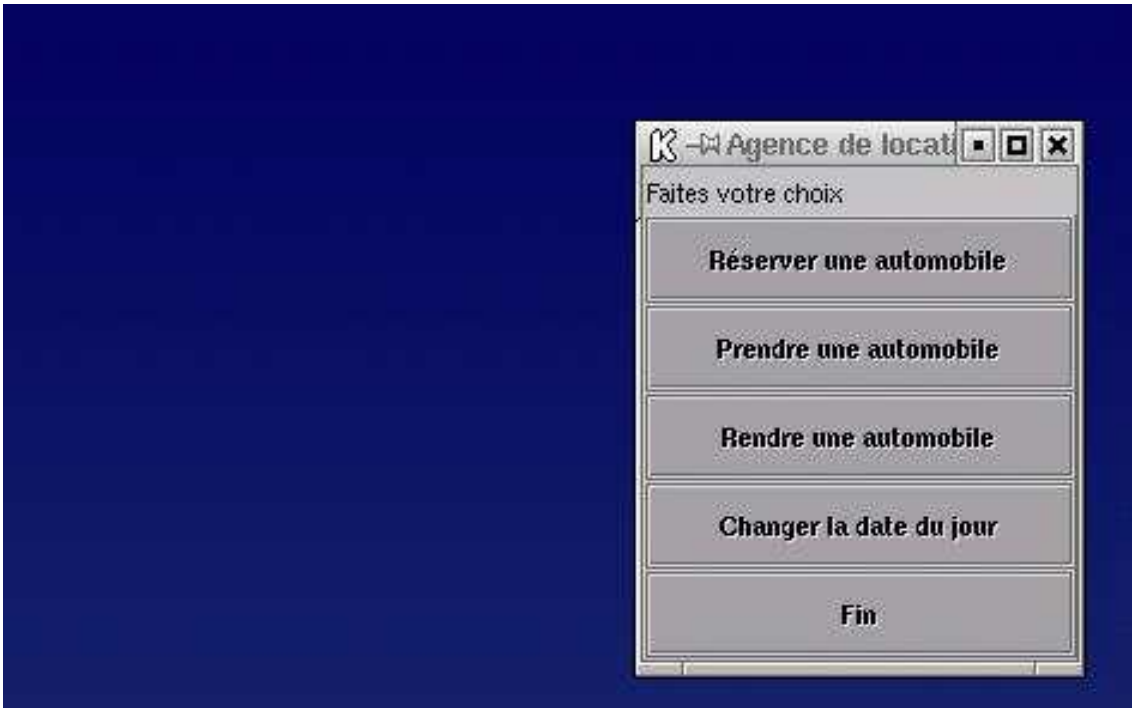


FIG. 1.3 – Lancement de l'application

pour la confirmation de réservation.

Considérons par exemple la boite de dialogue de confirmation de réservation apparaissant à la figure 1.6. A priori, cela paraît très complexe. Il faut une fenêtre avec tout le comportement que cela peut supposer, du texte à afficher, deux boutons «OK» et «Abandon» avec leur comportement. L'objet dialogue lui-même va orchestrer l'affichage de ces différents éléments puis réagir aux événements générés par l'utilisateur, comme les clicks de souris.

Heureusement, les langages comme Java nous offrent des bibliothèques entières de définitions d'objets toutes faites qu'il suffit de réutiliser pour écrire le programme : fenêtres, étiquettes, boutons, événements, etc. La définition de l'objet `DialogueConfirmationResa` en Java est faite comme suit. D'abord, la définition doit indiquer à quelle application elle appartient. La ligne (1) la définit comme le «package» `AgenceLocation`. On doit ensuite indiquer à Java quelles bibliothèques vont être utilisées ; c'est l'objectif des lignes (2-3). Un commentaire explique brièvement ce que fait cet objet (4-11), puis débute la définition proprement dite (12-59). Les lignes (12-14) donnent le nom de l'entité qui est définie, `DialogueConfirmationResa`, puis indique que cette entité doit se comporter comme une entité qui existe déjà, `Dialog`, à laquelle on va ajouter de nouvelles choses. On indique ensuite que `DialogueConfirmationResa` va donner une définition à un comportement standard prédéfini qui s'appelle `ActionListener` et qui va permettre de recevoir la notification des événements générés par l'utilisateur.

```
1.  package AgenceLocation ;
2.  import java.awt.* ;
3.  import java.awt.event.* ;

4.  /**
```

FIG. 1.4 – Faire une réservation

```
5.   * DialogueConfirmationResa
6.   *
7.   *     Cette classe définit le dialogue qui va afficher les informations
8.   *     concernant une réservation et demande la confirmation du client.
9.   *
10.  * @author Jacques Malenfant
11.  */
12. public class            DialogueConfirmationResa
13.                         extends Dialog
14.                         implements ActionListener {
```

Les lignes (15-20) définissent les données de l'objet. L'objet `DialogueConfirmationResa` connaît l'objet borne qui l'a lancé par son élément de donné, appelé variable d'instance, `borne` (16). Il possède deux boutons : le bouton de confirmation `okButton` (18) et le bouton d'infirmation `abandonButton` (20).

```
15.   /** La borne graphique à laquelle le dialogue est attaché     */
16.   private BorneGraphique        borne ;
17.   /** Le bouton de confirmation                                 */
18.   private Button        okButton = new Button("OK") ;
19.   /** Le bouton d'infirmation                                   */
20.   private Button        abandonButton = new Button("Abandon") ;
```

Les lignes (21-41) définissent ce que l'on appelle le constructeur de l'objet `DialogueConfirmationResa`. Ce constructeur est une procédure, appelée méthode dans la terminologie objet, dont le rôle est

FIG. 1.5 – Terminer la réservation

d'initialiser l'objet dialogue lors de sa création. Il reçoit trois paramètres : la borne graphique qui l'a lancé (25), le titre à afficher sur la fenêtre (26) et le message à afficher et qui doit être confirmé (27). Ce message est du genre : "Vous avez réservé une petite voiture du 24 septembre 2001 au 29 septembre 2001."

La ligne (30) est un énoncé Java `super` qui dit que l'initialisation de l'objet `DialogueConfirmationResa` se fait comme l'initialisation de l'objet `Dialog` auquel il ressemble pour ce qui concerne la borne qui l'a lancé et le titre à afficher sur la fenêtre. Ensuite, on conserve dans la variable `borne` cette borne qui nous a lancé, puis on crée les éléments qui doivent apparaître dans la fenêtre : étiquette contenant le message (32), ajouté à la fenêtre (33), puis les deux boutons (35-36) ajoutés sur la fenêtre (39) selon un format de grille 1×2 (34, 37-38). L'énoncé `pack` (40) sert simplement à préparer la fenêtre pour affichage.

```
21.   /**
22.    * Constructeur de dialogue de confirmation de réservation
23.    */
24.   public                DialogueConfirmationResa(
25.       Frame p,          // la borne graphique à laquelle on est attaché
26.       String title,     // le titre à afficher
27.       String message    // le message devant être confirmé
28.       )
29.   {
30.     super(p, title) ;
31.     borne = (BorneGraphique)p ;

32.     Label messageLabel = new Label(message) ;   // étiquette message
```

FIG. 1.6 – Confirmer la réservation

```
33.     add("Center", messageLabel) ;               // ajout dans l'affichage
34.     Panel p1 = new Panel(new GridLayout(1, 2)) ; // affichage en grille
35.     okButton.addActionListener(this) ;          // j'écoute le bouton OK
36.     abandonButton.addActionListener(this) ;     // j'écoute le bouton abandon
37.     p1.add(okButton) ;                  // ajout du bouton OK à la grille
38.     p1.add(abandonButton) ;             // ajout du bouton abadon à la grille
39.     add("South", p1) ;                  // ajout de la grille à l'affichage

40.     pack() ;                            // prépare l'affichage graphique
41.   }
```

Les lignes (46-58) définissent la méthode `actionPerformed` du comportement `ActionListener` que l'objet a promis de mettre en œuvre. Cette méthode est appelée automatiquement lorsque l'utilisateur de la borne va cliquer sur l'un des boutons de la fenêtre de dialogue. Le click se traduira par un objet événement qui sera passé en paramètre à la méthode. cet objet événement contient les informations relatives à l'événement qui vient de se produire, comme le bouton qui a été cliqué. La méthode demande donc à l'événement quelle est la source de l'événement et compare avec ses deux boutons (50, 53). Selon le bouton concerné, l'action entreprise sera de cacher la fenêtre de dialogue (51, 55) puis de mettre à jour dans l'objet borne graphique la variable qui dira si la réservation est confirmée (52) ou non (56).

```
42.   /**
43.    * actionPerformed est la méthode automatiquement appelée lorsqu'un événement
44.    * d'action se produit sur une entité qu'on écoute.
45.    */
46.   public void           actionPerformed(
47.       ActionEvent e     // objet événement qui vient de se produire
```

```
48.       )
49.   {
50.     if ( e.getSource() == okButton ) {  // si la source est le bouton OK
51.         setVisible(false) ;             // rend le dialogue invisible
52.         borne.setConfirmationResa(true) ;       // rapporte l'info à la borne
53.     } else if ( e.getSource() == abandonButton ) {
54.                                         // si la source est le bouton abandon
55.         setVisible(false) ;             // rend le dialogue invisible
56.         borne.setConfirmationResa(false) ;      // rapporte l'info à la borne
57.     }                                           //  if ( e.getSource() == ... )
58.   }
59. }
```

On voit ici comment l'utilisation de nombreux objets et de leurs comportements a permis de casser la complexité du problème en une myriade d'entités ayant un rôle spécifique à jouer, l'orchestration étant réalisée en demandant successivment à ces différents objets de réalier l'une ou l'autre de leurs opérations.

1.2.3 Discussion

Cet exemple illustre bien l'idée du calcul comme simulation. Les entités participant au problème sont représentées par des objets qui en jouent le rôle dans l'univers virtuel du programme. Les actions sont initiées par l'envoi de message et l'action elle-même est modélisée par le code de la méthode exécutée pour répondre au message. Requêtes et méthodes simulent donc les actions réelles des entités réelles. De façon générale, la conception d'un programme en approche objet consiste d'abord à décrire cette simulation sous forme de texte narratif, comme nous venons de la faire, puis à identifier les entités et les actions à représenter. Les entités correspondent le plus souvent à des noms (substantifs) dans le texte ou scénario de la simulation, alors que les actions correspondent le plus souvent aux verbes. Une fois identifiées, les entités et les actions sont modélisées informatiquement par des objets et leurs méthodes.

Tout déconcertant qu'il puisse paraître de prime abord, le vocabulaire de la programmation par objets recouvre bien, comme on l'a vu au chapitre précédent, des concepts bien connus de la programmation impérative. L'état d'un objet est simplement les valeurs affectées à l'ensemble des ses variables. Une méthode n'est autre qu'une procédure qui a ses paramètres formels, qui est appelée avec des paramètres réels et qui peut avoir des effets de bord sur l'état de l'objet qui l'exécute. Un objet est une collection de variables et de procédures.

Pourquoi alors introduire un vocabulaire nouveau ? Simplement pour un question d'efficacité de la métaphore qui va permettre de tirer un savoir sur l'organisation des objets et l'utilisation de leur savoir-faire. La métaphore objet repose simplement sur une espèce d'antropomorphisme qui voit les objets comme des agents et un programme un peu comme une pièce de théâtre ou un film. Le scénario de l'application se formalise peu à peu pour devenir le programme lui-même.

L'intérêt de la métaphore des objets est multiple. Au premier chef, nous avons mentionné cette idée que la métaphore de la simulation permet au programmeur d'aborder les problèmes en utilisant toutes les intuitions usuelles qu'il peut avoir sur les entités et leur comportement dans la vie de tous les jours. Dans la mesure où une grande partie du travail de l'informaticien consiste à formaliser des processus qui existent dans la vie courante (par exemple, administratifs), cette métaphore joue un rôle extrêmement important.

Un autre intérêt des objets est liée à la réutilisation. D'une application à l'autre, dans un

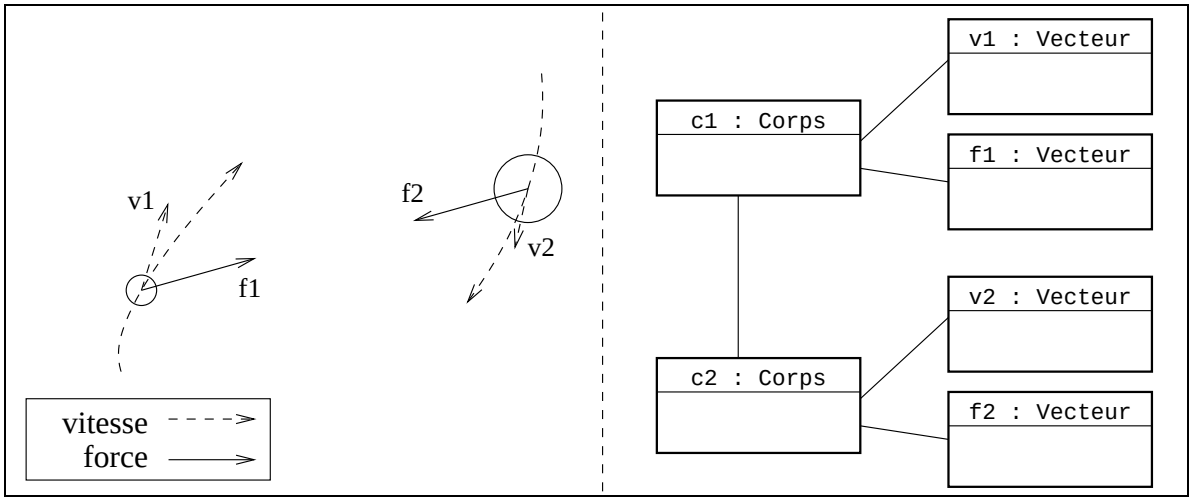


FIG. 1.7 – La simulation du problème des deux corps.

certain champ d'activité, on retrouve souvent plusieurs des objets identifiées dans les applications précédentes. Cela nous amène à une seconde métaphore pour la programmation par objets, c'est-à-dire la construction d'applications à partir des briques de bases normalisées. La programmation est, encore aujourd'hui, une activité en grande partie artisanale. L'objet semble offrir la possibilité d'une rationalisation similaire à celle qui a été vécue dans d'autres industries (automobile, électronique), c'est-à-dire la production d'applications par assemblage de composants logiciels standardisés. Des avancées dans cette direction sont actuellement observées chez les producteurs de logiciels utilisant l'approche objet.

1.2.4 Un second exemple : le problème des n corps

Considérons un second exemple concret tirée de la physique : le problème des n corps. Ce problème consiste à déterminer la trajectoire de n corps soumis à la force d'attraction gravitationnelle. Il a été largement étudié et utilisé à la fois en astrophysique, pour calculer les trajectoires des corps dans l'espace, mais aussi en physique atomique ou dans l'étude des plasmas.

La figure 1.7 illustre la physique du problème limité à deux corps, et indique comment on va le simuler informatiquement avec des objets. Deux corps symétriques exercent l'un sur l'autre une force d'attraction selon des vecteurs $f1$ et $f2$ opposés mais d'égales normes. Cette force s'exerce sur les centres de gravité respectifs des deux corps selon la droite qui relie ces deux centres. qui sont parallèles passe par leurs . Les deux corps se déplacent dans la direction instantanée donnée par leurs vecteurs vitesse $v1$ et $v2$. Un corps possède également une certain accélération. La force de gravitation s'exerce de manière à modifier la vitesse et l'accélération des corps de même que leur direction. La trajectoire des deux corps dans le temps est la solution recherchée.

Ce problème n'a pas de solution analytique dès que le nombre de corps augmente et que les conditions initiales sortent de quelques cas très particuliers. Pour le résoudre, on procède numériquement, en intégrant les équations différentielles. On peut voir cette simulation numérique comme une simulation selon la métaphore objet en adoptant l'approche suivante :

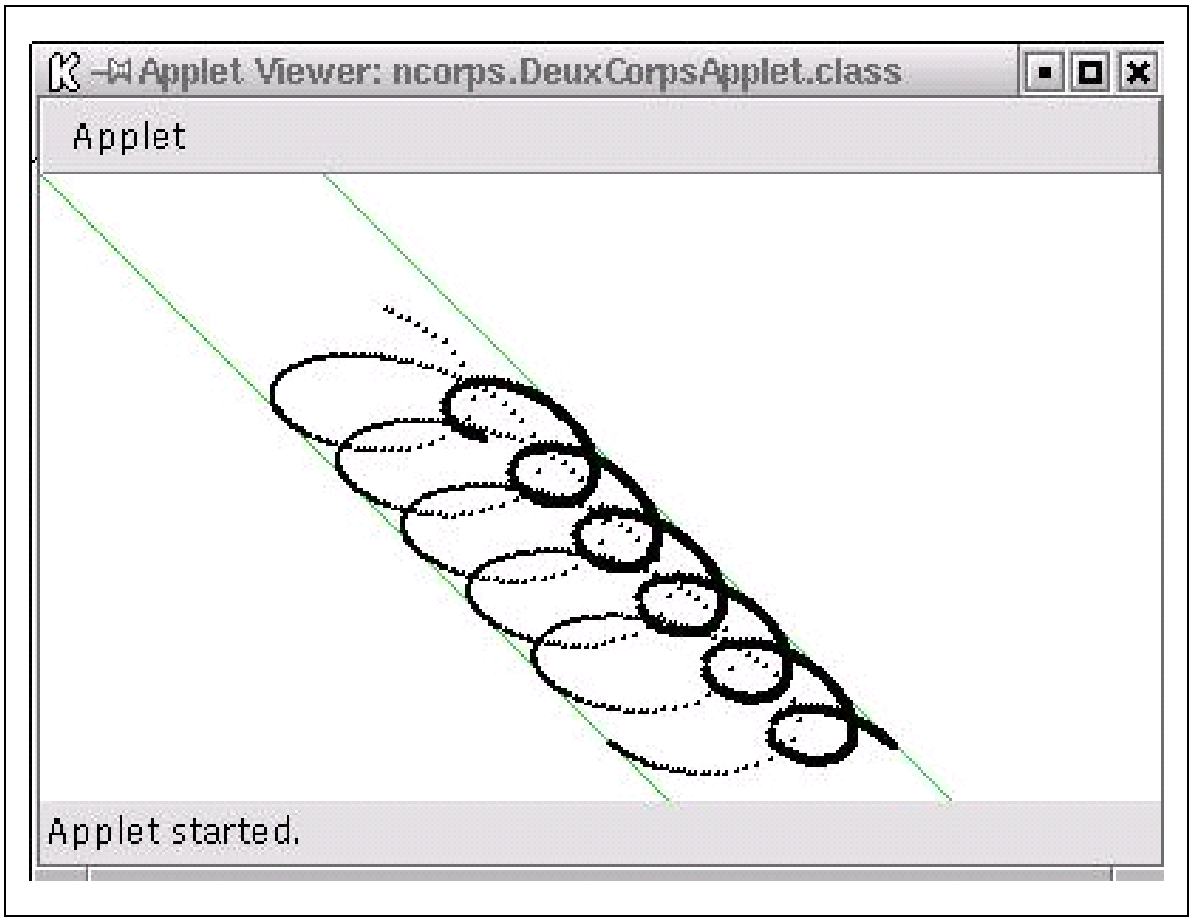


FIG. 1.8 – Une trajectoire à deux corps.

- Chaque corps est représenté par un objet.
- Le suivi du mouvement est assuré en calculant la position des corps à des instants discrets $t = t_0, t_0 + \Delta t, t_0 + 2\Delta t, ...$
- Les caractéristiques essentielles du corps sont : sa position, sa vélocité et sa masse ; chacune est représentée par une variable dans l'objet :
  - Variable `position` : un vecteur à deux dimensions.
  - Variable `velocite` : un vecteur à deux dimensions.
  - Variable `masse` : une valeur réelle.
- À chaque instant $t_n = t_0 + n\Delta t$, on calcule la force exercée par l'autre corps à $t_n - \frac{\Delta T}{2}$ selon la formule :

$$F = \frac{Gm_1m_2}{r^2}$$

  où r est la distance entre les deux corps, m_i la masse du corps i et G la constante de gravitation de Newton. Cette force s'exerce sur un vecteur suivant la droite reliant les centres des deux corps.
- Pour calculer la force exercée par le corps j sur le corps i, i envoie le message `calculeForce` à j, en lui passant en paramètre ses propres masse et position.
- De la force, on obtient l'accélération subie en divisant par la masse, ce qui permet de calculer la vélocité et la position à l'instant suivant t_{n+1}.

- Les calculs de force, d'accélération, de vélocité et de position utilisent des vecteurs : ils sont aussi représentés par des objets.
- Le suivi se faisant à l'écran, chaque objet corps sait se dessiner sur in canevas graphique lorqu'on lui envoi le message `dessineToiSur` en lui passant l'objet représentant le canevas.
- L'objet canevas sait dessiner des rectangles, des ovales, des lignes des arcs, du texte, et ce dans différentes couleurs. L'objet corps n'a pas à se soucier de la façon dont le canevas graphique s'y prend pour ce faire ; il lui suffit d'envoyer les messages de dessin avec les paramètres appropriés.
  Exemple : position du centre plus longeur des deux axes pour une ovale.

Une trajectoire à deux corps apparaît à la figure 1.8. Elle a été obtenue à partir des conditions initiales suivantes. Deux corps sont placés sur un axe horizontal. Le corps de gauche a un poid de moitié inféreiru à celui de gauche. Le corps de gauche possède une vitesse initiale allant vers le bas et celui de droite possède la même vitesse initiale mais dans la direction vers le haut. Le problème se généralise à n corps en faisant simplement la somme des forces exercées sur chaque corps par les autres corps. Le calcul de la nouvelle accélération, de la nouvelle vélocité puis de la nouvelle position se fait ensuite de la même façon. La figure 1.9 propose une trajectoire à trois corps obtenue en ajoutant à la configuration à deux corps précédente un troisième corps situé entre les deux précédents et un peu plus bas.

1.3 Le chemin à parcourir

L'objet est apparu comme un mécanisme de décomposition, d'abstraction et de recomposition capable de s'attaquer effectivement au passage du problème réel à un calcul virtuel en exprimant la solution. Cette vision de l'objet, essentiellement liée à sa relation au monde réel, est complétée par une autre vision de l'objet, liée à sa relation au calcul informatique, qui le voit comme un moyen pour structurer les programmes. Cette seconde vision part des «particules élémentaires» que sont les données directement manipulables par le processeur et les instructions machines et remontent vers l'objet comme moyen pour structurer ces «particules élémentaires» en abstractions modélisant les entités du monde réel.

Cette double vision n'est pas surprenante : elle est courante en science où l'on peut très bien comprendre une entité en partant du tout et en étudiant ses propriétes, par le raisonnement déductif, ou encore en partant des ses éléments constitutifs et en étudiant leurs interrelations, par le raisonnement inductif. Nous allons aussi présenter la seconde vision de l'objet, ce qui nous amènera à étudier plus précisément le fonctionnement de l'ordinateur. Nous verrons ainsi comment l'objet est à la fois une abstraction de l'ordinateur tout en permettant de construire des abstractions des entités de l'espace de problème réel.

Le chemin qui reste à parcourir est stimulant. D'abord, nous nous intéresserons aux aspects liés à la technique de programmation par objets. Cela passera par l'apprentissage d'une notation, le langage Java. Nous verrons les techniques liées à l'objet en soi, mais aussi les techniques liées à la gestion du flôt de contrôle par des énoncés comme les alternatives et les itérations. Nous verrons également des aspects plus méthodologiques, comme la démarche de décomposition liée à la métaphore objet, mais aussi la décomposition descendante liée au flôt de contrôle. Nous verrons aussi une démarche visant à l'exactitude et à la qualité des programmes : l'utilisation d'assertions et de la programmation contractuelle.

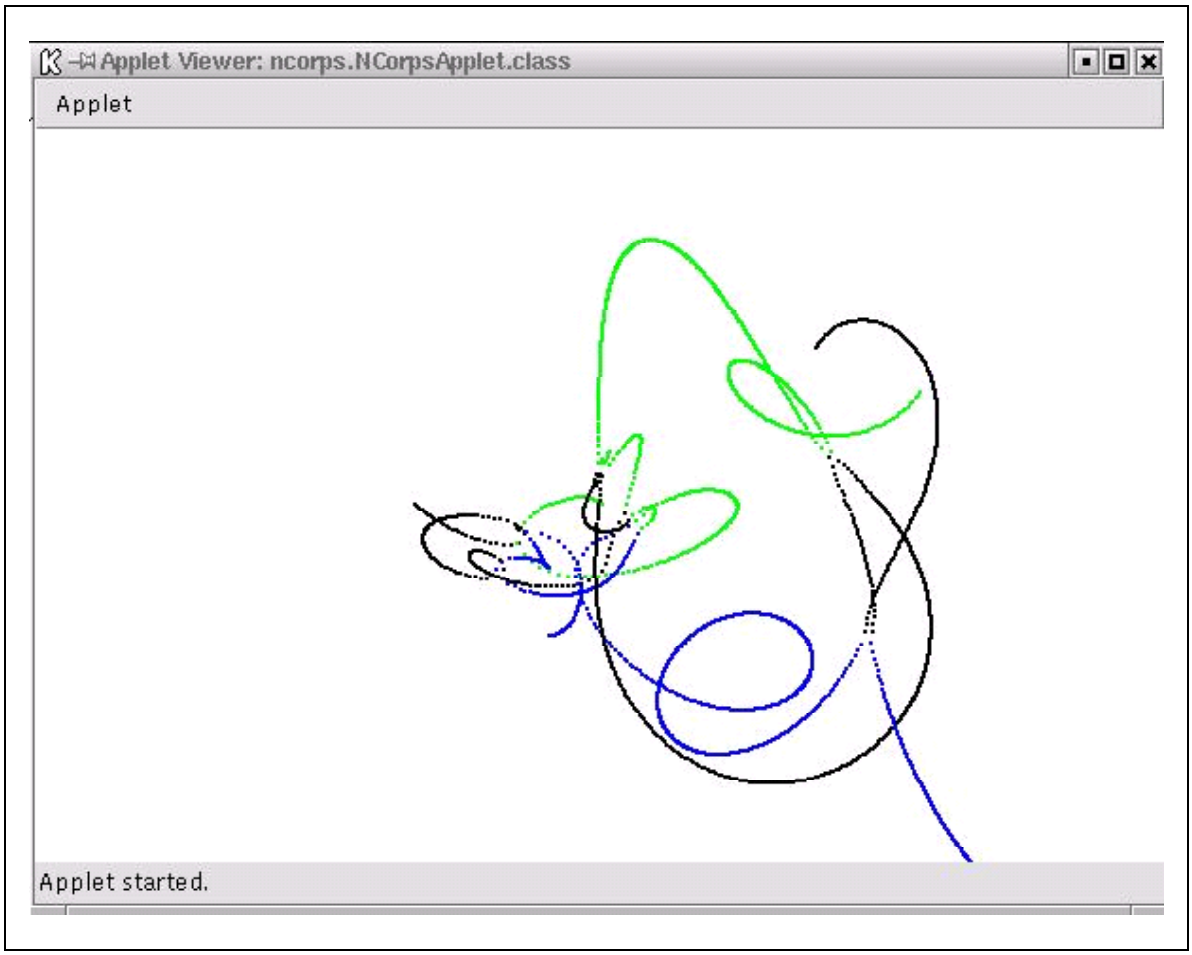


FIG. 1.9 – Une trajectoire à trois corps.

1.4 Éléments de savoir-faire : premiers pas en Java

1.4.1 Utilisation de Java

Java est le langage de programmation utilisé pour exprimer les concepts introduits dans ce document. En tant que langage, il s'agit d'abord et avant tout d'une notation permettant d'écrire des programmes. Mais ces programmes ne sont pas directement exécutables par un ordinateur, qui lui ne sait exécuter que des instructions machine. Pour exécuter un programme dans un langage comme Java, il faut d'abord le traduire en langage machine. En Java, cette traduction se fait en deux étapes :

1. le programme est traduit en un jeu d'instruction intermédiaire, plus souvent appelé *code octal* Java, puis
2. le programme en *code octal* Java est exécuté par un interpréteur, plus souvent appelé *machine virtuelle* (*Java Virtual Machine, JVM*).

La traduction en code octal est réalisée par un outil que l'on appelle un *compilateur*. Le compilateur et la machine virtuelle Java sont des programmes préalablement compilés et prêts à l'exécution sur une machine physique donnée. L'entreprise Sun^{TM} propose de tels

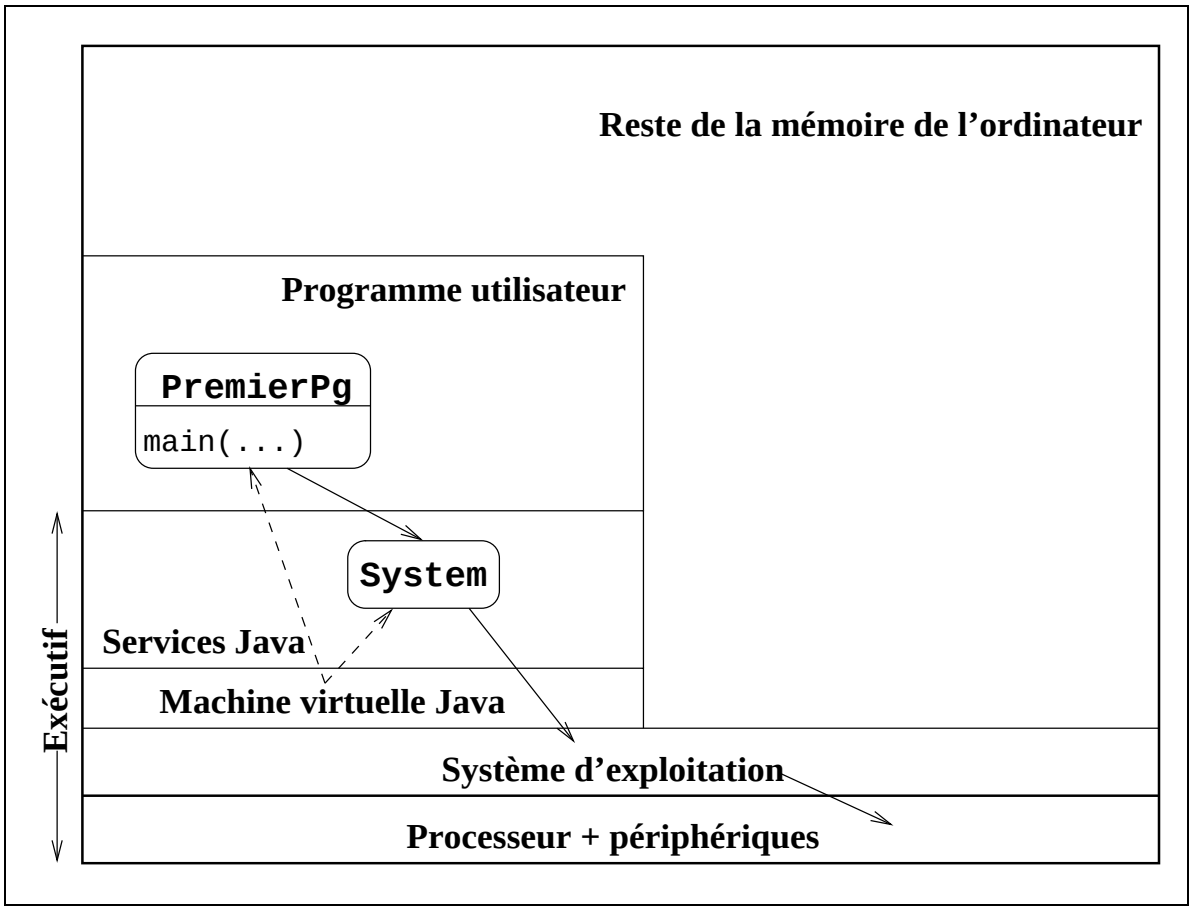


FIG. 1.10 – Organisation sommaire de la mémoire lors de l'exécution d'un programme Java.

programmes dans un environnement de développement standard appelé le Java SDK. La version courante de cet environnement au moment d'écrire ces lignes est la 1.4.1. Dans cet environnement, on trouve plusieurs programmes dont les programmes `javac`, le compilateur, et `java`, la machine virtuelle. On y trouve également des bibliothèques de programmes Java fournis de manière standard, appelée les *Java Foundation Classes, JFC*.

Pourquoi Java utilise-t-il une architecture à deux niveaux, avec un comilateur vers du code octal exécuté par une machine virtuelle plutôt qu'une compilation vers du code machine. Une réponse complète nécessite des connaissances informatiques relativement approfondies. En résumé, ce choix permet d'exécuter des programmes Java plus facilement sur un grand nombre de machines (c'est ce que les informaticiens appellent la *portabilité*) et il permet également une plus grande sécurité dans l'exécution des programmes.

Les services rendus par la machine virtuelle et les bibliothèques sont nombreux :
- exécution du code intermédiaire,
- gestion des entités formant un programme (chargement des classes, sécurité, ...),
- gestion de la mémoire de l'ordinateur,
- communication avec l'extérieur (gestion de l'écran, du clavier, etc.),
- beaucoup de bibliothèques prédéfinies (collections, graphiques, interfaces, ...),
- etc.

L'ensemble de ces services forment ce que l'on appelle le *système exécutif* de Java (ou, en anglais, le *run-time system*). La figure 1.10 présente sommairement l'organisation de la mémoire d'un ordinateur exécutant un programme Java. Le système d'exploitation (Linux ou Micorsoft Windows^{TM}) fournit un certain nombre de services, comme l'accès aux périphériques et la gestion des ressources de l'ordinateur. En lançant la machine virtuelle sur un programme utlisateur donné (ici `PremierPg`), le système exécutif de java rajoute une nouvelle couche de service directement accessibles aux programmes Java. Cet exécutif lance le programme utilisateur via un protocole particulier sur lequel nous reviendrons un peu plus loin. le programme utilisateur peut accéder aux services du système d'exploitation (et donc accéder aux périphériques) via des objets fournis par l'exécutif Java, et en particulier ceux fournis par la classe `System`. Cette classe possède des objets dans ses variables `out` et `in` pour écrire ou lire des châines de caractères à l'écran.

1.4.2 Écriture, compilation et exécution d'un programme

Pour développer un premier programme en Java, il faut : écrire le source dans un fichier '`.java`' (utilisation d'un éditeur de texte), compiler le programme source en une version exécutable , puis lancer le programme avec l'interpréteur de code octal Java. Écrire le source d'un programme Java consiste en fait à définir le code d'une classe. Le patron syntaxique Java pour une classe est le suivant :

```
<qualificatifs> class <NomDeClasse>
{
   <Corps de la classe>
}
```

où :
- `<NomDeClasse>` : débute par une majuscule
- `<qualificatifs>` : propriétés de la classe, dont par exemple la propriété d'être publique (qualificatif `public`)
- `<Corps de la classe>` : deux grandes parties
  1. les déclarations de variables suivies
  2. des déclarations de méthodes

Une classe comme `PremierPg` est appelée *classe racine* car elle définit la méthode `main`. Cette méthode joue un rôle particulier dans le protocole de lancement des programmes Java. Lorsque l'exécutif est lancé par la machine virtuelle Java, on lui indique le nom d'une classe qu'il faut exécuter. Cette classe est la version traduite en code octal, c'est-à-dire un fichier portant le nom de la classe mais se terminant par '`.class`'. Pour que tout se passe bien, il faut que ce fichier se trouve dans un des répertoires indiqués dans la variable d'environnement `CLASSPATH`. Pour lancer l'exécution de la classe chargée, la machine virtuelle lui envoie le message `main`. Ainsi, seules les classes définissant une méthode `main` peuvent être lancées par `java`.

Exemple 2 Dans le cas de la classe `PremierPg`, le code source à écrire à l'aide d'un éditeur de texte (par exemple, `xemacs`) est le suivant :

```
public class            PremierPg
{
```

```
  public static void main (String[] args)
  {
    System.out.println ("Bonjour le monde") ;
  }
}
```

Ce code source doit toujours être placé dans un fichier dont le nom est celui de la classe avec un extension '`.java`'. Pour compiler le programme, on fait :

```
...> javac PremierPg
```

Ceci va créer un fichier `PremierPg.class`, et l'exécution se fait par :

```
...> java PremierPg
```

Dans ce cas, cela imprime à l'écran la phrase suivante :

```
Bonjour le monde
```

□

1.4.3 Environnement de développement JDE

On développe rarement des programmes simplement avec un éditeur de texte et les outils de l'environnement de développement standard Java SDK. On utilise généralement aussi des outils de développement qui augmente la productivité des programmeurs en leurs offrant de nombreux services de support à l'activité de programmation. Le `Java Developpement Environnement ou JDE` est un tel outil, bien que relativement simple et limité. JDE se présente sous la forme d'un mode d'édition de l'éditeur de texte `(x)emacs`. Il propose plusieurs services, dont :

- l'aide à la composition du code en
  - faisant de la synthèse de commentaires de documentation,
  - complémentant les structures syntaxiques standards de java,
  - construisant des menus d'accès aux fichiers et aux éléments dans les fichiers (c'est-à-dire les éléments des classes, comme les variables et les méthodes) ;
- l'aide à la mise au point par un interpréteur de source Java.

L'interpréteur de source Java fourni avec JDE est un outil très rapide pour faire de petits essais sur des objets. Il se lance avec le menu, se présete comme un interprète de commande du système (Unix shell). On peut entrer des instructions Java et les exécuter une par une. On peut aussi faire exécuter des commandes propres à l'interpréteur, dont les principales sont :

- `print` : pour imprimer son argument,
- `show()` : pour basculer le mode d'impression automatique du résultat, et
- `exit()` : pour terminer l'exécution de l'interpréteur.

1.5 Exercices

1.5.1. Définir un objet qui modélise un interrupteur électrique. Quelles sont les informations à prendre en compte pour représenter un interrupteur ? Quelles sont les opérations que l'on peut faire sur un interrupteur ?

1.5.2. Reprendre les questions précédents mais cette fois-ci pour définir un objet qui modélise un réservoir d'un liquide quelconque.

Chapitre 2

Objet = procédures + données

L'objet, en tant qu'entité informatique, est l'aboutissement d'une longue évolution de la programmation impérative. Nous allons voir dans le présent chapitre qu'un objet est composé de procédures et de données. Pour arriver à cette synthèse des deux éléments fondamentaux du calcul par ordinateur, nous allons faire quelques rappels sur les fonctions avant de passer aux procédures, leur contre-partie impérative. Nous allons ensuite étudier l'utilisation des données informatiques pour la représentation de l'information. Le lien intime entretenu entre représentation d'information et traitement de cette même information va servir de justification profonde à l'introduction de l'objet comme entité synthétisant les deux aspects.

2.1 Introduction aux procédures

La procédure est aux langages impératifs ce que la fonction est aux langages fonctionnels. Rappelons donc les grands principes des fonctions avant de passer aux procédures impératives.

2.1.1 Des expressions aux fonctions

En programmation fonctionnelle, l'emphase est mise sur l'écriture d'expressions qui, après évaluation, retourne un certain résultat. Les langages impératifs, et donc Java, possèdent également un sous-ensemble permettant de définir des expressions. L'expression la plus simple est constituée d'un littéral, comme par exemple l'entier '`5`'. Le résultat d'une expression ne contenant qu'un littéral est la valeur de ce littéral, comme ici la valeur entière 5. Une expression peut aussi ne contenir qu'un identificateur de variable, comme par exemple '`x`'. Le résultat d'une telle expression est la valeur contenue dans cette variable au moment où l'expression est évaluée.

On peut ensuite écrire des expressions composées à l'aide d'opérateurs prédéfinis du langage. Par exemple, l'expression

```
x + 5
```

est formée de l'application de la primitive d'addition, dénotée par l'opérateur '`+`', sur le résultat des deux sous-expressions '`x`' et '`5`'. Le résultat d'une expression composée est obtenu en évaluant d'abord ses sous-expressions, puis en appliquant la primitive concernée à ces résultats. Ici, en supposant que la variable `x` contient la valeur 55, l'évaluation de la sous-expression

de gauche donne 55, l'évaluation de celle de droite donne 5, et l'application de la primitive d'addtion à 55 et 5 donne 60.

En résumé, l'évaluation d'une expression formée de littéraux, d'identificateurs de variables et de primitives dénotées par des opérateurs se fait en suivant trois grandes règles :

[Règle des littéraux] Le résultat d'une expression ne contenant qu'un littéral est la valeur de ce littéral.

[Règle des identificateurs de variables] Le résultat d'une expression ne contenant qu'un identificateur de variable est le contenu de cette variable au moment où l'évaluation est faite.

[Règle des primitives] Le résultat d'une expression formée d'une primitive dénotée par un opérateur appliqué à des sous-expressions est obtenu en évaluant d'abord les sous-expressions puis en appliquant la primitive sur le résultat de ces sous-expressions.

Java étant un langage typé, toute expression possède un type qui est celui de son résultat. En première analyse, c'est-à-dire pour des expressions ne contenant que des littéraux et des primitives, on peut déduire ces types de la manière suivante. Tout littéral est typé, et donc le résultat de son évaluation porte ce type. Une variable est déclarée d'un certain type, et donc le résultat d'une expression ne comprtant qu'un identificateur a pour type celui de cette variable. Enfin, les primitives de Java sont typées, c'est-à-dire qu'elles attendent des opérandes de certains types et retournent un résultat d'un cerain type également ; le résultat d'une expression composée est donc du type du résultat déclaré par la primitive appelée.

Notons que les opérateurs de Java sont dits *surchargés*, c'est-à-dire que l'opérateur `+` par exemple désigne plusieurs primitives qui sont choisies par le compilateur en fonction du type des opérandes : si les opérandes sont entières, la pimitive d'addition des entiers est appelée, si les opérandes sont réelles, c'est plutôt la primitive d'addition des réels qui est appelées, et enfin si les opérandes sont des chaînes de caractères, c'est la primitive de concaténation des chaînes qui est appelée.

Le compilateur Java est chargé de veiller à la concordance des types dans les programmes. Ainsi, la règle des primitives ne s'applique que si par ailleurs les types concordent. Si les sous-expressions retournent des résultats dont les types ne concordent pas avec les types attendus des opérandes de la primitive, l'erreur de type sera généralement signalée lors de la compilation du programme, ou encore une conversion de type sera réalisée pour rendre les types concordants.

De plus, une expression contenant plusieurs primitives peut devenir ambigüe en ce qui concerne l'ordre dans lequel les opérations doivent être effectuées. Par exemple, l'expression `5 - 4 - 7` peut donner 8 si elle est interprétée comme `5 - (4 - 7)` ou encore -6 si elle est interprétée comme `(5 - 4) - 7`. De même, l'expression `5 + 4 * 7` donnera 33 si elle est interprétée comme `5 + (4 * 7)` mais plutôt 63 si elle est interprétée comme `(5 + 4) * 7`. Outre le fait qu'il est toujours possible d'ajouter des parenthèses pour forcer un certain ordre d'évaluation, les langages comme Java définissent généralement des règles d'association et de précédence qui déterminent de manière non-ambigûe l'ordre d'évaluation en l'absence de parenthèses. Les types primitifs des littéraux de Java de même que la liste complète des opérateurs de Java ainsi que leurs règles d'association et de précédence sont données à la section 2.4.

Vers les fonctions

Lorsqu'une expression revient régulièrement dans un programme, il est légitime d'en éviter la répétition fastidieuse en «l'empaquetant» dans une fonction. Par exemple, le calcul des racines d'un polynôme de degré 2 utilise la formule bien connue :

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

On en tire deux expressions Java permettant de calculer les deux racines sur des valeurs données, c'est-à-dire `a` = 1.0, `b` = 5.0 et `c` = 4.0[1] :

```
(-5.0 + Math.sqrt(5.0 * 5.0 - 4.0 * 1.0 * 4.0)) / (2.0 * 1.0)
(-5.0 - Math.sqrt(5.0 * 5.0 - 4.0 * 1.0 * 4.0)) / (2.0 * 1.0)
```

La première expression s'évalue à -1.0, alors que la seconde s'évalue à -4.0.

De l'expression, on introduit la fonction. Il s'agit dans un premier temps de paramétrer l'expression selon ce qui varie entre les différents endroits où on veut l'utiliser. Dans les expression précédentes, il est clair que ce sont les valeurs de `a` ,`b` et `c` qui vont varier entre chaque calcul de racine. En remplaçant les valeurs spécifiques par des variables devenant des paramètres du calcul, il devient possible d'appliquer ce calcul à différentes valeurs. Il s'agit ensuite de nommer l'expression pour pouvoir y référer plusieurs fois sans avoir à tout répéter. Par exemple, on peut nommer la première fonction `racine1` et la seconde `racine2`.

En Java, les fonctions sont définies comme des méthodes qui retournent un certain résultat. Le patron syntaxique (simplifié, pour plus de détails voir §2.4) pour la définition d'une méthode jouant le rôle de fonction est :

```
<type du résultat> <nom>(<liste de paramètres formels>)
{
  <corps>
  return <expression> ;
}
```

Le type du résultat est soit le nom d'un type prédéfini de Java, comme `double`, ou encore le nom d'une classe si le résultat de la méthode est un objet. Le nom est un identificateur choisi par le programmeur qui est constitué d'une séquence de lettres et de chiffres débutant nécessairement par une lettre. La liste des paramètres formels et une séquence de paires `<type> <identificateur>` séparées par des virgules. Enfin le corps de la méthode est constitué d'une séquence de déclarations de variables locales suivie d'une séquence d'énoncés (tous deux pouvant être vide dans le cas des méthodes les plus simples). Une méthode jouant le rôle d'une fonction doit nécessairement retourner un résultat à son appelant. L'énoncé return à la fin de la méthode est donc alors obligatoire ; il retourne à l'appelant le résultat obtenu par l'évaluation de l'expression apparaissant à sa droite.

En suivant ce patron syntaxique, on peut définir les deux méthodes suivantes :

```
double racine1(double a, double b, double c)
{
```

[1] Ces expressions font apparaître la référence à la fonctione prédéfinie `Math.sqrt`. Considérons-la momentanément comme une simple primitive.

```
  return (-b + Math.sqrt (b * b - 4.0 * a * c)) / (2.0 * a) ;
}

double racine2(double a, double b, double c)
{
  return (-b - Math.sqrt (b * b - 4.0 * a * c)) / (2.0 * a) ;
}
```

Faire apparaître ces trois variables comme paramètres de la fonction permet donc de reprendre le calcul avec des valeurs différentes sans répéter l'expression. En effet, le pendant de la définition de méthode est l'appel de méthode qui permet de les exécuter. Par exemple, si on veut calculer la première racine pour des valeurs de `a`, `b` et `c` qui sont 3.0, 7.0, 2.0 respectivement, on peut utiliser l'expression dans laquelle les variables `a`, `b` et `c` ont été substituées par leur valeur respective :

```
(-7.0 + Math.sqrt (7.0 * 7.0 - 4.0 * 3.0 * 2.0)) / (2.0 * 3.0)
```

ce qui s'évaluera à -0.3333333. Mais il est plus simple d'utiliser l'appel de méthode qui, en Java, se présente sous la forme suivante :

```
racine1(3.0, 7.0, 2.0)
```

On retrouve le nom de la méthode, suivi des valeurs à substituer à chacun des paramètres dans l'ordre où ceux-ci apparaissent dans la définition de la méthode, séparées par des virgules. Si les paramètres de définition de la méthode sont appelés *paramètres formels*, les paramètres passés lors de l'appel de la méthode sont appelés *paramètres réels*.

L'appel de méthode est réalisé en quatre grandes étapes : évaluation des paramètres réels, transmission des paramètres, et exécution du corps de la méthode, retour du résultat à l'appelant. La première étape est nécessaire pour appeler une méthode avec des paramètres réels qui restent encore à calculer, comme dans l'appel `racine1(x + 10, x - y, 2 * y)`. Chacune des expressions en position de paramètre réel est d'abord évaluée avant l'exécution de la méthode comme telle. L'exécution du corps se fait, puis l'exécution de l'énoncé `return` provoque le retour du résultat au contexte appelant.

L'appel de méthode retournant un résultat fait partie intégrante de ce qu'il est possible d'écrire dans toute expression Java. Cela ajoute donc une nouvelle règle d'évaluation aux trois règles vues précédemment :

[Règle de l'appel de méthode] Le résultat d'une expression ne contenant qu'un appel de méthode est obtenu en évaluant d'abord chacune des expressions apparaissant en position de paramètre réel, puis en exécutant la méthode sur ces paramètres réels. Le résultat retourné par l'énoncé `return` de la méthode devient le résultat de l'appel de méthode, et donc de l'expression elle-même.

L'appel de la méthode en tant que tel suppose que les paramètres réels soient transposées au sein de la fonction de manière à permettre l'évaluation du corps de la méthode et de son énoncé `return`. Cette transmission des paramètres s'assimile à une substitution textuelle des paramètres réels aux paramètres formels dans le corps de la méthode. Cela dit, les choses sont rarement aussi simples, comme nous allons le voir par la suite.

Exemple 3 En guise d'exemple supplémentaire, considérons la sous-expression calculant le discriminant, c'est-à-dire $b^2 - 4ac$. Étant commune au deux méthodes `racine1` et `racine2`, cette sous-expression est une bonne candidate pour former une nouvelle fonction. Ses paramètres sont les valeurs de `a`, `b` et `c`. Cela nous donne :

```
double discriminant(double a, double b, double c)
{
  return b * b - 4 * a * c ;
}

double racine1(double a, double b, double c)
{
  return (-b + Math.sqrt(discriminant(a, b, c))) / (2.0 * a) ;
}

double racine2(double a, double b, double c)
{
  return (-b - Math.sqrt(discriminant(a, b, c))) / (2.0 * a) ;
}
```

Considérons maintenant l'évaluation pas à pas de l'expression `racine1(3.0, 7.0, 2.0)` :

Expression évaluée	Règle appliquée ou action
`racine1(3.0, 7.0, 2.0)`	appel de méthode
`(-7.0 + Math.sqrt(discriminant(3.0, 7.0, 2.0)))`	primitives sur '/'
`/ (2.0 * 3.0)`	
`(-7.0 + Math.sqrt(discriminant(3.0, 7.0, 2.0)))`	primitives sur '+'
`-7.0` ⇒ -7.0	littéraux
`Math.sqrt(discriminant(3.0, 7.0, 2.0))`	appel de méthode
`discriminant(3.0, 7.0, 2.0)`	appel de méthode
`3.0` ⇒ 3.0	littéraux
`7.0` ⇒ 7.0	littéraux
`2.0` ⇒ 2.0	littéraux
`7.0 * 7.0 - 4 * 3.0 * 2.0`	littéraux et primitives
...	
⇒ 25.0	
`return 25.0`	retour au contexte appelant
`Math.sqrt(25.0)`	méthode prédéfinie
...	
⇒ 5.0	retour au contexte appelant
`-7.0 + 5.0` ⇒ -2.0	addition
`(2.0 * 3.0)`	primitives sur '*'
`2.0` ⇒ 2.0	littéraux
`3.0` ⇒ 3.0	littéraux
⇒ 6.0	multiplication
⇒ 0.3333	division

□

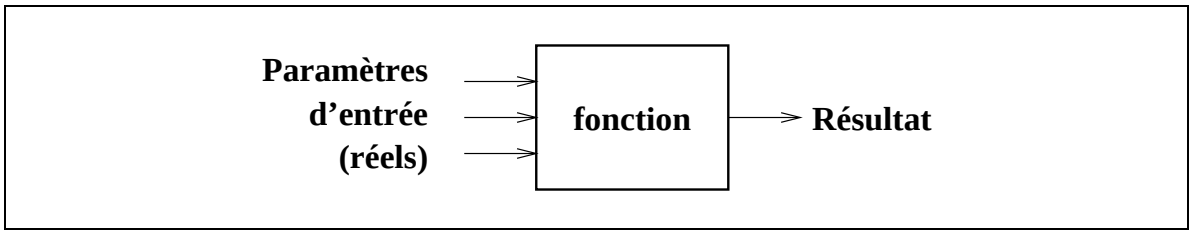


FIG. 2.1 – Abstraction fonctionnelle

Fonction comme abstraction

En quoi une fonction est-elle une abstraction ? Le mot abstraction ici doit être pris dans un sens très pratique. La fonction abstrait un certain calcul en le cachant derrière un nom et un protocole pour passer des paramètres à ce calcul, comme nous venons de le voir. À partir du moment où on utilise une fonction plutôt que l'expression elle-même, l'appelant devient en quelque sorte indifférent à l'expression utilisée pour faire le calcul. Tout ce qui l'intéresse, c'est comment faire exécuter ce calcul et comment en récupérer le résultat. Par exemple, que se passe-t-il si on modifie la méthode `racine1` de la façon suivante :

```
double racine1(double a, double b, double c)
{
  return (-b + Math.sqrt (b * b - (a + a + a + a) * c)) / (a + a) ;
}
```

Nous connaissances arithmétiques de base nous indiquent que le résultat de cette fonction sera le même que celui de la fonction précédente, peu importe les paramètres réels utilisés. Le résultat de la fonction étant insensible à la modification, et les paramètres formels les mêmes, l'appelant de la fonction n'y verra que du feu.

La fonction apparaît alors comme une boîte noire, dans laquelle on fait entrer des paramètres réels, et de laquelle on récupère un résultat (fig. 2.1). Le grand avantage de l'abstraction est donc l'indépendance entre l'intérieur de la fonction et les appelants. Le second intérêt est l'organisation que les fonctions permettent de donner aux programmes. Une fonction permet d'abstraire un certain calcul, qui lui-même peut utiliser d'autres fonctions. On peut ainsi créer différents niveaux d'abstraction, en créant des fonctions dont le calcul, tout en étant très complexe, peut être compris sans avoir à en connaître les détails de réalisation.

Enfin, en Java, comme dans la plupart des langages de programmation, la méthode forme une barrière d'encapsulation qui la rend opaque à ses appelants. Les noms des paramètres formels, les variables locales de même donc que le corps, ne sont pas accessibles de l'extérieur.

2.1.2 De la variable mathématique à la variable impérative

Contrairement à la variable mathématique, qui représente une inconnue, la variable impérative est un nom donné à un mot mémoire par un programme. La variable impérative remplace donc l'adresse du mot mémoire correspondant (fig. 2.2). En mathématique, une formule fait intervenir des inconnues. Elle nous permet de calculer un résultat après avoir remplacé toutes les inconnues par des valeurs. Ce *principe de substitution*, c'est-à-dire que substituer une variable par sa valeur dans une expression ne change pas le résultat de cette expression, implique que jamais une inconnue ne va changer de valeur en cours de calcul.

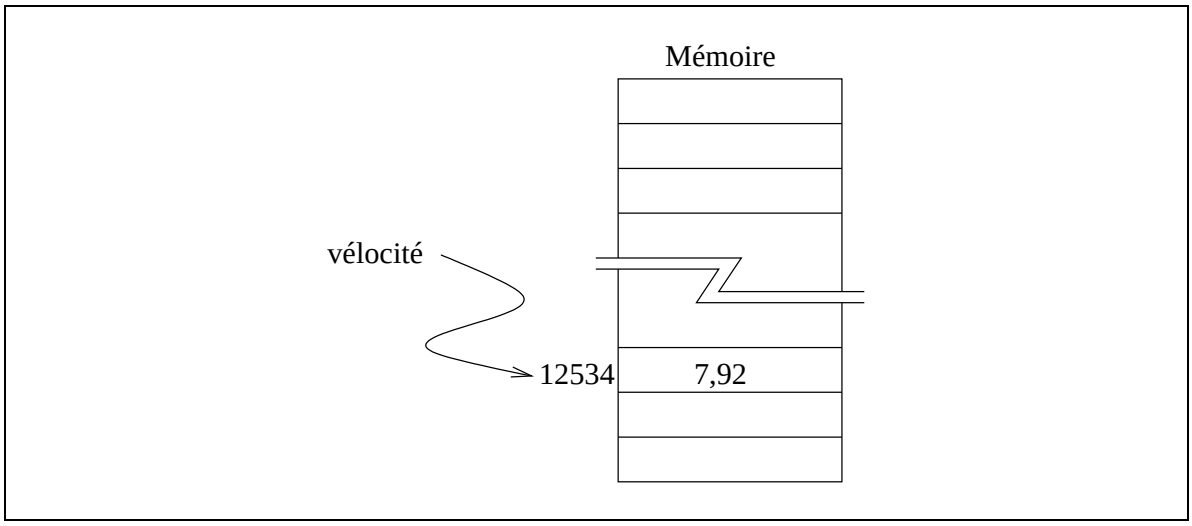


FIG. 2.2 – La variable impérative

Avec un programme et les variables impératives, les choses sont complètement différentes. Une variable dans un programme désigne un mot mémoire dont le contenu peut changer autant de fois que le programme écrit une valeur dans ce mot durant son exécution. Le principe de substitution fondamental en mathématique n'est plus valide en programmation impérative. En effet, entre deux occurrences de la même variable dans le programme, la valeur de celle-ci peut fort bien avoir changée. Il n'y a donc plus de valeur unique que l'on puisse lui substituer.

C'est pour cela qu'on peut raisonner sur une formule mathématique selon le principe de substitution d'une inconnue par sa valeur (elle devient alors connue). Dans un programme, on ne peut raisonner ainsi et on doit adopter le principe des traces et de séquence d'états mémoire pour en comprendre le résultat.

On dit que la variable impérative permet simplement de s'abstraire des adresses mémoire. L'un des avantages est de rendre les programmes plus lisibles. Mais cette abstraction permet également à l'ordinateur une bien plus grande flexibilité. Par exemple, que faire si l'adresse désignée par une portion de programme pour contenir une donnée est déjà utilisée par une autre portion de programme ? En utilisant un nom plutôt qu'une adresse, on permet à l'ordinateur de choisir des mots mémoire libres pour contenir les données d'une portion de programme, peu importe l'utilisation faite par les autres portions de programme des mots de la mémoire.

Description des états mémoire

La notion d'état mémoire est fondamentale. Il est donc important de bien la comprendre et de bien comprendre sa relation à la notion d'exécution d'instructions sur un ordinateur de type von Neumann. Chaque instruction d'une machine est spécifiée en terme des modifications qu'elle fait sur l'état de la machine. Comprendre le résultat d'une séquence d'instructions demande donc de comprendre comment chacune des instructions exécutées a modifié l'état de la mémoire et comment ces modifications se sont enchaînées de façon à produire une séquence d'états mémoire depuis un état initial jusqu'à un état final.

Pour manipuler la notion d'état mémoire, nous allons nous donner une façon de les décrire. Déjà, la trace d'un programme machine nous permet de suivre la séquence des états mémoire produits par l'exécution des instructions du programme. On voit déjà dans cet exemple qu'un état mémoire peut être assimilé à une correspondance un à un entre adresses et contenus de

mots mémoires. La manipulation de la mémoire à partir des adresses, comme nous l'avons vu, est aride. Si nous passons à l'utilisation de variables pour s'abstraire des adresses, il devient possible de décrire la mémoire comme l'association d'une variable à une valeur.

Comment décrire cette association ? D'un part, il est évident que nous pouvons substituer les noms de variables aux adresses dans les traces. Mais cette présentation est encore trop simplistes pour permettre de raisonner sur les programmes et sur leur exactitude. Nous allons utiliser, pour des raisons qui deviendront plus claires par la suite, une formulation logique. En logique, établir qu'une variable est associée à une valeur donnée, c'est équivalent à dire que la variable est égale à cette valeur. Dans l'exemple du discriminant, dire qu'un état mémoire est caractérisé par le fait que les valeurs 1, 5 et 2 sont associées respectivement aux variables `a`, `b` et `c` peut ainsi s'exprimer par la formule logique :

$$\mathtt{a} = 1 \wedge \mathtt{b} = 5 \wedge \mathtt{c} = 2$$

En logique, l'évaluation d'une telle proposition se fait dans un état, un état associant à chaque variable une valeur ce qui permet de faire chaque comparaison, et si les trois comparaisons d'égalité sont vrais, la proposition sera vraie. Dans ce sens, on peut dire que la proposition défini un état particulier, l'état dans lequel elle est vraie.

En complexifiant un peu les propositions, on peut arriver à décrire non pas un seul état, mais plutôt des ensembles d'états. Déjà, si l'état contient trois variables `a`, `b` et `c`, une proposition qui n'en mentionne que deux :

$$\mathtt{a} = 1 \wedge \mathtt{b} = 5$$

peut être interprétée comme désignant tous les états dont les valeurs de `a` et `b` sont respectivement 1 et 5, mais pour lesquels la valeur de `c` est indifférente. De plus, en utilisant la disjonction, on peut désigner explicitement certains états. La proposition suivante désigne ainsi les états où `a`, `b` et `c` valent respectivement 1, 5, 2 ou 2, 3, 6 :

$$(\mathtt{a} = 1 \wedge \mathtt{b} = 5 \wedge \mathtt{c} = 2) \vee (\mathtt{a} = 2 \wedge \mathtt{b} = 3 \wedge \mathtt{c} = 6)$$

Elle désigne donc un ensemble d'états qui contient deux éléments. Mieux encore, en utilisation des relations, il devient possible de décrire des états possédants certaines propriétés précises. Par exemple, si un état contient deux variables `x` et `y` et que l'on veut préciser que cet état représente un point sur le bissectrice du premier quadrant du plan cartésien, on peut l'exprimer par :

$$\mathtt{x} \geq 0 \wedge \mathtt{y} \geq 0 \wedge \mathtt{x} = \mathtt{y}$$

Cette proposition désigne tous les états tels que `x` et `y` sont positifs et égaux, ce qui désigne bien le lieu géométrique de la bissectrice du premier quadrant.

On voit donc que le langage de la logique permet non seulement de décrire un état mémoire, mais aussi des ensembles d'états mémoire possédant des propriétes précises. Nous allons utiliser dans le reste de ce module cette possibilité pour définir les ensembles d'états dans lesquels des instructions peuvent s'exécuter sans faire d'erreur. Nous y reviendrons.

Premier énoncé impératif : l'affectation

Tout comme le passage de l'adresse numérique à la variable impérative en est un jalon fondamental, la reconstruction des instructions de la machine sous la forme d'ordres impératifs

plus proche de l'expression humaine est au cœur de la quête pour des langages de programmation de haut niveau. Étant elle-même cruciale au modèle de von Neumann, la modification de l'état de la mémoire arrive en tête de ce processus.

Les instructions de la machine de von Neumann permettent la modification du contenu des mots mémoires. La variable impérative étant l'abstraction de la mémoire, l'énoncé d'affectation devient l'abstraction des instructions d'écriture dans les mots mémoire. L'énoncé d'affectation est construit à partir de trois éléments :

1. le nom de la variable cible de l'affectation,
2. l'opérateur d'affectation qui est la marque extérieure, syntaxique, de l'énoncé, et
3. une expression, dont le résultat deviendra le nouveau contenu de la variable cible.

Selon le langage de programmation choisi, l'apparence externe de cet énoncé peut évidemment varier. L'opérateur d'affectation de Java est '='. En utilisant la notation Java, l'affectation de la valeur 5 à la variable `x` s'écrit[2] :

```
x = 5
```

L'effet obtenu de l'exécution de cet énoncé est de prendre un état mémoire contenant une variable `x`, tel qu'il se présente avant d'exécuter l'affectation, et de le modifier de telle façon à ce que dans l'état mémoire après l'exécution de l'affectation, le contenu de la variable `x` soit 5. Dans la sémantique, la signification, d'un énoncé d'affectation, il y a un *avant* et un *après* ; entre les deux un changement irrémédiable s'est produit.

L'expression en partie droite de l'opérateur d'affectation se comprend dans le même sens que l'expression dans un langage de programmation fonctionnelle, comme CaML[3]. Il s'agit d'une expression, par exemple artihmétique, à évaluer dans l'état précédant l'affectation, et qui rendra un résultat. Ce résultat sera placé dans la variable cible de l'affectation.

La principale différence entre l'expression dans un langage impératif par rapport au langage fonctionnel vient de ce que les variables intervenant dans l'expression sont par définition des variables impératives. Cette différence est marquée par la possibilité de voir changer la valeur d'une variable en cours d'évaluation de cette expression. Hérésie encore une fois pour le mathématicien, ceci se comprend par la nature impérative de ces langages, bien sûr. Nous verrons plus loin comment, au cours d'une expression, on peut être amené à modifier l'état mémoire. Mais, il suffit de dire pour l'instant que la vision présentée ci-haut d'un seule modification de l'état mémoire est idéalisée.

Pour tout mathématicien, l'énoncé d'affectation le plus surprenant, et à la limite le plus incompréhensible est :

```
x = x + 1
```

Comment, en effet, `x` pourrait-il être *égal* à `x + 1` ? On aura compris que cette interrogation oublie que l'opérateur '=' n'est pas justement l'opérateur de comparaison d"égalité mais bien un opérateur d'affectation. La signification opérationnelle de cet énoncé d'affectation consiste

[2]On comprendra que le principal défaut d'utiliser l'opérateur '=' comme opérateur d'affectation est de le confondre avec l'opérateur de comparaison d'égalité. C'est pour cette raison que Pascal, comme d'autres langages de programmation impérative, a choisi d'utiliser un opérateur manifestement différent du comparateur d'égalité utilisé en mathématique (et en logique, comme nous l'avons vu). Notons au passage qu'en Java, l'opérateur de comparaison d'égalité s'écrit '=='.

[3]Bien que dans le module de programmation fonctionnelle en première année de DEUG MIAS de l'UBS, CaML soit utilisé comme un langage fonctionnel pur, il est bel et bien en réalité un langage impératif. Cela dit, tout le monde s'entend pour reconnaître que CaML en tant que langage est dominé par sa facette fonctionnelle.

à évaluer l'expression `x + 1` dans l'état mémoire précédant l'énoncé d'affectation, puis de produire un nouvel état mémoire, à partir de l'état mémoire précédent, par modification la valeur de la variable cible `x`. On pourrait définir ce processus de manière informelle en décrivant l'effet de l'instruction d'affectation sur la mémoire par une transformation de l'état mémoire avant en un état mémoire après. En supposant que la valeur initiale de `x` est 10, on a :

{La variable `x` *a pour valeur 10}*
```
x = x + 1
```
{La variable `x` *a pour valeur 11}*

Si nous utilisons notre approche logique de description des états mémoire, nous pouvons représenter l'explication précédente en annotant l'énoncé d'affectation de la description de précédant l'énoncé, puis la description de l'état suivant l'énoncé.

$\{\mathtt{x} = 10\}$

```
x = x + 1
```

$\{\mathtt{x} = 11\}$

Ce type d'annotations sont appelées *assertions*. Il faut lire ce qui précède de la façon suivante. Si l'état observé au moment d'exécuter l'énoncé d'affectation est tel que la première assertion $\mathtt{x} = 10$ est vraie, alors après l'exécution de l'énoncé d'affectation, l'état mémoire sera tel que la seconde assertion $\mathtt{x} = 11$ sera vraie. De cette façon, il est possible d'expliquer l'effet des énoncés.

Il est possible d'obtenir systématiquement la première assertion de la seconde en introduisant l'effet de l'énoncé d'affectation et en notant que la seconde assertion s'intéresse à la variable après l'affectation alors que la première s'adresse à la variable avant l'affectation. Considérons ces deux moments comme deux variables distinctes notée $\mathtt{x}_{\text{avant}}$ et $\mathtt{x}_{\text{après}}$. L'exemple devient :

$\{\mathtt{x}_{\text{avant}} = 10\}$
$\mathtt{x}_{\text{après}}$ `=` $\mathtt{x}_{\text{avant}}$ `+ 1`
$\{\mathtt{x}_{\text{après}} = 11\}$

Si on remplace $\mathtt{x}_{\text{après}}$ dans la seconde assertion par la partie droite de l'affectation, on obtient $\{\mathtt{x}_{\text{avant}} + 1 = 11\}$, d'où $\{\mathtt{x}_{\text{avant}} = 10\}$.

En utilisant les assertions, il est en fait possible de définir formellement la signification d'un énoncé en donnant une règle de transformation de l'antécédent vers le conséquent, c'est-à-dire de l'assertion avant vers l'assertion après (ou vice versa). Cette approche pour raisonner sur les programmes impératifs et en comprendre le fonctionnement est due à Hoare. Soit A une assertion, notons A^e_x l'assertion A dans laquelle toutes les occurences libres[4] de x ont été remplacées par e. La règle pour l'affectation devient alors :

$\{A^e_x\}$ `x = e` $\{A\}$ (ou réciproquement $\{A\}$ `x = e` $\{A^x_e\}$)

L'impression que la règle fonctionne «en sens inverse» vient du fait que l'on substitue dans A la valeur de la variable après l'affectation par l'expression qui donne la valeur affectée. En appliquant cette règle au cas précédent, on obtient bien la même chose que précédemment. L'antécédent a' est obtenu par substitution de `x + 1` pour `x` dans le conséquent c' ; l'antécédent a n'est qu'une simplification de a' en passant le 1 de l'autre côté de l'égalité puis en effectuant la simplification $10 + 1 - 1 = 10 + 0 = 10$:

[4]Par libre ici, on entend que l'on ne substitue que les occurrences de `x` référent à la variable affectée dans l'instruction, car d'autres `x` peuvent apparaître ailleurs dans le programme. De plus, l'expression `e` ne doit pas contenir elle-même de modifications d'états par affectation.

$\{a : \texttt{x} = 10\}$
$\{a' : \texttt{x + 1} = 10 + 1\}$

```
x = x + 1
```

$\{c' : \texttt{x} = 10 + 1\}$
$\{c : \texttt{x} = 11\}$

Il est possible de généraliser nos expressions de description d'états mémoire. Dans le cas précédent nous aimerions pouvoir dire que pour n'importe quelle valeur de X entière, on observe :

$\{\texttt{x} = X\}$

```
x = x + 1
```

$\{\texttt{x} = X + 1\}$

De la même façon que l'utilisation de relations dans les assertions, l'utilisation des quantificateurs de la logique $\exists$ (il existe) et $\forall$ (pour tout) permet de définir un ensemble d'états mémoire. Dans le cas précédent, nous sommes en mesure de dire que pour tout X, à partir d'un état mémoire où la variable `x` a pour valeur X, l'état mémoire produit par l'énoncé d'affectation en sera un où la variable `x` aura pour valeur $X + 1$. On notera cela de la façon suivante :

$$(\forall X \in I\!N)\{\texttt{x} = X\} \ \texttt{x = x + 1} \ \{\texttt{x} = X + 1\}$$

Il faut bien comprendre ici que nous utilisons deux niveaux de langage différents pour en quelque sorte faire une vérification croisée du programme lui-même et de l'intention du programme. L'opération d'addition de l'énoncé d'affectation est celle du langage de programmation et a un effet sur le programme. L'opération d'addition de la seconde assertion est l'opération mathématique bien connue sur $I\!N$. La seconde exprime mathématiquement ce que doit faire la première.

La séquence d'énoncés

Dans la droite ligne de l'évolution à la suite de l'énoncé d'affectation vient la composition séquentielle d'énoncés. Par sa nature même, et comme nous l'avons vu dans les exemples de programmes machine, la programmation invite à accumuler les ordres impératifs les uns à la suite des autres pour les exécuter ensuite en séquence. Dans la plupart des langages de programmation impérative, la séquence d'énoncés est obtenue simplement en écrivant les énoncés les uns après les autres, le plus souvent en les séparant syntaxiquement par le symbole de ponctuation ' ;'. Les points d'observation de la séquence sont tout simplement au début, à la fin et ente chacun des énoncés composant la s'équence. En annotant comme précédemment, cela donne :

$\{\texttt{x} = X\}$

```
x = x + 1 ;
```

$\{\texttt{x} = X + 1\}$ $\quad (\Rightarrow \{\texttt{x} = X + 1 \wedge \texttt{2 * x} = 2(X + 1)\})$

```
y = 2 * x ;
```

$\{\texttt{x} = X + 1 \wedge \texttt{y} = 2(X + 1)\}$ $\quad (\Rightarrow \{\texttt{x} = X + 1 \wedge \texttt{y + 1} = 2(X + 1) + 1\})$

```
y = y + 1
```

$\{\texttt{x} = X + 1 \wedge \texttt{y} = 2(X + 1) + 1\}$

La notion de composition vient de ce que, selon l'approche logique précédente, l'état mémoire produit par l'exécution du n$^{\text{e}}$ énoncé de la séquence devient l'état mémoire dans lequel est exécuté le (n + 1)$^{\text{e}}$ énoncé. Cet enchaînement donne sa puissance au paradigme, mais devient en même temps sa faiblesse dans la mesure où la compréhension d'une séquence d'énoncé demande d'en comprendre les effets, pas toujours si apparents, sur l'état mémoire.

Soient e_1 et e_2 deux énoncés, la règle de transformation pour l'énoncé séquentiel e_1 ; e_2 s'exprime de la manière suivante :

si $\{A\}\ e_1\ \{A'\}$ **et** $\{B\}\ e_2\ \{B'\}$ **et** $\{A'\} \Rightarrow \{B\}$ **alors** $\{A\}\ e_1$; $e_2\ \{B'\}$

De façon intéressante, l'utilisation des assertions permet de comprendre la séquence d'énoncés précédente uniquement par son effet global sur l'état mémoire. Rien ne sert en effet de connaître précisément les énoncés composant la séquence, ni les assertions intermédiaires. Seules comptent pour la compréhension globale de la séquence la première et la dernière assertion. Cette capacité à comprendre une portion de programme pour son effet global est un élément essentiel dans l'apprentissage et l'utilisation de la programmation impérative.

Exemple 4 Programmer est difficile ! Arsac proposait en 1984 la séquence d'énoncés suivante :

```
...
x = x + y ;
y = x - y ;
x = x - y
```

Que fait-elle ?

Il est très difficile de voir à quoi servent ces trois énoncés sans raisonner de façon systématique sur le contenu des variables. Appliquons les annotations à cet exemple en posant comme valeur de `x` et `y` les constantes A et B, respectivement :

$\{\texttt{x} = A \wedge \texttt{y} = B\}$

```
x = x + y ;
```

$\{\texttt{x} = A + B \wedge \texttt{y} = B\}$

```
y = x - y ;
```

$\{\texttt{x} = A + B \wedge \texttt{y} = A\}$

```
x = x - y
```

$\{\texttt{x} = B \wedge \texttt{y} = A\}$

L'analyse des première et dernière assertions montre clairement que la séquence d'énoncés réalise l'échange des valeurs de `x` et `y` ! □

Programmer un ordinateur est difficile parce qu'il faut traduire une intention, le résultat que l'on veut obtenir, en une séquence d'opérations qui vont mener à ce résultat. En général, l'intention ne se retrouve pas explicitement dans les opérations. Il est donc important de rendre les programmes les plus explicites possibles à ce sujet. Comme les opérations elles-mêmes ne sont pas explicites, c'est par l'utilisation judicieuse du nommage (en utilisant des noms donnant une signification aux données et traitements ainsi nommées), et aussi par l'utilisation appropriée de commentaires dans les programmes qu'il est possible de rendre explicite l'intention d'un programme à destination d'un auter programmeur qui chercherait à comprendre ce que fait le programme.

L'exemple précédent illustre aussi un autre aspect important des assertions, plus précisément la possibilité de déduire les assertions successives d'une séquence d'énoncés à partir de l'assertion initiale. Dans le cas de l'exemple d'Arsac, la déduction des assertions est relativement simple ; il suffit d'utiliser l'expression de l'énoncé d'affectation et de remplacer par les valeurs des variables données par l'assertion précédente. Dans le cas de programmes plus complexes, les règles de déduction utilisent des techniques plus sophistiquées. Un langage logique de description des états muni de règles de déduction prenant en compte les énoncés d'un langage de programmation est appelé *logique de programme* ; elle est la base du domaine de la *sémantique axiomatique* des programmes dont l'objectif est de faire des preuves formelles de propriétés sur les programmes (dont l'exactitude).

2.1.3 La procédure

La procédure en programmation impérative est également une abstraction basée sur le nommage et la paramétrisation. Par contre, plutôt que d'abstraire l'expression, on abstrait maintenant l'énoncé. Une procédure se définit donc par un nom, une liste de paramètres formels et un corps. Le corps d'une procédure est constitué d'une séquence d'énoncés. La conséquence immédiate en est que l'appel d'une méthode ne retourne pas normalement de résultat mais agit plutôt en modifiant l'état de la mémoire, comme tout énoncé.

En Java, une méthode qui ne retourne pas de résultat est l'équivalent d'une procédure. Le patron syntaxique permettant de définir une telle méthode est identique à celui que nous avons vu précédemment, à cette différence près que le type du résultat est nécessairement `void`, un mot-clé indiquant qu'il n'y a pas de résultat à cette méthode. N'ayant pas de résultat à retourner, une telle méthode n'utilise pas d'énoncé `return`[5] ; l'exécution de la méthode se termine simplement lorsque le flôt d'exécution parvient à la fin du corps de la méthode.

```
void <nom>(<liste de paramètres formels>)
{
  <corps>
}
```

Considérons une version procédurale du calcul des deux racines du polynôme de degré 2. La méthode suivante prend trois paramètres `a`, `b` et `c`, calcule les deux racines, les rendant

[5] Il n'est pas stricto sensu interdit d'utiliser un énoncé `return` (sans expression de retour cependant) dans une méthode procédurale. Cela dit, sauf cas exceptionnel, ce n'est pas une bonne pratique car cela induit généralement des flôts de contrôle difficile à comprendre.

disponibles au contexte appelant en affectant les résultats à deux variables `r1` et `r2` déclarées comme variable locale à l'objet :

```
public class ExempleRacines
{
  double r1, r2 ;

  void racines(double a, double b, double c)
  {
    r1 = (-b + Math.sqrt(b*b - 4*a*c))/(2*a) ;
    r2 = (-b - Math.sqrt(b*b - 4*a*c))/(2*a) ;
  }
}
```

Tout comme la fonction, la procédure est une abstraction, au sens développé précédemment. Notez que si nous modifions la méthode précédente pour n'effectuer qu'une fois les calculs communs aux deux affectations, on obtient une procédure équivalente :

```
  void racines(double a, double b, double c)
  {
    double t1, t2 ;

    t1 = Math.sqrt(b*b - 4*a*c) ;
    t2 = 2*a ;
    r1 = (-b + t1)/t2 ;
    r2 = (-b - t1)/t2 ;
  }
```

Les appelants ne sont pas touchés par cette modification dans la mesure où le protocole pour passer les valeurs à la procédure et pour récupérer le résultat reste le même. Toute abstraction qu'elle est, la procédure introduit par rapport à la fonction, une notion de résultat indirect. Ici, il s'agit de la modification de la valeur des variables `r1` et `r2`. Comme ces résultats n'apparaissent pas explicitement dans la définition de la procédure, ils sont souvent appelées *effets de bord* de la procédure, car ils supposent un paramètre «fantôme» à la procédure, l'état mémoire, qui est modifié par l'appel de la procédure (fig. 2.3). Ce pseudo-paramètre est toujours implicite pour toute procédure dans les langages impératifs. De la même façon qu'un énoncé d'affectation agit sur l'état mémoire, une procédure agit elle aussi sur cet état mémoire, bien que cet état mémoire n'y apparaisse pas explicitement.

Les effets de bord sont considérés comme une source majeure de difficulté dans la compréhension des programmes, et sont donc à éviter autant que faire se peut. À chaque fois qu'il est possible d'utiliser une méthode fonctionnelle pour réaliser l'abstraction d'un calcul, il faut préférer cette solution au recours à une méthode procédurale avec effet de bord. Il va de soi que notre exemple de `racines` ici est plus imple à écrire en méthode procédurale car il est difficile de retourner deux résultats d'une méthode fonctionnelle. Cependant, la solution consistant à définir deux méthode fonctionnelles pour calculer les deux racines serait vraisemblablement préférée par le programmeur consciencieux.

Il est intéressant de noter ici un effet important de l'abstraction procédurale (tout comme de l'abstraction fonctionnelle). La procédure agit comme une frontière qui masque sa définition

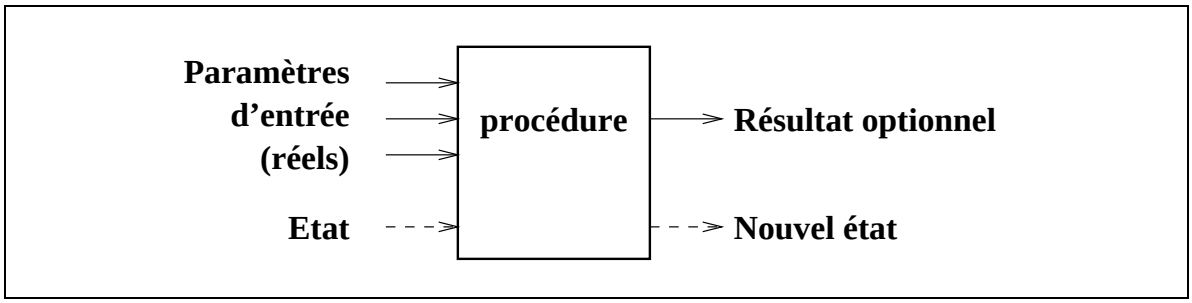


FIG. 2.3 – Abstraction procédurale

interne (corps) à ses utilisateurs (appelants). En particulier, les variables locales à la procédure ne sont ni visibles, ni utilisables de l'extérieur. Cet aspect est extrêmement important, dans la mesure où il donne un puissant outil de structuration de la mémoire. Sur une grande mémoire, si tout énoncé peut éventuellement modifier n'importe quel mot mémoire, ou variable, la compréhension du programme devient un défi insurmontable.

Tout comme nous avions fait remarquer qu'un programme machine devient plus simple à comprendre si on adopte le principe une donnée - un mot mémoire, la frontière autour de la procédure permet de réaliser un autre principe de bon sens qui est la séparation des variables entre les différentes parties d'un programme. Chaque partie d'un programme travaille sur ses propres variables, à l'exclusion de toute autre partie, sauf pour les variables qui servent de protocole de passage d'information d'une partie à l'autre (comme `r1` et `r2` dans notre exemple).

L'observation de ce principe de bon sens permet d'obtenir une propriété très importante des «bons» programmes (c'est-à-dire faciles à comprendre) : *la localité*. La propriété de localité consiste simplement à observer que tout ce qui est nécessaire pour comprendre une partie de programme, une procédure par exemple, est ce qui se trouve dans cette partie de programme (toujours modulo les informations échangées aux frontières). Sans localité, les éléments nécessaires pour comprendre une partie de programme pourraient se retrouver n'importe où dans le programme ce qui, pour des programmes de très grande taille, rend la chose impossible.

Un programme observant le principe de localité peut être découpé en parties relativement indépendantes, autour desquelles il est possible de tracer des frontières par lesquelles seules les informations volontairement passées dans les protocoles d'appels sont visibles. On dit alors que le programme est *modulaire*.

Notons aussi qu'en réalité, dans plusieurs langages de programmation, au lieu de procédures et de fonctions distinctes, il existe un concept unique de fonctions procédurales, c'est-à-dire de «fonctions» qui peuvent ou non retourner un résultat et qui peuvent également modifier l'état de la mémoire. C'est le cas des méthodes de Java qui peuvent modifier l'état de l'objet les exécutant et retourner un résultat à leur appelant.

Méthodes et assertions

Pour expliciter le protocole utilisé pour récupérer les résultats de la méthode `racines`, il est possible d'utiliser des assertions. L'appel à la méthode `racines` peut être annotée des assertions suivantes :

$\{\mathtt{a} = A \wedge \mathtt{b} = B \wedge \mathtt{c} = C \wedge \mathtt{b}^2 - 4\mathtt{ac} \geq 0 \wedge \mathtt{a} \neq 0\}$

```
racines(a, b, c)
```

$\{\mathtt{r1} = \frac{-B+\sqrt{B^2-4AC}}{2A} \wedge \mathtt{r2} = \frac{-B-\sqrt{B^2-4AC}}{2A}\}$

Une telle explicitation a aussi pour effet de rendre clair ici que les valeurs A, B et C utilisée comme paramètre réel de la méthode doivent respecter certaines contraintes, comme ici le fait que le discriminant soit non-négatif et que la valeur A ne soit pas zéro. Bien sûr ces contraintes viennent du fait qu'on va extraire la racine carrée du discriminant et diviser par un mutiple de A.

Il serait intéressant d'accrocher en quelques sortes ces conditions à la méthode. L'utilisateur de la méthode pourrait alors vérifier, sans avoir à regarder le code de la méthode, que les valeurs passées à celle-ci sont conformes à ce qui est attendu, et ainsi être assuré du bon déroulement du calcul et de l'obtention d'un résultat correct à la fin de ce dernier. Pour ce faire, on va associer à la définition de la méthode deux assertions : une *précondition* et une *postcondition*.

La précondition est une assertion portant sur les paramètres formels, et parfois aussi sur les variables visibles depuis cette méthode (dont les variables d'instance de l'objet exécutant la méthode), et qui définit la relation qui doit être vérifiée sur les valeurs des paramètres réels et des variables visibles qui doit être vraie lors de l'appel de la méthode pour que celle-ci s'exécute correctement. La postcondition est une assertion portant sur le résultat et sur les variables visibles et qui définit la relation qui sera vraie sur ce résultat et sur ces variables visibles après l'exécution de la méthode.

La méthode `racines` ainsi annotée se définit comme suit, en utilisant une norme de plus en plus établie pour la définition des pré- et postconditions en Java :

```
public class ExempleRacines
{
  static double r1, r2 ;

  /**
   * @pre     b*b - 4*a*c >= 0.0 && a != 0.0
   * @post    r1 == (-b + Math.sqrt(b*b - 4*a*c))/(2*a) &&
   *          r2 == (-b - Math.sqrt(b*b - 4*a*c))/(2*a)
   */
  public static void      racines(double a, double b, double c)
  {
    r1 = (-b + Math.sqrt(b*b - 4*a*c))/(2*a) ;
    r2 = (-b - Math.sqrt(b*b - 4*a*c))/(2*a) ;
  }
}
```

En Java, les assertions se présentent sous la forme d'expressions booléennes (voir §2.4 pour la panoplie complète des opérateurs logiques et de comparaison de Java). Bien sûr, on peut se demander l'intérêt de répéter dans la postcondition le calcul apparaissant déjà dans le corps de la méthode. De fait, lorsque cela est possible, il est préférable de poser la postcondition de manière différente de manière à contre-vérifier le calcul fait dans la méthode. Ici, on peut vérifier que les valeurs de `r1` et `r2` sont des racines de l'équation du second degré en utilisant comme postcondition :

```
Math.abs(a*r1*r1 + b*r1 + c) < 0.0000001 &&
Math.abs(a*r1*r2 + b*r2 + c) < 0.0000001
```

où 0.0000001 est un seuil de tolérance admis dans l'erreur d'arrondi qui peut se produire dans le calcul.

2.2 Données, encodage et représentation

Après notre première incursion dans les énoncés de la programmation impérative, revenons sur les données et sur l'information, puisque c'est la principale utilisation des ordinateurs que le traitement des données et de l'information en général. Nous avons vu que la seule information manipulable par un ordinateur est l'information binaire. La conséquence importante de cet état de chose est que toute information autre que des nombres binaires entiers positifs devra être encodée en binaire pour être manipulée.

Les informations directement manipulées par l'ordinateur, c'est-à-dire pour lesquelles l'ordinateur dispose d'instructions opérant sur ces données, sont en réalité relativement peu nombreuses. Il s'agit de nombres entiers et réels, de caractères, de valeurs booléennes, et de plus en plus de sons et d'éléments d'images (*pixels* de *picture elements*). Pour toutes ces informations, des encodages sont prévus par les ordinateurs. Un encodage est pour l'essentiel une correpondance entre des valeurs et des nombres binaires ainsi que des opérations sachant traiter ces nombres binaires selon la signification qu'ils ont en fonction de l'information traitée.

En théorie, l'encodage peut être réalisé par une fonction mathématique associant à chaque valeur d'information (nombre, caractère, etc.) un code parmi l'ensemble des codes choisis (des nombres binaires bien sûr lorsqu'on encode pour un ordinateur). Soit I un ensemble de valeurs d'information et C un ensemble de codes possibles, une fonction d'encodage $enc : I \to C$ prend une valeur d'information $i \in I$ et lui associe un code $c \in C$. Idéalement, la fonction enc devrait être *totale* et être une *application injective*, c'est-à-dire :

(totale) Pour tout $i \in I$, il existe un $c \in C$ tel que $c = enc(i)$,
(application) Pour tout $i \in I$, il existe un unique $c \in C$ tel que $c = enc(i)$, et
(injective) Pour tout $c \in C$, il existe au plus un $i \in I$ tel que $c = enc(i)$.

Prenons un exemple. Un entier positif est aisément encodable en binaire en appliquant simplement un changement de base. L'opération d'addition sur les entiers positifs en base 2 s'applique alors sur l'encodage des deux opérandes et donne comme résultat un entier en base 2. Le principe même de l'encodage est que cet entier en base 2 est bien sûr l'encodage en binaire du résultat de l'opération d'addition des deux opérandes initiales. Dans ce cas, la fonction d'encodage peut s'exprimer analytiquement (conversion en base 2). Le schéma suivant exprime l'idée poursuivie :

$$\begin{array}{ccc} \text{opérande binaire} & \xrightarrow[+_2]{} & \text{résultat binaire} \\ \uparrow & & \downarrow \\ \text{opérande} & \xrightarrow[+]{} & \text{résultat} \end{array}$$

On notera cependant que la fonction en question ne peut être totale tout en étant une application injective. Sur un mot de taille finie, on ne pourra jamais représenter qu'un nombre fini de valeurs. L'ensemble des entiers étant infini, tous les entiers ne peuvent être représentés. On limite donc l'intervalle des entiers représentés, nous contentant alors d'une fonction d'encodage partielle. Pour des mots de 32 bits, l'intervalle des entiers naturels le plus couramment représenté par conversion en base 2 est $[0, 4.294.967.295]$.

Prenons un second exemple où la fonction d'encodage n'est pas exprimable par une expression analytique. Tous les ordinateurs manipulent aujourd'hui des caractères. Les opérations réalisables sur des caractères sont relativement simples : on peut les comparer entre eux et les afficher sur un écran ou les écrire sur du papier. Pour manipuler les caractères à l'aide d'un

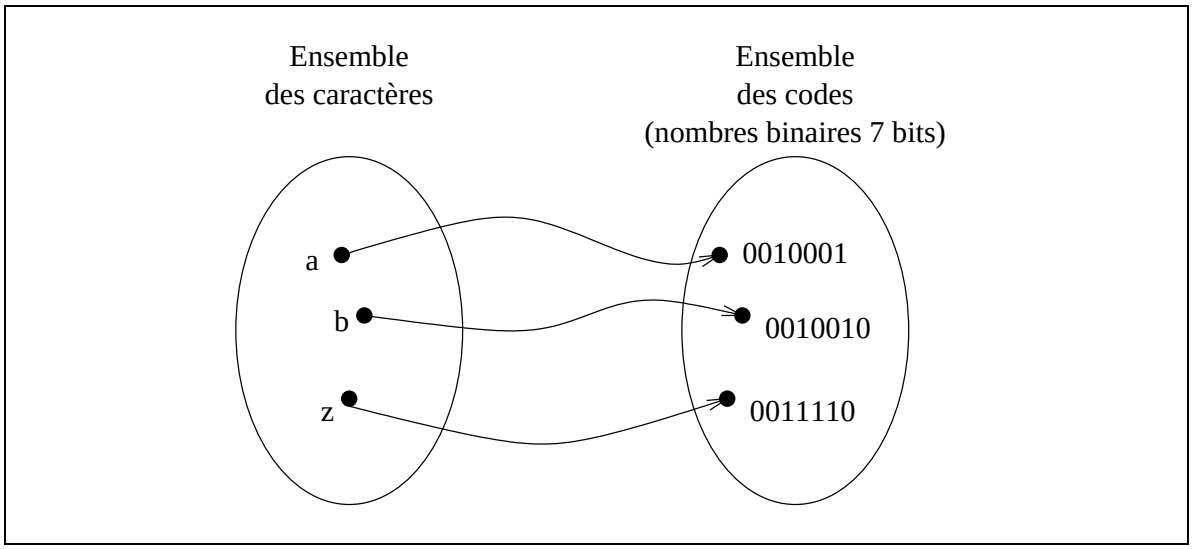


FIG. 2.4 – Encodage des caractères

ordinateur, il faut les encoder en binaire. Dans le cas des caractères, il n'existe pas d'encodage s'imposant a priori comme dans le cas des entiers positifs. Il faut simplement en choisir un. Il suffit donc d'établir une correspondance entre chacun des caractères et un nombre binaire (fig. 2.4). Comme les caractères sont relativement peu nombreux dans les alphabets occidentaux, on limite la taille des codes de caractères à 8 bits, ce qui permet de représenter $2^8 = 256$ caractères différents (caractères normaux, accentués, ponctuation, etc.).

La fonction d'encodage ne peut être exprimée analytiquement, puisque la correspondance est arbitraire. Cependant, la cardinalité de l'ensemble des valeurs d'information étant égale à celle des codes, elle peut être totale et être une application injective (en fait surjective).

À l'heure d'Internet, on comprendre que cet encodage a intérêt à être partagé par le maximum d'ordinateurs, de façon à pouvoir afficher correctement sur un ordinateur A un texte composé sur un ordinateur B. Très vite des normes d'encodage des caractères se sont imposées. Ce fut d'abord le cas aux États-Unis avec la norme ASCII. La norme ASCII définit une correspondance entre un ensemble de 128 caractères vers les nombres binaires sur 7 bits, de `0000000` à `1111111` (fig. 2.4). Le jeu de caractères ASCII comprend les 26 lettres, en majuscules et en minuscules, les 10 chiffres de 0 à 9, des caractères de ponctuation (virgule, point, point-virgule, deux points, ...) et des caractères de contrôle (retour à la ligne, tabulation, etc.). Presque tous les ordinateurs acceptent aujourd'hui les caractères encodés selon la norme ASCII. Un caractère est la plupart du temps encodé sur un octet ; en norme ASCII (7 bits), le bit de poid fort de l'octet est alors à 0.

Des normes se sont ensuite développées au niveau international avec les jeux de caractères ISO, dont le ISO-8859 qui est le jeu de caractères comprenant les caractères accentués français. Plus récemment, la norme UNICODE permet de représenter en une seule norme les caractères de la plupart des alphabets du monde, mais elle nécessite 16 bits pour représenter un caractère.[6]

[6]Le langage Java utilise la norme UNICODE pour représenter ses caractères.

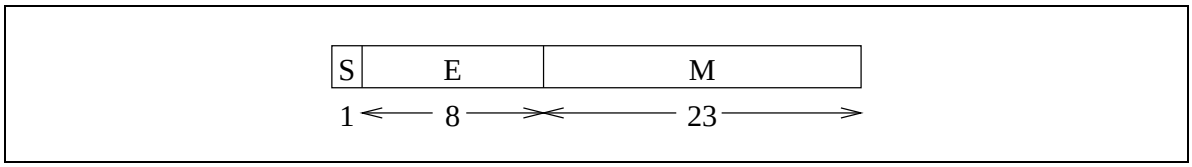


FIG. 2.5 – Représentation des réels selon la norme IEEE

De l'encodage à la représentation des données

Lorsque la relation entre un élément d'information et son pendant binaire se comprend, grosso modo, comme une relation un à un, on parle volontier d'encodage. Par contre, lorsque la relation se comprend plutôt comme l'agrégat de plusieurs encodages, on parle plutôt de *représentation*. Le cas le plus simple de représentation est celui des entiers relatifs, comprenant donc les entiers négatifs. La représentation la plus simple consiste à encoder la valeur absolue du nombre par conversion en base 2, puis d'accoller un bit de signe valant 0 si le nombre est positif et 1 s'il est négatif. On parle ici de représentation car l'encodage se comprend en deux morceaux disjoints. En réalité, cette représentation est peu utilisée. Les entiers relatifs sont généralement encodés par conversion en base 2 puis par complément à 2, encodage que nous ne détaillerons pas ici

Les nombres réels ont aussi fait l'objet de nombreuses représentation sur différentes machines avant d'en arriver à des normes du type de la norme IEEE (Institute for Electrical and Electronic Engineering). Un réel peut s'exprimer par un facteur multiplié par une certaine puissance de dix. Une première normalisation consiste à toujours représenter les réels dans une forme canonique, c'est-à-dire où la partie entière du facteur a une valeur entre 1 et 9 en valeur absolue. Ainsi les réels :

$100,0$, $0,0067$ et $-1,5$

sont représentés en forme canonique par :

$1,0 \times 10^{2}$, $6,7 \times 10^{-3}$ et $-1,5 \times 10^{0}$.

Il faut alors représenter le facteur qui est un nombre réel avec une partie entière d'un chiffre et une mantisse, plus un exposant qui est un entier relatif. La norme IEEE pour les réels procède de manière similaire, mais avec un nombre représenté en base 2. Pour un réel en base deux, la partie entière du facteur est nécessairement 1, puis il y a une mantisse donnant le nombre de chiffre significatifs, plus un exposant de deux (en lieu et place de l'exposant de 10). Dans le cas d'un réel simple précision sur 32 bits, la norme IEEE prévoit de représenter la mantisse sur 23 bits, plus un bit de signe. La partie entière étant toujours 1, elle n'est pas représentée explicitement. L'exposant est un entier relatif représenté sur 8 bits, selon le mode complément à 2. La figure 2.5 donne l'ordre dans lequel sont placée les différentes parties dans le mot mémoire de 32 bits.

Outre les données de base manipulables par l'ordinateur, toute information doit être représentée sous forme binaire pour être traitée. L'informaticien doit donc toujours définir la représentation d'une information avant d'écrire les programmes qui vont traiter ces informations. Un nom et une adresse, par exemple, pourront êtes représentés par une séquence de caractères. Un date devra être représentée par une séquence de caractères 0 à 9, deux pour le jour, deux pour le mois et deux à quatre pour l'année. Le 25 mai 1999 peut donc être représenté par trois groupes de caractères : un groupe de deux caractères 25 pour le jour, un second groupe de deux caractères 05 pour le mois, et enfin un groupe de deux ou quatre caractères pour l'année. Si l'année est représentée sur deux caractères, le siècle est implicite. Dans la plupart des pro-

grammes ce siècle implicite est le vingtième, d'où l'interprétation de l'encodage 010100 comme représentant le 1 janvier 1900. Ce siècle implicite fut à l'origine des problèmes de passage à l'an 2000 des ordinateurs et de leurs logiciels.

La représentation d'informations conduit à construire des agrégats de données élémentaires qui, selon une interprétation, permet de traiter la représentation puis de retrouver l'information résultante par décodage de l'interprétation du résultat. Cette façon de représenter une information par un agrégat de données élémentaires est fondamentale en informatique. En termes de mémoire de la machine, ces agrégats de données élémentaires seront conservées dans un groupe de mots mémoire. Un élément important de compréhension d'un programme est certainement la facilité avec laquelle le lecteur du programme saura reconnaître qu'un certain groupe de données élémentaires ne sont pas indépendantes mais plutôt qu'elles forment un groupe cohérent représentant une information précise. Reconnaître facilement l'interprétation de cette représentation est également crucial pour la compréhension d'un programme.

Exemple 5 Un rectangle dont les côtés sont parallèles aux axes du plan cartésien peut être représenté de plusieurs façons. On a vu qu'on peut le représenter par la position dans le plan de ses points supérieur gauche et inférieur droit. On peut aussi le représenter par le coin inférieur gauche ainsi que sa hauteur et sa largeur. Sachant qu'un point est représenté par son abcisse et son ordonnée, on voit que dans les deux cas, le rectangle est représenté par quatre nombres réels. Il est donc important pour comprendre un programme manipulant ces rectangles de reconnaître quels sont les quatre nombres entrant dans la représentation de chaque rectangle, ainsi que l'interprétation à leur donner (deux points ou un point et des mesures de hauteur et largeur). □

Typage

Toute donnée étant représentée en binaire, l'examen d'une série de bits ne donne aucune information sur la donnée que ces bits représentent. C'est l'interprétation de ces bits qui importe. Une série de bits donnée formant un mot mémoire, comme par exemple :

```
01100111010110100110000110001001
```

peut aussi bien représenter un entier naturel encodé en binaire, un entier relatif encodé avec un bit de signe ou encore en complément à 2, quatre caractères représentés en ISO-8859, ou encore un réel selon la norme IEEE.

C'est dire que si nous désignons ce mot mémoire par une variable, disons `v`, il est utile d'indiquer comment interpréter le contenu de cette variable. C'est le rôle du *typage*. Un type désigne tout à la fois une interprétation de la série de bits constituant une donnée, interprétation qui associe une valeur à la série de bits comme nous l'avons vu, que les opérations applicables aux valeurs de ce type. Dans la plupart des langages de programmation, on associe un type à une variable qui devient alors une propriété de cette variable : ce type va servir à s'assurer que chaque fois qu'on utilise le contenu de cette variable on l'utilise bien selon la bonne interprétation. Cette vérification est appelée *vérification de type*.

Exemple 6 Soit `v1` et `v2` deux variables. Si on écrit l'énoncé d'affectation :

```
v1 = v2
```

la vérification de type consiste à s'assurer avant de transférer le contenu de la variable `v2` dans la variable `v2` que ces deux variables sont bien de même type, pour être sûr que l'interprétation du contenu de `v1` soit bien conforme à la valeur qui est effectivement dans cette variable. □

De nombreuses opérations sont typées, c'est-à-dire qu'elles ne s'appliquent qu'à des données dont l'interprétation est compatible avec ce que fait l'opération. Une addition peut se faire sur des données de type entier ou réel, mais n'a pas de sens sur des données de type caractère. Si on écrit l'énoncé :

```
v1 = v2 + 10
```

l'interprétation du contenu de `v2` doit être compatible avec le fait de faire une opération d'addition sur cette valeur. De plus, la mise en œuvre de l'opération d'addition dépend de la représentation des données sur lesquelles elle s'applique. L'addition sur les entiers relatifs ne se fait pas de la même façon selon qu'ils sont représentés en valeur absolue plus un bit de signe ou en complément à 2.

Types abstraits

L'équation suivante résume de façon concise ces idées :

$$\text{Type en informatique} = \text{valeurs} + \text{opérations} + \text{représentation/encodage}$$

Les opérations définies par un type sont réalisées en pratique par des procédures qui tiennent compte de l'encodage et de la représentation des valeurs du type. L'une des idées fondamentales en informatique, sur laquelle nous reviendrons plus loin (chapitre 3), est celle des types abstraits de données. La notion de type abstrait de données réalise la séparation entre la mise en œuvre d'un type et son utilisation. Bien sûr, lorsqu'on définit un type comme le type réel IEEE, la mise en œuvre des différentes opérations nécessitent une connaisance précise de la représentation des valeurs de ce type. Cette connaissance n'est cependant pas très utile à l'utilisateur de ce type, pour qui la seule chose importante est de savoir comment créer des valeurs du type réel et comment réaliser des opérations sur ces valeurs.

Un type abstrait de données est un type où il y a une séparation nette entre mise en œuvre et utilisation, séparation telle que l'utilisateur n'a pas à connaître et ne connait ni la représentation des valeurs du type, ni la mise en œuvre précise des opérations sur ces valeurs. Un type abstrait présente à l'utilisateur une vision externe, on dit souvent une *interface*, qui lui indique comment créer des valeurs et comment utiliser les différentes opérations.

La plupart du temps, les types primitifs fournit par les langages de programmation sont des types abstraits. Les réels en sont un exemple. Dans à peu près tous les langages, la création d'un réel se fait en écrivant le nombre syntaxiquement dans le programme et les opérations sont accessibles via les opérateurs arithmétiques courants. Nul n'est besoin de comprendre les subtilités de la norme IEEE pour utiliser les réels dans un programme. De la même façon, les listes en CaML forment un type de données abstraits dans la mesure où on peut les créer et leur appliquer les opérations `hd`, `tl`, etc., sans connaître quoi que ce soit à leur mise en œuvre.

Le grand avantage des types abstraits de données pour les types primitifs est l'indépendance des programmes par rapport au choix de représentation et d'encodage qui souvent dépendent des machines. Pour les types conçus par le programmeur (comme par exemple un type rectangle), les mêmes avantages tiennent : l'utilisateur est isolé des questions de mise en œuvre et donc la mise en œuvre peut éventuellement être modifiée sans que les programmes utilisant le type soient touchés. Il y a donc une diminution du couplage entre les parties du programme utilisant le type et celles qui le mette en œuvre, ce qui limite la portée des changements apportés au programme.

Représentation et assertions

Le typage est le premier moyen permettant de s'assurer d'une certaine cohérence dans l'utilisation des données binaires pour représenter une certaine information. Il permet donc d'assigner une interprétation significative à une zone de mémoire qui autrement ne serait vue que comme un patron de bits, de 0 et 1, quelconque. Les systèmes de typage des langages de programmation ne sont généralement pas suffisants pour exprimer toutes les contraintes d'une représentation cohérente d'une information.

Reprenons notre exemple du rectangle[7] représenté par deux points : son coin supérieur gauche et son coin inférieur droit. La typage permet de s'assurer que l'agrégat de quatre mots mémoires utilisés pour représenter le rectangle est interprété comme quatre nombre réels qui sont les abscisses et les ordonnées des deux points. Remarquons que cela n'est pas suffisant pour que deux points, interprétés comme un coin supérieur gauche et un coin inférieur droit, forment bel et bien un rectangle. Il faut aussi que l'abcisse du coin inférieur droit soit strictement supérieure à celle du coin supérieur gauche, et que l'ordonnée du coin inférieur droit soit strictement inférieure à celle du coin supérieur gauche.

Ce genre de contrainte peut s'exprimer avec une assertion. Si `x1` et `x2` sont respectivement les abcisses du coin supérieur gauche et du coin inférieur droit, et que `y1` et `y2` en sont les ordonnées respectives, l'assertion suivante exprime la propriété de cohérence de cette représentation du rectangle :

$$\{\texttt{x1} < \texttt{x2} \wedge \texttt{y2} < \texttt{y1}\}$$

Cette assertion doit être vraie en tout point du programme où la représentation du rectangle est stable, c'est-à-dire pas en cours de modification. On peut admettre en effet que si une translation du rectangle est faite, il y a un point dans le programme auquel l'abcisse ou l'ordonnée d'un des points a été modifié et pas ceux de l'autre point, moment transitoire pendant lequel la représentation du rectangle est incohérente. Considérons par exemple la séquence d'affectations suivante effectuant une translation du rectangle par un certain Δ_x et un certain Δ_y :

```
x1 = x1 + delta_x ;
y1 = y1 + delta_y ;
x2 = x2 + delta_x ;
y2 = y2 + delta_y ;
```

Entre la première affectation (à `x1`) et la dernière affectation (à `y2`), il est tout à fait possible que l'assertion précédente soit fausse. Par exemple, si on applique la translation `delta_x` = 10 et `delta_y` = -15 au rectangle représenté par :

$$\{\texttt{x1} = 20 \wedge \texttt{y1} = 20 \wedge \texttt{x2} = 25 \wedge \texttt{y2} = 10\}$$

l'annotation de l'exécution de la séquence donne :

$\{\texttt{x1} = 20 \wedge \texttt{y1} = 20 \wedge \texttt{x2} = 25 \wedge \texttt{y2} = 10 \wedge \texttt{delta\_x} = 10 \wedge \texttt{delta\_y} = -15\}$

```
x1 = x1 + delta_x ;
```

$\{\texttt{x1} = 30 \wedge \texttt{y1} = 20 \wedge \texttt{x2} = 25 \wedge \texttt{y2} = 10 \wedge \texttt{delta\_x} = 10 \wedge \texttt{delta\_y} = -15\}$

```
y1 = y1 + delta_y ;
```

[7] dont les côtés sont parallèles aux axes du plan cartésien.

$\{\texttt{x1} = 30 \wedge \texttt{y1} = 5 \wedge \texttt{x2} = 25 \wedge \texttt{y2} = 10 \wedge \texttt{delta\_x} = 10 \wedge \texttt{delta\_y} = -15\}$

```
x2 = x2 + delta_x ;
```

$\{\texttt{x1} = 30 \wedge \texttt{y1} = 5 \wedge \texttt{x2} = 35 \wedge \texttt{y2} = 10 \wedge \texttt{delta\_x} = 10 \wedge \texttt{delta\_y} = -15\}$

```
y2 = y2 + delta_y ;
```

$\{\texttt{x1} = 30 \wedge \texttt{y1} = 5 \wedge \texttt{x2} = 35 \wedge \texttt{y2} = -5 \wedge \texttt{delta\_x} = 10 \wedge \texttt{delta\_y} = -15\}$

On constate facilement que l'assertion de cohérence du rectangle n'est pas vérifiée après la première affectation et ce, jusqu'après la dernière affectation. On remarque que ces points de programme correspondent à la modification de la représentation par une opération sur le rectangle. Si on excepte ces points de programme qui jouent un rôle particulier, l'assertion doit toujours être vérifiée.

Une assertion qui est vérifiée ainsi durant toute la durée du programme est appélée un *invariant*. Dans le cas des invariants sur les représentations, on les appelle *invariant structurel de la représentation*. À chaque fois qu'une représentation d'une information est conçue, il est utile et important de savoir exprimer l'invariant structurel de cette représentation. Cet invariant structurel sera d'une aide précieuse pour détecter dans le programme les traitements (fonctions, procédures, etc.) incorrects dans la mesure où ils laisseraient une représentation dans un état violant son invariant.

Une observation intéressante à faire ici est que la méthode peut permettre de délimiter et de localiser les points de programmes où un certain invariant structurel d'une représentation n'est pas vérifié. Reconsidérons la translation et abstrayons ce traitement dans une procédure dont les paramètres sont la variation en x et y appliquée au rectangle :

```
void translation(double delta_x, double delta_y)
{
  x1 = x1 + delta_x ;
  y1 = y1 + delta_y ;
  x2 = x2 + delta_x ;
  y2 = y2 + delta_y ;
}
```

Si on exécute cette méthode, on se rend compte que si l'invariant structurel du rectangle est vérifié à l'entrée de la méthode `translation`, comme le montre la première assertion avant l'affectation à `x1` dans la trace précédente, et il est à nouveau vérifié à la sortie de la méthode `translation`, comme le montre la dernière assertion après l'affectation à `y2`. L'abstraction de la translation par une méthode permet donc de d'enfermer les points de programmes où l'invariant structurel n'est pas vérifié. En systématisant cette approche, il devient plus facile de comprendre un programme car on peut garantir que partout sauf dans les méthodes manipulant une certaine représentation, l'invariant structurel de cette représentation est vérifié. Les méthodes manipulant une certaine représentation étant en nombre relativement faible, cela limite les endroits où la compréhension du programme nécessite de comprendre les détails de modifications d'une représentation.

2.3 Objet comme abstraction de la mémoire et des opérations

En pratique, l'état de l'ensemble de la mémoire d'un ordinateur personnel est quelque chose de difficile à imaginer tant la taille devient grande. On s'intéresse donc à des moyens de diviser cette mémoire en plus petites entités que l'on pourra comprendre individuellement. C'est le bon vieux principe du *diviser pour règner* qui s'applique pour diviser la mémoire en unités modulaires.

Prenons une analogie avec les mathématiques. Le savoir mathématique est structuré en théories, qui elles-mêmes sont structurées en axiomes et théorèmes. Chaque théorème peut être compris individuellement. Une fois qu'on a compris la preuve du théorème, on peut l'utiliser dans la preuve d'un autre thèorème, sans avoir à comprendre à nouveau sa preuve.

En informatique, on cherche à procéder de la même façon. On cherche à diviser un programme en entités plus petites. On veut que ces entités soient compréhensibles individuellement, puis utilisables par d'autres entités. L'utilisation de procédures et de fonctions va dans ce sens. On peut comprendre individuellement ce qu'une procédure *p1* réalise et comment elle le réalise. Ensuite, on peut utiliser cette procédure dans une autre procédure *p2* ; pour comprendre *p2*, il faudra comprendre le rôle joué par *p1*, mais pour cela il suffit de comprendre ce que fait *p1* et non comment elle le fait.

Exemple 7 Pour trouver la racine carrée d'un nombre, il y a deux approches classiques. On peut utiliser le développement en série de la fonction racine carrée ce qui donne un polynôme en x approximant cette fonction ; pour calculer la racine carrée d'un nombre X, il suffit alors d'évaluer le polynôme pour $x = X$. L'autre approche est celle de Newton qui consiste à poser l'équation $x^2 - X = 0$ et trouver une valeur de x qui solutionne l'équation ; la recherche de cette valeur se fait par approximation successive.

Dans un programme où on utilise une fonction racine carrée, il est facile de comprendre par examen de cette fonction comment la racine carrée est calculée. Lorsqu'une autre procédure utilise la fonction racine carrée, il suffit de savoir que la fonction calcule la racine carrée pour comprendre la procédure. □

Pendant longtemps, la procédure a été le moyen de découpage des applications. L'ennui est que la procédure n'exprime que le calcul. Le découpage en procédures permet donc de mieux comprendre le calcul réalisé par un programme, mais pas de comprendre les données. On a vu précédemment que les données se présentent le plus souvent comme des agrégats de données élémentaires qui sont interprétées par le programmeur et par les procédures comme formant un tout cohérent représentant quelque chose qui transcende chaque données élémentaire.

La compréhension d'un agrégat de données élémentaires est donc indissociable de son interprétation, laquelle est visible dans les procédures qui traitent ces données. Et vice versa, le calcul réalisé par une procédure sur un agrégat de données élémentaires n'est compréhensible que dans la mesure où on y reconnaît l'interprétation de quelque chose de plus grand et de cohérent.

Exemple 8 Si on revient sur l'exemple du rectangle représenté par deux points, eux-mêmes représentés par deux coordonnées réelles. On ne comprendra qu'une procédure réalise la translation du rectangle que si on comprend qu'elle réalise la translation des deux points, et on ne réalisera qu'un procédure tranlate un point que dans la mesure où on comprend que l'addition aux deux coordonnées représente bien une translation. On voit donc bien comment la com-

préhension de la représentation du rectangle conditionne la compréhension de la procédure de translation, et vice versa. □

Objet = données + procédures

L'objet a été inventé pour unifier données et procédures. Il s'agit donc de mettre ensembles l'agrégat de données représentant une entité particulière et les procédures qui opérent sur ces données.

Un objet peut être vu comme un petit ordinateur : il possède une mémoire, constituée de références à d'autres objets et de données élémentaires, et il possède des procédures qui représentent les calculs que l'on peut lui faire faire. On peut parler de l'état d'un objet, c'est-à-dire l'ensembles des valeurs courantes de ses variables. Les variables d'un objet forment une représentation pour l'entité modélisée par cet objet. Comme nous l'avons vu précédemment, une représentation obéit la plupart du temps à un invariant qui détermine la propriété des états des variables qui correspondent à un état cohérent de la représentation. Dans le cas d'un objet, on parle généralement d'*invariant d'objet* ou encore d'*invariant structurel de l'objet.* L'invariant d'un objet est respecté en tout point de programme où l'exécution n'est pas au sein d'une méthode de l'objet ; lorsqu'on est au sein d'une méthode de l'objet, comme la méthode de translation du rectangle, il se peut qu'en certains points l'état de l'objet soit incohérent parce qu'il est en cours de modification.

Les méthodes d'un objet opèrent donc sur cet objet en modifiant son état. L'état d'un objet vérifiant un certain invariant structurel, il est aussi possible de comprendre ses méthodes en termes de préconditions sur l'état de l'objet et sur les valeurs des paramètres de la procédure, et de postconditions sur l'état de l'objet et sur le résultat éventuel retourné par la procédure (fonctionnelle). Le grand avantage de cette approche est ensuite que l'objet peut se comprendre uniquement en terme des pré- et post-conditions sur ses méthodes, et éventuellement de son invariant structurel.

Outre l'exemple du rectangle qu'il est possible de représenter comme un objet, voici un exemple désormais classique d'objet.

Exemple 9 Historiquement, l'un des premiers exemples d'objet a été la tortue Logo[8]. En Logo, la tortue apparaît à l'écran et répond aux ordres *avance* ou *recule* d'un certain nombre de pas, ainsi que *gauche* et *droite* pour changer de direction. Elle porte également une plume qui, lorsqu'abaissée, laisse une trace sur son sillage.

L'objet informatique représentant la tortue Logo possède une position, elle-même représentée par une abcisse et une ordonnée dans les coordonnées de l'écran, les variables `x` et `y`. L'objet tortue possède également un `cap` (un angle par rapport à l'axe des abcisses) et elle possède un booléen `estLevee` dont la valeur est vrai si la plume de la tortue est levée et faux si elle est abaissée. Les requêtes que peut recevoir la tortue introduisent naturellement les méthodes correspondantes : `avance` et `recule` avec pour argument un nombre entier de pas à faire, `gauche` et `droite` pour faire changer le cap, ainsi que `montePlume` et `descendPlume` pour débuter et finir un tracé.

[8]Logo est un langage développé par Papert [Pap80] à la fin des années '70 pour permettre l'utilisation des ordinateurs par des élèves de niveau élémentaire. Le systèmes Logo visualise à l'écran une tortue et l'enfant interagit avec le systèmes an donnant des ordres à cette tortue. La tortue est donc l'élément principal d'interaction de ce langage. L'objectif de Logo était d'apprendre aux élèves des notions élémentaires de géométrie en dessinant des formes sur l'écran.

L'invariant d'objet de la tortue Logo est lié à la représentation de l'écran sous forme d'un plan cartésien avec des limites précises. Si on considère le premier quadrant borné par `MaxX` en abcisse et `MaxY` en ordonnée, l'invariant d'objet spécifierait que l'état de la tortue doit comprendre une position à l'intérieur des bornes du quadrant et un cap dont la valeur est un angle «légal» :

$$\{0 \leq \texttt{x} \leq \texttt{MaxX} \wedge 0 \leq \texttt{y} \leq \texttt{MaxY} \wedge 0 \leq \texttt{cap} \leq 360\}$$

□

L'application systématique de l'approche qui définit un objet et ses méthodes autour d'un invariant structurel et de pré- et post-conditions s'appelle *programmation contractuelle*. La notion de contrat ici vient de l'idée selon laquelle si l'appelant d'une méthode d'un objet fournit des paramètres respectant la pré-condition de la méthode, il lui est garanti que le résultat rendu respectera la post-condition de la méthode (ceci peut inclure des propriétés sur l'état de l'objet au début de la méthode et à la fin). La programmation contractuelle sera introduite de manière plus complète au chapitre 6.

2.4 Éléments de savoir-faire : types primitifs, expressions et méthodes

2.4.1 Types de données primitifs et littéraux de Java

Rares sont les langages à objets qui manipulent toutes les entités comme des objets.[9] La plupart des langages définissent certains types de données comme étant primitifs, c'est-à-dire qu'ils pré-existent aux objets et sont manipulés de manière différente des objets. Notons qu'il n'est pas si simple de voir tout comme des objets. Un objet contenant de l'information, s'il ne contient que des références à d'autres objets, se pose la question de la poule et l'œuf, et plus précisément quand un objet contient-il une donnée effective. On doit bien atterrir quelque part, de la même façon que les listes Scheme finissent bien par contenir des éléments atomiques (nombres, caractères, etc.).

Les types atomiques ou primitifs de Java sont donc au nombre de 9. Leurs noms, ainsi que différentes informations apparaissent à la figure 2.6. Ce tableau vous donne la taille en bits nécessaire pour mémoriser une valeur du type en question, ses valeurs minimales et maximales (si cela s'applique), la valeur par défaut du type et le nom de la classe permettant de créer des objets réalisant sous forme d'objets des valeurs du type correspondant.

Les caractères sont représentés sur 16 bits et utilise l'encodage de la norme Unicode. Les types intégraux (entiers) `byte`, `short`, `int` et `long` utilisent tous une représentation en complément à 2. Tous ces types représentent des nombres entiers ; la différence entre chacun d'entre eux est la taille mémoire utilisée pour représenter les valeurs du type, et donc la plage des valeurs représentables dans chaque cas. Le tableau de la figure 2.6 donne les tailles et les plages pour chacun de ces types.

Les types réels `float` et `double` utilisent une représentation suivant la norme IEEE754. C'est donc cette norme qui spécifie la taille de la représentation, la plage des valeurs représentées et la précision disponible.

[9] Pour votre culture personnelle, sachez que le langage Smalltalk, l'un des premiers et des plus connus parmi les langages à objets, en est une exception notable.

Type	*Taille*	*Mini*	*Maxi*	*Défaut*	*Classe*
`boolean`	1 bit	–	–	`false`	`Boolean`
`char`	16 bits	Uni0	Uni$2^{15}-1$	Uni0	`Character`
`byte`	8 bits	-2^7	2^7-1	(byte)0	`Byte`
`short`	16 bits	-2^{15}	$2^{15}-1$	(short)0	`Short`
`int`	32 bits	-2^{31}	$2^{31}-1$	0	`Integer`
`long`	64 bits	-2^{63}	$2^{63}-1$	0l	`Long`
`float`	32 bits	IEEE754	IEEE754	0.0f	`Float`
`double`	64 bits	IEEE754	IEEE754	0.0	`Double`
`void`	–	–	–	–	`Void`

FIG. 2.6 – Informations sur les types primitifs de Java

On a vu que les types servent dans la déclaration des variables, des paramètres formels et du retour des méthodes. Une déclaration de variable dans un programme entraîne l'allocation d'un espace suffisant pour contenir les valeurs du type de la variable lors de l'exécution. Par exemple, la déclaration :

```
int i ;
```

implique l'allocation d'un espace d'au minimum 32 bits pour contenir une valeur entière, et le système va relier le nom de la variable `i` à l'emplacement mémoire alloué. Java ne tolère pas d'avoir des variables dont le contenu n'a pas reçu de valeur significative, valeur qui dépend bien sûr du type. La colonne 'Défaut' dans le tableau donne la valeur utilisée pour initialiser les variables selon leurs types. Par exemple, un variable entière `i` sera initialisée par la valeur 0, qui est la valeur par défaut pour les valeurs de type `int`.

Le langage Java étant un langage à objets, il est parfois utile de manipuler les valeurs de types primitifs comme des objets. À cet effet, Java propose des classes permettant de créer des objets pouvant jouer le rôle de valeur de chacun des types primitifs. Nous reviendrons plus loin sur l'utilisation de ces classes.

Les constantes en Java sont typées. Lorsqu'on écrit la constante `6.23` dans un programme Java, cette valeur a un type qui est par défaut `double`.[10] La colonne *Défaut* du tableau de la figure 2.6 illustre la syntaxe utilisée en Java pour produire des constantes de chacun des types. Le tableau de la figure 2.7 complète cette illustration. Bien que le type `String` ne soit pas un type primitif, il existe également une syntaxe permettant d'écrire des constantes chaînes de caractères entre guillemets, comme dans `"abc"`.

Lorsqu'une variable est déclarée du type `long`, par exemple, on peut construire des valeurs de type `long` pour les ranger dans la variable :

```
long uneVariableLong = 100L ;
```

En général cependant, le langage permet certaines conversions de type. Par exemple, la déclaration et affectation suivante est correcte :

[10] Contrairement à beaucoup de langages, les valeurs réelles en Java sont par défaut des valeurs en double précision (c'est-à-dire du type `double`).

Type	*syntaxe des constantes littérales*
`boolean`	`true`, `false`
`char`	`'a'`, `'b'`, ...
`byte`	`(byte)0`, `(byte)1`, ...
`short`	`(short)0`, `(short)1`, ...
`int`	`0`, `1`, ...
`long`	`0L`, `1L`, ...
`float`	`0.0f`, `1.3e4f`, ...
`double`	`0.0`, `1.3e4`, ...

FIG. 2.7 – Syntaxe pour l'écriture de constantes des types primitifs en Java

```
long uneVariableLong = 100 ;
```

Ici, la valeur `100` est de type `int` (voir le tableau de la figure 2.7), mais elle est convertie implicitement vers le type `long` avant d'être rangée dans la variable `uneVariableLong`. De façon générale, Java convertit implicitement les valeurs des types «plus petits» vers les types «plus gros», comme par exemple de `int` vers `long`, car la conversion n'implique pas de perte. Cependant, les conversions de types «plus gros» vers les «plus petits» ne sont pas faites implicitement car elles peuvent entraîner des pertes de précisions ou des débordement de capacité. La déclaration–affectation :

```
short uneVariableShort = 100 ;
```

est illégale, car on ne peut mettre une valeur du type `int` dans une variable de type «plus petit» `short`. Il faut convertir explicitement la valeur 100 en une valeur de type short avant l'affectation. Pour opérer une telle conversion, il faut le faire explicitement en utilisant un opérateur de conversion de type. Un opérateur de conversion de type est simplement le nom du type entre paranthèse, et il s'utilise en le plaçant devant la valeur à convertir :

```
short uneVariableShort = (short)100 ;
```

Les valeurs de type `byte` ou `short` sont toutes écrites en utilisant la conversion explicite de type, comme on peut le voir au tableau 2.7.

2.4.2 Expressions en Java

Les expressions en Java s'écrivent en notation infixe. Le tableau de la figure 2.8 résume les différents opérateurs prédéfinis en Java avec leur signification respective. Les expressions en notation infixe peuvent généralement être ambigües. L'expression artihmétique `1 + 2 * 3`, par exemple, n'a pas le même résultat selon qu'on évalue d'abord l'addition puis la multiplication ou l'inverse. De même, l'expression artihmétique `1 - 2 - 3` n'a pas le même résultat selon qu'on évalue les soustractions de gauche à droite ou de droite à gauche.

Il demeure toujours possible de parenthéser les expression de façon à forcer l'interprétation voulue. Cependant, pour éviter de surcharger les expressions de parenthèses, Java, comme

plusieurs autres langages, adoptent des conventions de précédence et d'associativité des opérateurs. Un opérateur ayant plus forte précédence s'évalue avant celui ayant une précédence moins forte. Par exemple, la multiplication a une précédence plus forte que l'addition. Dans notre exemple précédent, le résultat de `1 + 2 * 3` serait donc 7. De même, des opérateurs de même précédence s'évalue dans l'ordre prévu par leur associativité. Si l'associativité décidée est à gauche, les opérateurs s'évaluent de gauche à droite ; si elle est à droite, ils s'évaluent de droite à gauche. L'addition et la soustraction ont la même précédence et s'associe à gauche. Dans notre exemple précédent, le résultat de `1 - 2 - 3` est donc -4 (et non 2).

Le tableau de la figure 2.8 donne les opérateurs en ordre décroissant de précédence. L'associativité notée D veut dire droite, alors que G dénote l'associativité à gauche.

La conversion de types implicite s'applique également dans le cas des expresions mixtes, mélangeant différents types numériques. La règle générale consiste à convertir les valeurs de types «plus petits» vers les types «plus gros». Les conversions implicites se font au fil de l'évaluation de l'expression, donc selon l'ordre de priorité et l'associativité des opérateurs. Attention donc à l'affectation qui est un l'opérateur de plus basse priorité. Par exemple, l'énoncé de déclaration–affectation suivant :

```
double x = 10 / 3 ;
```

ne donnera vraisemblablement pas le résultat escompté. La division se faisant entre deux valeurs de type `int`, c'est l'opération de division entière qui s'applique ; le résultat sera 3, et il sera converti ensuite vers le type double, soit la valeur `3.0`, avant d'être affecté à la variable `x`. Si on veut obtenir le résultat $3,\overline{33}$ il faut convertir explicitement au moins une des deux valeurs :

```
double x = (double)10 / 3 ;
```

On remarque que l'opérateur de conversion de type étant de plus forte priorité que la division, il s'applique d'abord pour convertir la valeur `int` 10 en valeur `double` 10.0 ; la division réelle s'applique alors, avec conversion implicite de la valeur 3, et le résultat de type `double` est affecté à `x`.

2.4.3 Méthodes en Java (première version simplifiée)

Les méthodes de Java sont en réalité des *procédures fonctionnelles*. Comme toutes procédures, elles peuvent faire des effets de bords, en particulier sur l'état de l'objet en modifiant les valeurs de ses variables. À la manière des fonctions, elles peuvent aussi retourner un résultat. La déclaration d'une méthode doit donc faire intervenir au minimum les informations suivantes :

- un qualificatif (sur lequel nous reviendrons plus en détails en §3.5),
- le type de son résultat,
- son nom,
- la liste de ses paramètres formels, et
- son corps.

En Java, la forme simplifiée de la déclaration d'une méthode est :

```
<qualificatifs> <type-résultat> <nom>(<liste-paramètres-formels>)
{
```

	Opérateur	Type opérandes	Ass.	Opération réalisée
1	++	arithmétique	D	pré ou post-incrément (unaire)
	--	arithmétique	D	pré ou post-décrément (unaire)
	+, -	arithmétique	D	plus et moins unaire
	~	entiers	D	complément bit-à-bit (unaire)
	!	booléen	D	négation logique (unaire)
	(*type*)	tous	D	conversion de type
2	*, /, %	arithmétique	G	multiplication, division, reste
3	+, -	arithmétique	G	addition, soustraction
	+	chaînes	G	concaténation
4	<<	entiers	G	décalage vers la gauche
	>>	entiers	G	décalage vers la droite avec signe
	>>>	entiers	G	décalage vers la droite sans signe
5	<, <=	arithmétique	G	plus petit, plus petit ou égal
	>, >=	arithmétique	G	plus grand, plus grand ou égal
	instanceof	objet, type	G	comparaison de type
6	==	primitif	G	égal (valeur identique)
	!=	primitif	G	différent (valeur différente)
	==	objet	G	égal (réfère au même objet)
	!=	objet	G	différent (ne réfère pas au même objet)
7	&	entiers	G	et bit-à-bit
	&	booléen	G	et logique
8	^	entiers	G	ou exclusive bit-à-bit
	^	booléen	G	ou exclusive logique
9	|	entiers	G	ou bit-à-bit
	|	entiers	G	ou logique
10	&&	booléen	G	et conditionnel (court-circuit)
11	||	booléen	G	ou conditionnel (court-circuit)
12	? :	booléen, tous, tous	D	conditionnelle
13	=	variable, tous	D	affectation
	*=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, !=	variable, tous	D	affectation avec opération

FIG. 2.8 – Opérateurs Java

```
    <corps>
}
```

Le type du résultat peut être soit un type prédéfini de Java, comme `int`, `double`, etc. ou un nom de classe si le résultat est un objet. Le corps d'une méthode est constitué de déclarations de variables locales, comme celles que nous avons vues pour la méthode `main`, suivies d'une séquence d'énoncés.

La séquence d'énoncés peut comporter toutes les sortes d'énoncés de Java. Jusqu'à maintenant, nous avons vus deux sortes d'énoncés : les énoncés d'affectation et la séquence. Un

```
public class ExempleRacines
{
   double r1, r2;

      /**
       * @pre     b*b - 4*a*c >= 0.0 && a != 0.0
       * @post    r1 == (-b + Math.sqrt(b*b - 4*a*c))/(2*a) &&
       *          r2 == (-b - Math.sqrt(b*b - 4*a*c))/(2*a)
       */
      void racines(double a, double b, double c)
      {
         double t1, t2;

         t1 = Math.sqrt(b*b - 4*a*c);
         t2 = 2*a;
         r1 = (-b + t1)/t2;
         r2 = (-b - t1)/t2;
      }
}
```

FIG. 2.9 – Visibilité des variables dans les méthodes

énoncé joue un rôle particulier dans la définition des méthodes : l'énoncé `return`. Une méthode peut retourner un résultat. L'énoncé `return` sert à retourner explicitement ce résultat. L'énoncé `return` prend la forme suivante :

```
return <expression> ;
```

L'expression qui prend un peu une position d'argument du `return` doit avoir un résultat du type de celui déclaré comme résultat de la méthode. Lorsque l'exécution de la méthode arrive sur un énoncé `return`, l'expression qui lui est associée est évaluée et son résultat est retourné à l'appelant de la méthode. Un énoncé `return` peut apparaître partout où un énoncé peut apparaître dans un méthode. Comme il interrompt l'exécution de la méthode pour retourner un résultat à l'appelant, il apparaît le plus souvent en dernière position dans la méthode. Ceci n'est cependant pas obligatoire[11] ; s'il y a des énoncés après l'énoncé `return`, il ne seront tout simplement pas exécutés. Il est tout de même rare qu'on utilise un `return` autrement qu'à la fin d'une méthode. La dispersion d'énoncés `return` un peu partout au sein d'une méthode est plus souvent qu'autrement source de complexité et de difficulté de compréhension.

Il existe en Java un énoncé `return` sans valeur de retour qui permet d'interrompre l'exécution d'une méthode et de retourner immédiatement le contrôle au contexte appelant sans atteindre la fin de la méthode. On utilise que très rarement cette forme car elle introduit aussi une source de complexité et de difficulté de compréhension des programmes.

[11] Bien qu'en pratique, le compilateur Java essaie de vérifier que toutes méthodes déclrant un type de retour autre que `void` se termine bien par un énoncé `return`. Comme l'analyse de la méthode faite par le compilateur n'est pas parfaite, on est souvent obligé de mettre un énoncé `return` en fin de méthode pour éviter qu'il ne nous signale un avertissement (*warning*).

Variables visibles depuis une méthode

Une méthode a le pouvoir de modifier la valeur de variables par des énoncés d'affectation. Il est donc important de comprendre les limites imposées par Java à ce pouvoir par la notion d'encapsulation. L'ensemble des variables visibles depuis une méthode, et donc modifiables par elle, sont par défaut les variables locales à la méthode et les variables d'instance de l'objet sur lequel la méthode s'exécute.

La figure 2.9 illustre ces deux sous-ensembles par l'emboitement autour de la déclaration de la méthode `racines`, qui délimite les variables locales à la méthode, et l'emboitement autour de la définition de `ExempleRacines` qui délimite les variables d'instance des l'objets possédant et donc étant susceptible d'exécuter la méthode `racines`. À l'intérieur de la méthode `racines`, les énoncés ne peuvent utiliser que les variables de ces deux boîtes imbriquées : les variables locales `t1` et `t2` et les variables d'instance `r1` et `r2`.

Notons que la méthode `racines` peut aussi utiliser les paramètres formels `a`, `b` et `c`, mais ce ne sont pas des variables au même sens que les précédentes, puisque Java interdit de faire des affectations sur les paramètres formels.

Nous verrons au prochain chapitre que, sous certaines conditions, des variables autres que les précédentes et appartenant à des objets qui seraient passés en paramètres réels à une méthode ou encore qui seraient contenus dans les variables d'instance, peuvent devenir accessible à la méthode. Cet accès est donné de manière très restrictive et volontaire par le programmeur dans certain cas où cela simplifie le programme. Ces moyens sont utilisés avec parcimonie pour éviter de violer trop profondément les principes d'encapsulation dont nous avons déjà discutés longuement les avantages.

2.4.4 Assertions en Java

Depuis la version 1.4 de Java, un énoncé d'assertion a été ajouté au langage. Le patron syntaxique (le plus simple) pour écrire une assertion est le suivant :

```
assert <condition> ;
```

dont la signification est : lorsque l'énoncé est exécuté, la condition (une expression à réultat booléen) est évaluée ; si le résultat de la condition est vrai, l'exécution continue normalement, sinon une erreur est signalée et le programme s'arrête.

Exemple 10 Dans un objet représentant un cercle, on peut exprimer la condition selon laquelle le rayon est positif par l'assertion :

```
assert rayon >= 0.0 ;
```

□

Les utilisations de l'énoncé `assert` sont très nombreuses : assertions sur des points d'observation du programme, précondition en début de méthodes, postcondition en fin de méthode, invariants en fin de méthodes, etc. Ces utilisations seront développées plus loin dans ce document.

2.5 Exercices

2.5.1. En simplifiant les choses, on peut traduire un programme CaML en programme Java, il faut déterminer les types de variables globales du programme CaML (s'il y en a) et trouver une correspondance avec un type de Java pour en faire la déclaration dans une classe racine. Ensuite, on doit traduire chacune des fonctions CaML en méthode Java, en appliquant la démarche suivante :

1. trouver les types Java correspndant aux types des arguments et au type du résultat de la fonction CaML pour en tirer la déclaration de la méthode Java,
2. trouver les opérateurs Java correspondants aux fonctions prédéfinies de CaML et traduire les expressions CaML vers des expressions équivalentes en Java, et
3. trouver des structures de contrôle Java[12] correspondant aux structures de contrôle utilisées dans la fonction CaML pour reconstruire le corps de la méthode en Java en utilisant ultimement un énoncé `return` pour retourner le résultat.

Enfin, il faut introduire une méthode `main` qui va remplacer la boucle lire-évaluer-écrire de CaML par des appels explicites aux méthodes.

(a) Prenez le programme CaML suivant et traduisez-le en Java :

```
let sommeUnAn = fun n -> (n * (n + 1))/2 ;;
```

(b) Faites de même avec le programme suivant :

```
let pi = 3.14159 ;;
let calculeVolume = fun r -> (4.0 *. pi *. r *. r *. r) /. 3.0 ;;
let calculeSurface = fun r -> 4.0 *. pi *. r *. r ;;
```

2.5.2. Considérez la construction dans une application d'objets représentant des formes géométriques courantes : cercle, carré, ellipse, triangle, rectangle, pentagone, etc. Proposez des représentations pour chacun de ces objets. Dans certains cas, il peut y avoir plusieurs alternatives possibles ; discutez alors l'intérêt des unes par rapport aux autres. Quels seraient les invariants de structure dans vos représentations ?

[12] Les structures de contrôle Java seront introduites au chapitre 4.

Chapitre 3

Classes et objets

Dans ce chapitre, nous allons aborder le paradigme objet dans toute sa puissance. Jusqu'à maintenant, nous avons vu l'approche objet comme l'aboutissement ultime d'une évolution de la programmation impérative vers plus de puissance expressive, de structuration, d'encapsulation, de clarté, bref vers de meilleurs programmes. Dans le présent chapitre, nous allons approfondir l'approche objet et cette nouvelle façon de voir les calculs et les programmes qui aident le programmeur dans sa tâche de conception et de réalisation.

3.1 L'objet

Au delà de la métaphore, utile certes mais pas une panacée, essayons maintenant de concrétiser un peu ce qu'on entend par objet pour aller vers la programmation par objets dans un langage de programmation précis : Java.

3.1.1 Les principes et leur illustration

Tout objet procède de trois aspects : (1) ce qu'il est, par son *état*, (2) ce qu'il fait, par son *comportement*, et (3) comment il est appelé, par son *identité*. Nous avons vu l'évolution qui a mené des données et procédures impératives à l'objet comme entité unique unissant traitements et données. État, et donc données, de même que comportement ou savoir-faire, et donc traitements, sont donc deux éléments essentiels à la notion d'objet. L'autre concept important concernant l'objet est son identité. Chaque objet a une identité propre qui le distingue de tous les autres objets. Cette identité est une propriété indélibile de l'objet ; elle permet donc de suivre l'évolution d'un objet entre tous les états qu'il peut prendre au cours d'un calcul. En résumé, on peut donc étendre notre vision de l'objet à l'équation :

$$\text{objet} = \text{état (variables)} + \text{comportement (méthodes)} + \text{identité (identificateur d'objet)}$$

L'objet est un mécanisme d'encapsulation, dans la mesure où son état, qui est constitué par les valeurs de ses variables d'instance, est invisible de l'extérieur.[1] Pour utiliser un objet,

[1] Comme nous le verrons dans la partir éléments de savoir-faire, il est possible de rendre visible une variable d'instance d'un objet de l'extérieur, et donc de permettre l'accès à sa valeur depuis l'extérieur de l'objet. Ce n'est pas par contre le cas le plus normal. Par défaut, une variable d'instance d'un objet n'est pas visible et n'est pas accessible depuis l'extérieur de l'objet.

1. Tout objet a un identificateur d'objet unique et permanent qui sert à lui envoyer des messages.
2. Un objet possède des données, ses variables d'instance ; l'état d'un objet observé en cours d'exécution du programme est constitué de l'ensemble des valeurs de ses variables d'instance.
3. Un objet possède un comportement ou est doté de savoir-faire : ses méthodes. Une méthode est constituée d'un nom, d'un type de résultat, d'une liste de paramètres formels typés et d'un corps qui explicite ce que fait cette méthode lors de son activation.
4. La seule façon de réaliser une opération sur un objet consiste à appeler l'une de ses méthodes par un envoi de message. Pour envoyer un message à un objet, il faut connaître et utiliser l'identificateur de cet objet. Un message est constitué d'un nom de méthode et d'une liste de paramètres réels.
5. L'état d'un objet est rémanent entre les envois de messages.
6. Les variables d'instances sont généralement privées alors que les méthodes sont généralement publiques. L'objet a complet contrôle sur ce qu'il veut rendre privé ou public.

FIG. 3.1 – Principes de base de l'objet

il faut donc nécessairement passer par son comportement et s'adresser à lui par envoi de message grâce à son identificateur d'objet. Une frontière opaque entoure l'objet et masque à l'observateur extérieur ses variables d'instances, qui sont alors dites *privées*, tout en ne laissant voir que les méthodes, qui sont alors dites *publiques*.

La figure 3.1 regroupe les grands principes à la base de l'entité informatique objet, telle qu'elle est entendue dans tous les langages à objets aujourd'hui.

Les chaînes de caractères immutables et mutables

Pour concrétiser un peu les principes de base des objets, nous allons utiliser un exemple très simple : celui des objets chaînes de caractères tels qu'ils se présentent en Java. En Java, les constantes chaînes de caractères, comme par exemple `"Bonjour le monde"`, sont représentées par des objets créés à partir de la classe prédéfinie `String`.[2] Les objets de la classe `String` sont immutables, c'est-à-dire que leur état (la collection ordonnée des caractères qui forment la chaîne) ne peut pas être modifié. Une seconde classe prédéfinie, appelée `StringBuffer`, permet de créer des objets chaînes de caractères mutables, c'est-à-dire dont on peut modifier le contenu à volonté.

La classe `String` définit un certain nombre de méthodes, c'est-à-dire les comportements, pour les chaînes de caractères immutables. Par exemple, `String` définit la méthode `toUpperCase`. Pour obtenir d'un objet de la classe `String` qu'il exécute cette opération, il faut lui envoyer un message, comme il se doit selon le quatrième principe (figure 3.1). Le message `toUpperCase()` envoyé à une chaîne de caractères retourne une nouvelle chaîne de caractères équivalente au receveur sauf que tous les caractères en minuscules auront été remplacés par leur caractère

[2]La notion de classe sera introduite dans la prochaine section. Considérons ici qu'il s'agit d'une espèce de type qui permet de connaître quels messages peuvent être envoyés à ces objets.

correspondant en majuscule. Cette méthode fait partie des savoir-faire des objets de la classe `String`, tel que prévu par le troisième principe (figure 3.1). L'envoi de message en Java utilise une notation pointée. L'expression :

```
"ubs".toUpperCase()
```

retourne comme résultat la chaîne de caractères `"UBS"`, ce que nous allons illustrer de la manière suivante :

`"ubs".toUpperCase()` ⇒ `"UBS"`

Java exécute cette expression de la manière suivante. La constante `"ubs"` entraîne la création d'un objet de la classe `String`. Cet objet se voit attribuer un identificateur d'objet unique et permanent, selon le premier principe évoqué à la figure 3.1. Cet identificateur est ensuite utilisé par l'expression d'envoi de message `"ubs".toUpperCase()` qui entraîne l'exécution de la méthode `toUpperCase` de la classe `String` sur l'objet créé à partir de `"ubs"`. L'exécution de cette méthode crée une nouvelle chaîne de caractères, un objet avec son propre identificateur d'objet, qui représente la chaîne `"UBS"`.

Un message s'adresse à un objet qui devient en quelque sorte un paramètre sur lequel va travailler la méthode. Outre ce destinataire, tout comme les procédures, les méthodes peuvent avoir des paramètres. Par exemple, la méthode `concat` de la classe `String` prend une chaîne de caractères en argument et retourne une nouvelle chaîne de caractères qui est la concaténation de la séquence de caractères de la chaîne réceptrice avec la séquence de caractères de la chaîne argument. Par exemple,

`"Université".concat(" de Bretagne sud")` ⇒ `"Université de Bretagne sud"`

La classe `StringBuffer`, pour sa part, permet de créer à partir d'une chaîne de caractères immutable de la classe `String` un objet chaîne de caractères mutable, c'est-à-dire dont l'état peut être modifié. Par exemple, l'énoncé :

```
StringBuffer s1 = new StringBuffer("Université") ;
```

déclare une variable `s1` de type `StringBuffer`, puis l'expression `new StringBuffer("Université")` crée un objet de la classe `StringBuffer` à partir de la chaîne `"Université"` et retourne son identificateur d'objet unique qui est rangé dans la variable `s1` par affectation.

Si maintenant on exécute l'énoncé :

```
StringBuffer s2 = s1 ;
```

l'affectation réalisée consiste à copier dans `s2` l'identificateur d'objet qui a été préalablement rangé dans `s1`. Si on envoie le message `toString`, dont le rôle générique en Java coniste à retourner une représentation de l'objet receveur sous forme de chaîne de caractères, à l'objet dont l'identificateur est dans `s2`, on obtient la chaîne de départ :

`s2.toString()` ⇒ `"Université"`

De même, pour se convaincre que `s1` et `s2` réfèrent bien tous les deux au même objet, modifions l'objet référé par `s1` en lui envoyant le message `append` qui ajoute à la suite des caractères de la chaîne les caractères contenus dans la chaîne passée en argument :

```
s1.append(" de Bretagne sud") ;
```

Cette fois-ci, les deux références nous retournent la même chaîne de caractères :

`s1.toString()` ⇒ `"Université de Bretagne sud"`

`s2.toString()` ⇒ `"Université de Bretagne sud"`

De manière plus directe, on peut vérifier que deux variables contiennent bien le même identificateur d'objet en utilisant l'opérateur de comparaison '=='. Ici, l'expression `s1 == s2` retourne vrai. Par contre, si on crée un nouvel objet par :

```
StringBuffer s3 = new StringBuffer("Université de Bretagne sud");
```

bien que les objets référés par `s1` et `s3` contiennent des séquences de caractères identiques, l'expression `s1 == s3` retourne faux car les deux variables ne contiennent pas le même identificateur d'objet. En fait, il y a bel et bien deux objets différents, mais qui ont le même état. On constate donc que non seulement tout objet a un identificateur d'objet unique qui le distingue de tous les autres objets, selon le premier principe, mais qu'en plus l'identificateur d'objet est la seule chose qui permette à coup sûr de distinguer deux objets qui auraient par ailleurs le même état.[3]

Chaque objet possède son état propre. Si on modifie l'objet référé par `s3`

```
s3.append(".");
```

les deux objets ne sont plus identiques puisque maintenant la chaîne représentée par l'objet référé par `s3` est maintenant `"Université de Bretagne sud."` alors que celle référée par `s1` est toujours `"Université de Bretagne sud"` (sans le point final).

3.1.2 Création et utilisation des objets

Informatiquement, un objet peut être créé puis recevoir des messages auxquels il répond en activant les méthodes correspondantes. Mais que veut dire création d'un objet ? Quelle est la forme générale d'un envoi de message ?

Création d'un objet

Nous savons, depuis le chapitre précédent, qu'une variable n'est qu'une façon de désigner un espace mémoire qui va contenir sa valeur. Pour créer un objet, il faut donc allouer à cet objet une séquence de mots mémoire qui va permettre de conserver les valeurs de toutes ses variables. Tout objet répondant à un invariant structurel, il est impératif que dès la création de l'objet, ses variables reçoivent des valeurs telles que l'invariant soit observé préalablement à toute utilisation de cet objet. Parce que ces valeurs seront les valeurs initiales des variables, l'opération qui consiste à attribuer aux variables de l'objet des valeurs respectant l'invariant immédiatement après la création est appelée *initialisation*. En résumé, on peut dire que :

$$\text{Création d'un objet} = \text{allocation d'espace mémoire} + \text{initialisation}$$

C'est lors de la création de l'objet que lui est attribué son identificateur d'objet unique. Le résultat de l'opération de création est cet identificateur d'objet, qui devient une valeur manipulée comme n'importe quelle autre valeur dans le langage (entier, caractère, etc.). On peut affecter cet identificateur d'objet à une variable, le passer en paramètre à une méthode ou encore le retourner comme résultat d'une méthode.

[3] La classe `Object` de Java déclare une méthode `equals` à un paramètre qui définit aussi une égalité de référence. Cependant, certaines classes, comme `String` rédéfinissent cette méthode pour faire une égalité des valeurs. Ainsi, `"Université".concat(" de Bretagne sud").equals("Université de Bretagne sud")` ⇒ `true`. Ce n'est pas le cas de la classe `StringBuffer`, malheureusement.

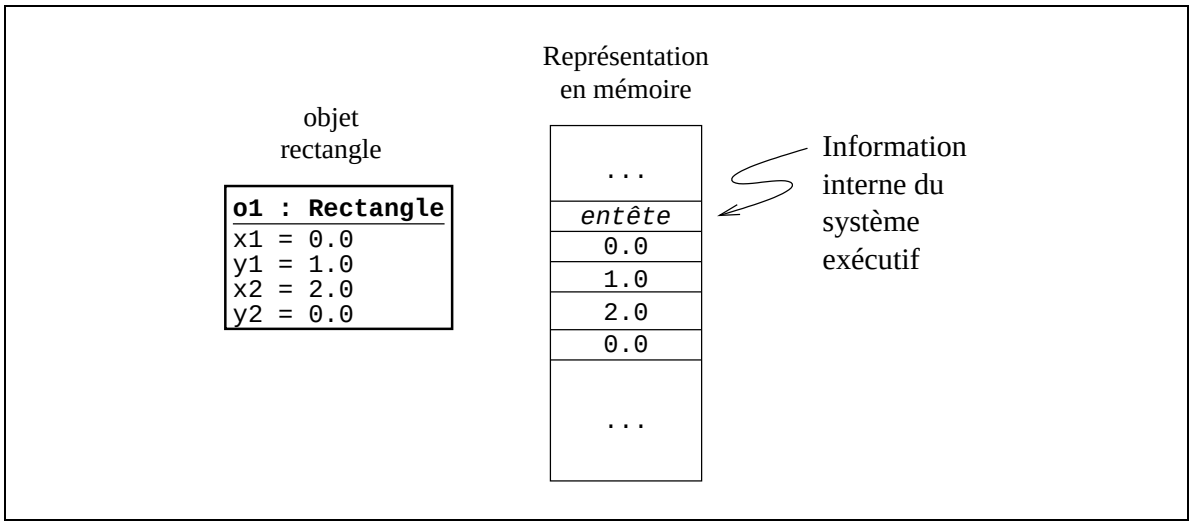


FIG. 3.2 – Notation visuelle de l'objet rectangle et sa représentation en mémoire

Exemple 11 Prenons le cas d'un objet rectangle. Cet objet possède quatre variables réelles. Pour créer un rectangle, le système va d'abord allouer quatre mots mémoire pour contenir les valeurs des quatre variables. Ces quatre mots vont recevoir des valeurs initiales respectant l'invariant du rectangle, par exemple le coin supérieur gauche (0, 1) et le coin inférieur droit (1, 0). Enfin, le système va attribuer un identificateur d'objet à cet objet, disons `o1`.[4] Cet identificateur est retourné comme résultat de la création. La figure 3.2 donne une représentation visuelle des objets et illustre la représentation en mémoire de cet objet rectangle. □

La forme précise de l'opération de création dépend du type de langage de programmation considéré. Pour Java, elle sera présentée à la section suivante (§3.2).

Envoi de message

L'objet créé, on peut lui envoyer des messages. Comme nous l'avons fait remarquer, l'envoi de message est à l'approche objet ce que l'appel de procédure est à la programmation impérative et l'appel de fonction à la programmation fonctionnelle. De la même façon que lors d'un appel de procédure ou d'un appel de fonction, un message contient le nom de la méthode appelée, et la liste des arguments qui seront passés en paramètre à la méthode. Les arguments doivent correspondre en nombre et en type avec les paramètres formels de la méthode de l'objet. La méthode est activée, exécutée, et elle peut retourner un résultat. Nous reviendrons au prochain chapitre (§4.5) sur les effets précis de l'appel de méthode dans un programme.

La forme générale de l'envoi de message en Java utilise une notation pointée pour relier l'expression s'évaluant au receveur et le message comme tel, qui est lui-même constitué du nom de la méthode à activer et des arguments de l'appel :

```
<receveur>.<nom-méthode>(<argument1>, <argument2>, ...)
```

[4] L'identificateur est un simple valeur binaire, comme toute autre valeur sur un ordinateur. La représentation précise de cet identificateur d'objet est cachée à l'utilisateur, comme il se doit pour un type de données primitif. Nous utilisons ici des noms compréhensibles pour faciliter la lecture du document.

La partie `<receveur>` est une expression dont le résultat doit être un identificateur d'objet. Cette partie peut donc être n'importe quelle expression dont le résultat est un identificateur d'objet ; le cas le plus courant est d'utiliser une variable ou un argument de méthodes dont le contenu est un identificateur d'objet. Le mot-clé `this` peut aussi être utilisé comme receveur et il désigne l'objet au sein duquel l'envoi de message est fait ; cela permet donc à un objet de s'envoyer un message à lui-même, et donc aussi de faire des envois de messages récursifs. Par défaut, un envoi de message sans receveur explicite a pour receveur implicite `this`.

La partie `<nom-méthode>` est un nom de méthode que doit connaître le receveur. La liste d'arguments `<argument1>, <argument2>, ...` est tout à fait classique : c'est une liste d'-expressions dont les résultats vont servir de paramètres réels à la méthode, comme pour les fonctions et procédures dans les autres langages.

La différence importante entre l'envoi de message et l'appel de fonction ou de procédure vient de ce que le choix de la méthode utilisée pour répondre à un envoi de message dépend de l'objet qui reçoit le message. En effet, très souvent, plusieurs objets différents peuvent savoir répondre au même message, mais de manière différente. C'est ce que l'on appelle le *polymorphisme*.

> **polymorphisme :** *nom masc.* — de *poly-*, et *morphisme*. Caractère de ce qui est polymorphe.
>
> **polymorphe :** *adj.* — de *poly-*, et *morphe*. Qui peut se présenter sous différentes formes.

Une opération est polymorphe si elle peut s'appliquer à des objets de différentes formes. Cette situation arrive très souvent en pratique dans les programmes à objets et fait partie des avantages du paradigme. Nous l'illustrons par une exemple bien connu.

Exemple 12 Considérons une application qui doit tracer des figures géométriques à l'écran. Il va de soi que l'opération de traçage d'un rectangle n'est pas la même que celle d'un triangle ou que celle d'un cercle. Pour exprimer ces différences dans un traitement, l'approche classique va consister à vérifier le type de figure à tracer et à appliquer le traitement spécifique à chaque cas :

```
si la figure à tracer est un rectangle alors
  exécuter l'opération de traçage de rectangles
sinon si la figure à tracer est un triangle alors
  exécuter l'opération de traçage de triangles
sinon si la figure à tracer est un cercle alors
  exécuter l'opération de traçage de cercles
```

Que se passe-t-il si on ajoute un nouveau type de figure ? Il va falloir modifier ce traitement pour y introduire un nouveau test sur le type de figure et introduire le traitement approprié. En pratique, cela peut se répéter à beaucoup d'endroits dans un programme. Chaque ajout ou retrait peut donc entraîner de nombreuses modifications, sujettes à erreur et oubli. □

Le besoin qui est exprimé ici est celui d'adapter la manière de faire une opération en fonction de la valeur sur laquelle l'opération est faite. L'approche objet propose une solution élégante à ce problème en dotant l'objet de sa capacité de traitement propre. C'est de ce principe que résulte le polymorphisme, c'est-à-dire le fait qu'une même opération puisse prendre des

formes différentes, ici en fonction de l'objet sur lequel elle s'applique. L'objet figure peut avoir sa propre méthode pour le traçage à l'écran. C'est lors de l'envoi de message que l'objet receveur va relier le message à sa méthode. Ainsi, selon que le message de traçage est envoyé à un rectangle, un cercle ou un triangle, c'est la méthode de traçage du rectangle, du cercle ou du triangle qui sera appliquée. Si une nouvelle forme est ajoutée, il suffit de lui définir une méthode de traçage, et alors, partout où elle pourra être utilisé dans un programme comme figure, l'envoi du message de traçage activera sa propre méthode de traçage.

L'objectif est de rendre les programmes plus facilement modifiables en les rendant indépendants des types de valeurs sur lesquels ils opérent et de l'ensemble de types effectivement utilisés. Pourquoi en effet la partie de programme qui demande le traçage des objets figure devrait-elle connaître les types de figures existantes ? Dans l'approche objet, elle n'a pas à les connaître. Et partant de là, cette partie de programme devient alors indépendante des types de figures et donc moins sujette à des modifications ultérieures.

Le principe derrière l'envoi de message est donc qu'une opération est réalisée en demandant à un objet de la faire par envoi d'un message, et l'objet sait répondre à cet envoi de message en exécutant le traitement approprié à l'opération qui lui est demandé. La résolution d'un envoi de message demande donc de rechercher la méthode à appliquer en fonction de l'objet receveur, puis de l'appliquer pour répondre au message. En résumé, on peut dire que :

$$\text{Envoi de message} = \text{rechercher} + \text{appliquer}$$

Exemple 13 Retour sur les figures géométriques. Selon l'approche objet, on peut définir une opération `affiche` sans argument que tout objet représentant une figure géométrique sait réaliser. Pour un objet, savoir réaliser une opération signifie avoir une méthode qui correspond à cette opération. Sur un rectangle r, un triangle t et un cercle c, on aura :

```
r.affiche()
t.affiche()
c.affiche()
```

Et si la variable `forme` contient un objet d'un des types de formes géométriques, on peut l'afficher en faisant :

```
forme.affiche()
```

C'est en fonction du contenu de la variable `forme`, et donc de l'objet auquel elle réfère, que le choix de la méthode sera fait entre l'affichage dun rectangle, d'un triangle ou d'un cercle. Si l'objet est un cercle, c'est la méthode des cercles qui est appelée, alors que si c'est un triangle, c'est la méthode des triangles qui le sera. □

Le fait que la recherche de la méthode appropriée pour répondre à un message se fait lors de l'exécution du programme plutôt que lors de la compilation mène à ce que l'on appelle la *liaison tardive ou retardée* du message à la méthode utilisée pour y répondre. Dans les langages classiques, le lien entre un appel de procédure par exemple et la procédure qui sera exécutée est fait une fois pour toute lors de la compilation du programme. En programmation par objets, la liaison entre un message et la méthode appliquée se fait à l'exécution du programme.

Pour exploiter pleinement le polymorphisme dans un langage typé comme Java, il faut utiliser le concept d'*héritage*, que nous abordons plus loin.

3.1.3 Architectures logicielles et coopération entre objets

Un programme est un artéfact et à ce titre il possède une architecture qui définit son organisation interne. D'autres artéfacts courants possèdent également une architecture. Une maison peut être de plein-pied ou à étages, son organisation peut être à aires ouvertes, comme un «*loft*», ou encore à pièces fermées. Une automobile peut être à traction avant ou arrière, berline ou «*break*», etc. L'architecture dicte l'aspect général d'un artéfact. Chaque cas particulier sera ensuite organisé, assemblé, selon un plan pour répondre à une demande précise.

L'architecture n'est pas une science exacte, mais plutôt un ensemble de principes qui ont démontré leur efficacité pour répondre à des besoins. Trouver une bonne architecture est donc autant affaire de connaissance que d'expérience. L'enseignement de la programmation est donc, comme l'architecture, fondé autant sur la connaissance de langages de programmation que sur l'étude de programmes exhibant des architectures connues pour efficaces. Comment cela se reflète-t-il en programmation par objets ?

Un calcul en approche objets est réalisé par la coopération d'objets qui s'échangent des messages pour se confier mutuellement des tâches. L'objet receveur d'un message réalise alors la tâche correspondante en exécutant la méthode appropriée, puis retourne le résultat éventuel à l'expéditeur du message. Lors de l'exécution de la méthode, le receveur peut à son tour envoyer un message à un autre objet pour lui confier une partie de la tâche, selon le principe traditionnel de découpage descendant, sur lequel nous revenons dans un chapitre subséquent.

Il est cependant important de se rappeler que pour envoyer un message à un autre objet, l'expéditeur doit connaître l'identificateur d'objet du receveur. Par analogie, les échanges de messages en approche objet s'apparentent plus à des appels téléphoniques qu'à des émissiosn radio. De la même manière qu'il faut connaître le numéro de téléphone de quelqu'un pour lui téléphoner, il faut connaître l'identificateur d'objet pour lui envoyer un message. Connaître peut signifier ici deux choses : soit l'identificateur d'objet est contenu dans une variable d'instance, soit l'objet l'a reçu en paramètre du message reçu. Dans les deux cas, le code de la méthode exécutée va pouvoir utiliser l'identificateur d'objet pour envoyer un message.

Les liens qui se créent entre objets qui connaissent les identificateurs d'objets des uns et des autres sont des liens dits d'*agrégation*. Les liens d'agrégation entre objets tissent un réseau, ou une espèce de toile qui dictent comment les tâches et les sous-tâches pourront être propagées d'un objet à un autre par envois de message successifs. En théorie toute topologie est possible pour ce réseau, mais en pratique des formes reviennent avec régularité dans les programmes. Ces patrons qui se répètent reflètent qu'en réalité, un petit nombre (relativement) de *schémas de coopération* précis permettent de résoudre beaucoup de problèmes courants.

Exemple 14 Le lien de *client–fournisseur* est probablement le schéma de coopération le plus courant entre deux objets. L'objet `A`, appelé le *client*, utilise les services d'un objet `B`, appelé le *fournisseur*, pour réaliser une tâche donnée. Par exemple, un objet représentant un polynôme peut être le client d'un objet représentant une imprimante graphique capable de tracer la courbe du polynôme. Ici, les envois de message se font toujours dans le sens du client vers le fournisseur, et jamais dans l'autre sens. □

Exemple 15 Le lien *constitué-de* qui relie le tout à ses parties, comme par exemple une voiture à sa carosserie, son moteur, etc., est aussi un schéma de coopération courant dans les programmes. Un message, par exemple pour démarrer la voiture, va être répercuté par l'objet automobile vers les parties mises en œuvre dans l'opération de démarrage. À la différence du lien *client-fournisseur*, il est presque toujours interdit qu'un objet soit une partie de plusieurs

touts. Comment en effet un unique moteur pourrait être une partie de plusieurs voitures ? A contrario, dans le lien *client-fournisseur*, il est tout à fait possible qu'un objet fournisseur rende des services à plusieurs objets clients. □

Un schéma de coopération est un patron de liens entre quelques objets qui sont utilisés selon une certaine regularité par ces derniers. Il traite d'organisation locale, un peu comme dans le plan d'un maison un certains nombre de pièces vont partager un couloir et permettre à l'occupant de se rendre de la chambre aux toilettes par exemple.

Plus globalement, l'assemblage de l'ensemble des objets d'un programme reflète l'*architecture* de ce programme. Encore une fois, cette architecture ne doit rien au hasard. Comme toute autre organisation, par exemple une banque ou une équipe de football, elle reflète une structure de responsabilités et des flux d'informations conçus pour résoudre le problème posé.

Exemple 16 Peut-être l'exemple le plus connu d'architecture, au sens large, de systèmes de traitement de l'information en général est l'architecture dite *client-serveur*. Selon cette architecture, le système est organisé sous la forme d'un ordinateur central chargé des traitements et d'un certains nombre de postes clients. Le rôle des postes clients consiste simplement à entrer les requêtes et les données puis à présenter à l'opérateur les résultats. La plupart des systèmes de réservation, comme ceux de billets d'avion, fonctionnent selon cette architecture. □

Un exemple du monde objet : l'architecture MVC

L'exemple le plus connu d'architecture en programmation par objets est l'architecture d'interaction *MVC*, pour modèle-vue-contrôleur. Considérez un automate bancaire. Il est doté d'un écran, de boutons et d'un clavier (essentiellement numérique, plus quelques touches de sélection ou de validation). C'est la *vue* de la machine. L'objet vue dans l'achitecture MVC est chargé de présenter les informations à l'écran.

À l'autre extrémité, le système informatique de la banque comporte un objet compte qui représente les informations concernant votre compte bancaire et les opérations que vous pouvez réaliser. C'est le *modèle*. L'objet modèle est chargé de traiter les transactions. Il est consulté par la vue lorsque celle-ci doit afficher des informations sur votre compte. Lorsqu'une information du modèle affichée par la vue est modifiée, le modèle en informe la vue pour que celle-ci remette à jour l'information présentée à l'écran. Il se crée donc un lien de consultation de la vue vers le modèle, mais aussi un lien de dépendance du modèle vers la vue. Entre la vue et le modèle est placé un objet qui est activé lorsque vous pressez un bouton ou introduisez votre carte. C'est le *contrôleur*. Le rôle de l'objet contrôleur est de relayer les demandes faites via les boutons en termes de transactions à réaliser sur le modèle.

L'architecture MVC attribue à chacun de ces objets des rôles précis et définit les liens de consultation et de dépendance (voir figure 3.3) permettant par exemple d'afficher les informations adéquates à l'écran en fonction des commandes données et des résultats de traitement.

Voici un scénario simplifié d'exécution selon cette architecture (un vrai automate bancaire est en réalité bien plus complexe). L'utilisateur insère sa carte dans l'automate, ce qui génère un événement qui sera envoyé sous la forme d'un message à l'objet contrôleur. Le contrôleur exécute la méthode correspondant à l'événement insertion de carte en envoyant à son tour un message à l'objet vue pour que s'affiche un écran de saisie du code secret de la carte. Lorsque l'utilisateur valide le code entré, un retour est fait au contrôleur sous la forme code secret. Le contrôleur envoie alors un message au modèle pour voir si le code secret est bon. Si oui, la vue va afficher un écran de saisie de la transaction à réaliser, avec des informations puisées

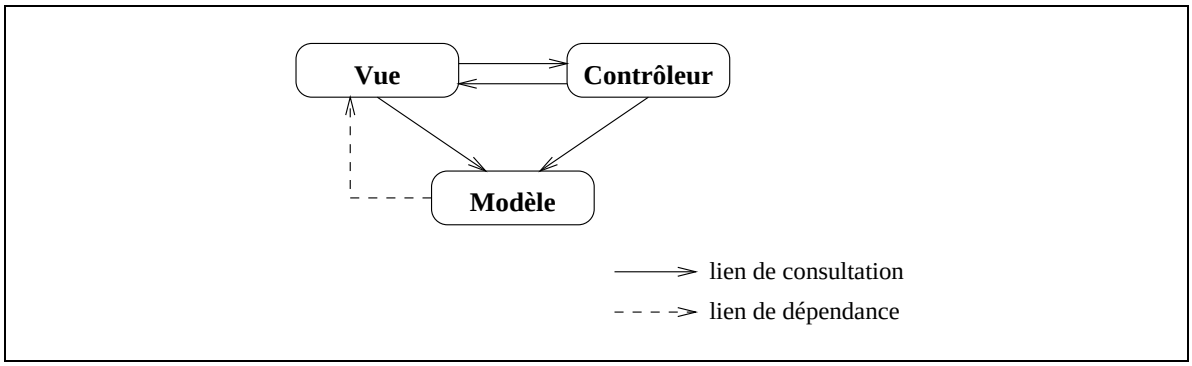


FIG. 3.3 – Architecture MVC (modèle-vue-contrôleur).

auprès du modèle. Si les informations venant du modèle affichées à l'écran sont modifées, le lien de dépendance entre modèle et vue fera en sorte que la vue sera notifiée du changement pour rafraîchir son affichage.

3.2 La classe

Nous l'avons dit, l'objet est un ensemble de données et d'opérations, dont la conception est centrée sur le comportement et fondée sur la notion de responsabilité. Dans les langages comme Java, les objets sont décrits par des classes, une notion qui est liée (mais pas identique) à celle des types que nous avons déjà vue.

3.2.1 Classe comme descripteur d'objets

Si on se reporte à la discussion sur le typage de la section 2.2, données plus opérations ressemblent fort à ce que nous avons appelé type. Rappelons-nous :

$$\text{Type} = \text{valeurs} + \text{opérations} + \text{représentation/encodage}$$

Parler d'objets amène donc naturellement à observer la similitude entre les objets et donc d'une notion de type qui regroupe l'ensemble des objets ayant le même ensemble de données et les mêmes opérations. Dans les langages à objets, c'est la notion de *classe* qui joue en quelque sorte le rôle de type. Il existe plusieurs définitions de la classe, selon le point de vue duquel on se place. Dans le droit fil de la notion de type, on peut proposer :

> **classe :** *nom. fém. – lat. classis.* Ensemble d'individus ou d'objets qui ont des caractères communs.

Cette définition ensembliste peut être interprétée de deux façons. Selon une première interprétation qui met l'emphase sur la notion d'ensemble défini en extension, elle correspond bien à la notion de type vu comme un ensemble de valeurs muni d'opérations. C'est ce que nous avons appelé un type abstrait. Nous y reviendrons. Une seconde interprétation de cette définition est plutôt fondée sur la notion d'ensemble défini en intention. La classe est alors vue comme une description des caractères communs de ses membres.

En programmation par objets, on considèrera qu'une classe regroupe les objets ayant la même structure, c'est-à-dire les mêmes variables, et le même comportement, c'est-à-dire les

1. Tout objet est instance d'une classe.
2. Toutes les instances d'une classe possède le même nombre, les mêmes noms et les mêmes types de variables d'instances.
3. Chaque objet possède ses propres valeurs pour ses variables d'instances et donc son propre état.
4. Toutes les instances d'une classe ont le même savoir-faire et peuvent donc répondre aux mêmes messages.

FIG. 3.4 – Principes des langages à classes

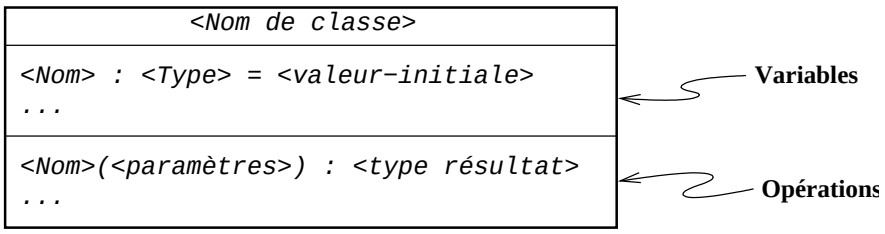


FIG. 3.5 – Notation visuelle pour les classes

mêmes méthodes. Ces objets seront dits *instances* de la classe qui les regroupe. Dans la plupart des langages à objets, la classe dans un programme se présente donc comme une définition en intention, c'est-à-dire la description des objets qui en seront instances. Elle décrit la représentation de ses objets en donnant la liste de leurs variables, et elle décrit les opérations applicables à ses objets en donnant la définition de ces opérations sous forme de méthodes.

Les objets appartenant à une même classe ne se distinguent donc que par leur identité, qui est unique, et éventuellement par les valeurs de leurs variables, chaque objet associant à ses variables ses propres valeurs. En effet, si on parle de l'ensemble des cercles, tous les cercles répondent à une forme bien précise, c'est-à-dire le lieu géométrique des points à distance égale, appelée rayon, d'un même point appelé centre ; mais chacun des cercles peut avoir sa propre taille, c'est-à-dire sa propre valeur de rayon, et son propre point central. La figure 3.4 résume les grands principes des langages à classes.

La figure 3.6 illustre cela en utilisant une notation visuelle pour les classes expliquée à la figure 3.5. La classe `Tortue`[5], décrit les objets tortues en établissant la liste des variables et les opérations connues de ces objets ; deux objets tortue `t1` et `t2` font partie de la classe `Tortue` et possèdent leurs propres valeurs pour les variables.

La classe répond à une autre question que nous avons esquivée jusqu'ici. Si d'un point de vue purement opérationnel la création d'un objet implique l'allocation de mémoire et l'initialisation

[5]Il s'agit d'une modélisation incomplète de la tortue Logo [Pap80], élément principal d'interaction de ce langage d'abord conçu pour apprendre aux enfants des notions de géométrie de manière ludique mais constructive. La tortue Logo apparaît à l'écran et répond aux ordres *avance* ou *recule* d'un certain nombre de pas ainsi que *gauche* et *droite* pour changer de direction. Elle porte également une plume qui, lorsqu'abaissée, laisse une trace à l'écran. L'élève peut ainsi apprendre des notions élémentaires de géométrie en dessinant des formes sur l'écran par une suite d'ordres donnés à la tortue.

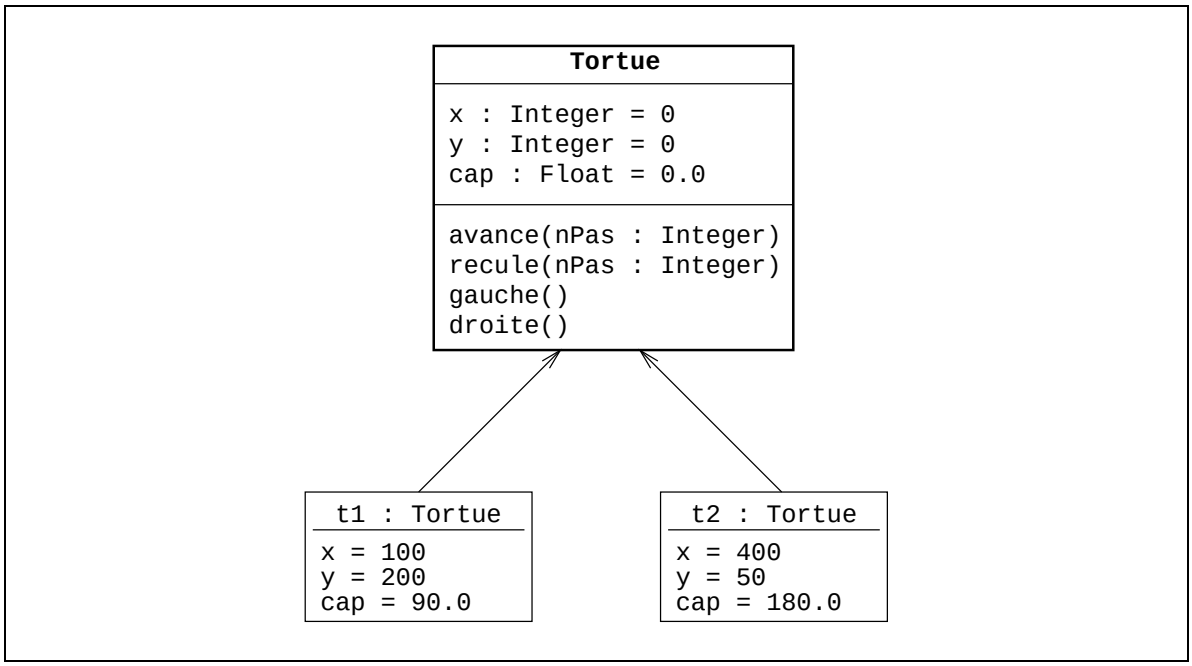


FIG. 3.6 – Exemple de la classe `Tortue` avec deux objets

des variables de l'objet, comment détermine-t-on ce que contient un objet ? Depuis les grecs, les philosophes s'interrogent pour savoir si l'essence précède ou non l'existence. Le concept de table, à l'image duquel toutes les instances réelles de tables seraient créées, est-il immannent ou n'est-il qu'induit à partir des objets tables existant dans la réalité ? La notion intensionnelle de classe va dans le sens de l'essence qui précède l'existence, alors que la notion extensionnelle va plutôt dans le sens inverse.

Les langages à objets ont répondu à leur manière à cette question en adoptant un principe de base[6] :

Tout objet est instance d'une classe !

La classe est donc la représentation du concept qui donne une description précise des objets, et c'est à partir de cette description que les instances sont créées. Plus spécifiquement, la classe est définie par :

1. Un nom unique qui permet de référencer le concept et d'en créer des instances.
2. La liste des variables, avec leurs types respectifs, que vont posséder tous les objets instance de cette classe.
3. La liste des opérations qui peuvent être appliquées aux objets de la classe, et qui vont donc former le savoir-faire commun à toutes ses instances.
4. La définition des méthodes qui mettent en œuvre les opérations d'une manière qui dépend de la représentation choisie pour les instances de la classe.

La liste des opérations pouvant être appliquées aux objets est constituée de l'ensemble des *signatures* des méthodes, c'est-à-dire leurs noms avec leur type de retour et leurs paramètres formels. Cette liste constitue ce que l'on appelle *l'interface* de l'objet, car il s'agit de la seule information nécessaire pour utiliser l'objet. De fait, la classe réalise l'encapsulation de l'objet

[6]De fait, la réalité des langages à objets n'est pas si simple, mais la plupart obéissent bien à ce principe.

dont nous avons parlé à la section précédente en faisant en sorte que seule l'interface de la classe soit visible de l'extérieur. Tous les autres éléments, que ce soient les variables d'instance ou le corps des méthodes, sont effectivement masqués aux autres parties du programme.

3.2.2 Instantiation

L'objet est l'entité dite dynamique, c'est-à-dire qui existe pendant l'exécution du programme et qui sert à faire des calculs. Pour créer un objet, on utilise la description donnée par la classe, comme s'il s'agissait d'un moule, ou d'un plan. En effet, l'approche objet considère la création d'un objet comme une opération réalisée à partir d'un plan, lequel est donné par sa classe. La création d'un objet à partir du plan fournit par la classe est appelée *instantiation*. Bien que chaque table partage une forme et un fonction communes à l'ensemble des tables, chacune possède ses caractéristiques propres. Elle a une taille, une couleur, un style, etc. En programmation par objets, les objets se distinguent par les valeurs de leurs variables d'instance, ou par leur état.

En Java, l'instantiation est réalisée par l'expression primitive `new`, dont la forme générale est :

```
new <nom-classe>(<liste-paramètres-initialisation>)
```

La partie `<nom-classe>` indique le nom de la classe à partir de laquelle l'objet doit être créé. La liste des paramètres d'initialisation sert à déterminer quelle méthode d'initialisation doit être appelée pour initialiser l'objet ; la sélection de la méthode d'initialisation se fait en fonction du nombre et du type des arguments. Deux méthodes d'initialisation d'une classe donnée ne peuvent donc pas avoir exactement le même nombre et les mêmes types d'arguments. La méthode d'initialisation déterminée, on l'active avec les paramètres donnés dans l'expression `new`. Le résultat de l'expression `new` est l'identificateur d'objet de l'objet créé. En tant que valeur manipulable comme n'importe quelle autre valeur en Java, son type est la classe de laquelle l'objet a été créé.

Exemple 17 Dans ses grandes lignes, la classe `Tortue` illustrée précédemment s'écrit de la manière suivante en Java :

```
class Tortue {
  int x ;
  int y ;
  double cap ;

  public Tortue() {
     x = 0 ;
     y = 0 ;
     cap = 0.0 ;
  }

  public Tortue(int x_init, int y_init, double cap_init) {
     x = x_init ;
     y = y_init ;
     cap = cap_init ;
  }

  public void avance(int nombrePas) { ... }
  public void recule(int nombrePas) { ... }
  public void gauche() { ... }
  public void droite() { ... }
}
```

Le nom de la classe apparaît après le mot-clé `class`. La définition de la classe comprend les déclarations des variables d'instance, une méthode du nom de la classe chargée d'initialiser les objets tortue lors de leur création, et la liste des méthodes applicables aux objets tortue. Voir la partie «Éléments de savoir-faire» (§3.5) pour plus de détails.

La création des deux objets tortue se fait de la façon suivante :

```
Tortue t1 = new Tortue() ;
Tortue t2 = new Tortue(100, 100, 45.0) ;
```

L'état donné pour `t1` et `t2` à la figure 3.6 est le résultat d'une séquence d'envoi de messages à ces deux objets après leur création. □

3.2.3 La méthode

La méthode est à la programmation par objets ce que la fonction est à la programmation fonctionnelle et la procédure à la programmation impérative. Elle donne le moyen de mettre en œuvre les savoir-faire des objets, tels que définis par la classe, sous la forme de bouts de code exécutable paramétré. Une méthode, comme nous l'avons vu, possède un nom, un type de résultat, une liste de paramètres formels typés et un corps qui explicite ce que fait cette méthode lors de son activation.

Comme pour une procédure, le corps d'une méthode est constitué d'une séquence d'énoncés que l'on va exécuter lors de son activation, activation qui est faite suite à un envoi de message à un objet. La séquence d'énoncés formant le corps d'une méthode contient des énoncés d'affectation, mais aussi d'autres types d'énoncés que nous le verrons au prochain chapitre (§4.5). Ces énoncés peuvent aussi contenir des expressions. Comme nous l'avons vu, l'affectation et les expressions peuvent faire référence à des variables, soit pour leur affecter de nouvelles valeurs, soit pour en utiliser la valeur courante.

Si l'objet et la classe fournissent un premier mécanisme d'encapsulation, en cachant les variables, les méthodes proposent aussi une forme d'abstraction et encapsulent l'information propre à chaque opération, c'est-à-dire les variables locales à la méthode et plus généralement le corps de la méthode.

Pour comprendre la frontière qui existe autour d'une méthode et donc quelles sont les variables et quelles sont les valeurs utilisables dans une méthode, il est important de comprendre le contexte dans lequel la méthode est exécutée. La méthode a toujours accès à ses variables locales et à ses paramètres, comme une procédure ou une fonction classiques. Par ailleurs, lorsqu'un message est envoyé à un objet, nous avons dit que cela active la méthode correspondant au message sur l'objet receveur. Le contexte dans lequel s'exécute la méthode est donc celui de l'objet receveur du message et sa frontière d'encapsulation délimite un second cercle de variables accessibles : les variables de l'objet receveur dont les noms et les types ont été déclarées par la classe de cet objet, et dont les valeurs sont détenues par l'objet receveur lui-même. Viennent ensuite en Java des notions de visibilité entre objets qui sont présentées à la section 3.5.

Exemple 18 Dans l'exemple de la classe `Tortue`, on peut écrire le code de la méthode `gauche` comme suit :

```
public void gauche() {
   cap = cap + 90.0 ;
```

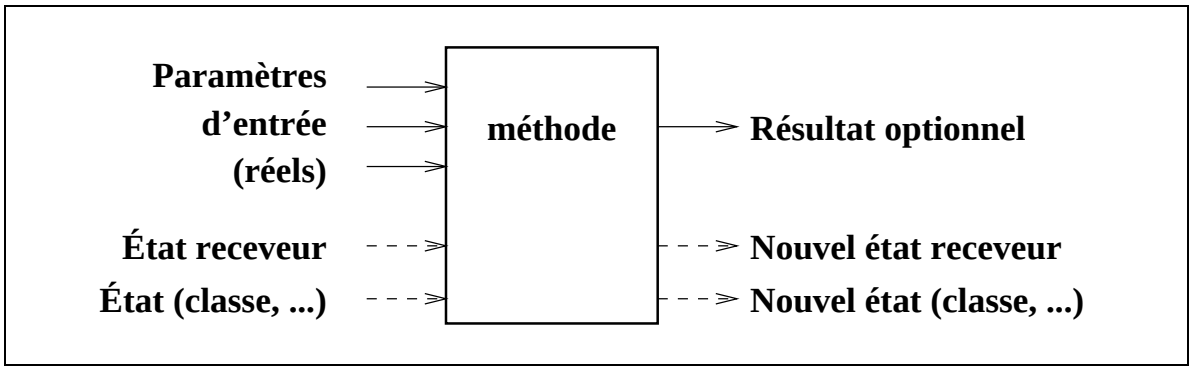


FIG. 3.7 – Méthode comme boîte noire

```
}
```

Cette méthode fait tourner la tortue à gauche, c'est-à-dire change son cap de 90 dégrés vers la gauche. L'envoi de message `t1.gauche()` va activer la méthode `gauche` dans le contexte de l'objet `t1`. La variable `cap` à laquelle la méthode fait référence est donc celle définie par la classe `Tortue`. On sait que chaque tortue possède son propre espace mémoire correspondant au nom de variable `cap`. L'espace mémoire visé par l'exécution de la méthode `gauche` est celui associé à la variable `cap` dans l'objet receveur du message `t1`. La référence à `cap` dans l'expression en partie droite de l'affectation va donc aller chercher la valeur de la variable `cap` dans l'objet `t1` et la référence en partie gauche va indiquer que l'affectation se fait sur la variable `cap` dans `t1`. □

Les méthodes en programmation par objets sont des procédures fonctionnelles. Elles peuvent modifier l'état de l'objet et retourner un résultat. Toutes les méthodes ne font pas nécessairement les deux.[7] Certaines ne font que modifier l'état de l'objet receveur. Ce sont alors des procédures «pures» que l'on nomme généralement des *modifieurs*. D'autres méthodes ne font que retourner un résultat. Ce sont alors des fonctions «pures» que l'on nomme généralement *accesseurs*. Cette distinction n'a d'intérêt pour l'instant que pour faciliter le raisonnement sur ce que fait un programme. Lorsqu'une méthode est reconnue comme un accesseur, il n'est pas nécessaire de chercher d'autres effets que le résultat qu'elle retourne.

Les méthodes peuvent aussi modifier l'état des variables visibles : variables de classes (statiques), variables accessibles (publiques, protégées, de package) des objets passés en paramètres ou contenus dans les variables visibles depuis la méthode, etc. Ces possibilités d'accès à des variables hors du contexte du receveur brisent l'encapsulation et rendent les programmes plus sensibles aux modifications. Il est donc fortement suggéré de les limiter le plus possible.

La figure 3.7 résume la vision de la méthode comme boîte noire. La méthode de type accesseur ne modifie pas l'état du receveur, ni aucun autre état d'ailleurs, mais retourne un résultat. La méthode de type modificateur a un effet sur l'état du receveur et peut aussi modifier des variables partagées via la classe (voir ci-bas). L'objectif de la programmation par objets est cependant de limiter à un minimum les modifications autres que celles affectant le receveur du message, car les modifications à des variables partagées entraînent généralement des difficultés de mise au point et de compréhension des programmes, comme nous l'avons vu précédemment.

[7] On comprendra qu'une méthode doit au minimum faire l'un ou l'autre. En effet, si une méthode ne modifie pas l'état de l'objet ou l'état du système en général et qu'elle ne retourne pas de résultat, alors son activation ne peut avoir d'effet mesurable et elle devient inutile.

3.2.4 Partage d'informations via la classe

Une autre façon de voir la classe consiste à la considérer comme un réservoir d'informations communes à l'ensemble de ses instances. Nous avons déjà vu que la description de la représentation, c'est-à-dire les noms et types des variables d'instance, de même que l'interface et les méthodes applicables aux instances sont définies dans la classe. On peut donc considérer que les instances partagent ces informations via leur classe.

Cette notion de réservoir d'informations partagées a amené plusieurs langages à objets à ajouter trois nouveaux éléments à la description des classes : la définition de constantes et de variables communes à l'ensemble des instances de la classe ainsi que la définition de méthodes dites de classe (sur lesquelles nous reviendrons plus loin).[8] Dans nombre de cas, l'ensemble des objets formant une classe contient des objets particuliers qu'il est utile de nommer d'une façon particulière et de rendre facilement accessibles à toutes les instances. En mathématique, par exemple, on distingue souvent les éléments neutres de certaines opérations ou les éléments absorbants.

Exemple 19 Nous verrons plus loin un exemple d'objets polynômes. On peut distinguer le polynôme 0 (celui dont tous les coefficients sont 0) et créer avec lui la constante `ZERO` connue (partagée) par tous les objets polynômes via la classe `Polynome`. □

Dans plusieurs cas également, un ensemble d'instances d'une classe peut avoir à partager une information commune, mais qui peut être modifiée en cours d'exécution. En déclarant une variable de classe contenant cet objet, l'ensemble des instances de la classe vont pouvoir accéder à cet objet et «voir» simultanément une éventuelle modification.

Exemple 20 Dans un jeu de billard, l'ensembles des boules de billard partagent la même table de billard sur laquelle elles évoluent. L'état de la table de billard peut changer (nombre de boules en jeu, joueurs, le joueur dont c'est le tour de jouer, etc.). En définissant dans la classe une variable de classe `table` contenant l'objet table de billard, toutes les instances de boules pourront y accéder facilement. □

Les méthodes de classe, quant à elles, sont utilisées le plus souvent pour fournir une bibliothèque de fonctions mises à la disposition des programmeurs. En Java par exemple, la classe `Math` définit ainsi un ensemble de fonctions mathématiques usuelles : `sin` (sinus), `cos` (cosinus), `tan` (tangente), ..., `pow` (mise à une puissance), `log` (logarithme), `sqrt` (racine carrée), etc.

3.2.5 Héritage

La classification comme méthode d'organisation hiérarchique de la connaissance fut certainement l'une des avancés importantes pour la compréhension d'organisations complexes. Linné est resté ainsi célébre pour sa classification des espèces naturelles. La programmation par objets se distingue des autres paradigmes par l'introduction de cette organisation hiérarchique pour structurer les programmes. En programmation par objets, une classe peut être

[8] Le spécialiste notera que nous nous confinons aux langages à objets où les classes ne sont pas des objets. La notion de méthodes de classe, tout comme celle de variables de classe, que nous utilisons font référence à des variables semi-globales dites statiques en Java ou les variables de *pool* en Smalltalk. Nous ne faisons pas référence aux variables d'instance et méthodes de la métaclasse.

définie comme étant une sous-classe d'une autre classe, héritant ainsi de toutes les caractéristiques de sa superclasse. C'est la relation d'*héritage*.

L'héritage a deux rôles principaux en programmations par objets : l'organisation des classes selon des taxonomies et la réutilisation de code. L'utilisation de taxonomies permet de rendre explicites les relations entre les différents objets d'une application, et donc entre leurs classes respectives. Dire qu'une classe `B` est une sous-classe de `A`, c'est affirmer clairement que les instances de `B` possèdent les mêmes caractéristiques que celles de `A`, peut-être en étant un peu plus spécialisées. Les chats sont des mammifères parce qu'ils possèdent les caractéristiques des mammifères plus des caractéristiques propres qui les différencient des autres mammifères.

L'utilisation de l'héritage pour la réutilisation est liée à la notion de *description différentielle*. Nous avons vu qu'une classe dans un programme à objets est une description de ces instances. Dans les langages à classes, l'héritage permet de définir une classe `B` comme sous-classe de `A` en se contentant de donner dans `B` que les caractéristiques propres aux instances de `B` et en héritant les caractéristiques de `A`. Lorsqu'on dit qu'un chat est un mammifère, il n'est pas nécessaire de répéter qu'une chatte possède des mammelles qui servent à nourrir ses petits ; cela fait partie des caractéristiques d'un mammifère qui s'applique donc aux chats.

En termes de programmation par objets, les variables et les méthodes définies par la superclasse deviennent de facto des variables et des méthodes de la sous-classe, auxquelles la sous-classe se contente d'ajouter ses éléments spécifiques. On peut donc considérer que la sous-classe réutilise des éléments préalablement définis dans la super-classe. Bien sûr, la sous-classe n'est pas forcée de réutiliser sans condition ce que la super-classe définit. Les langages à classes permettent tous à une sous-classe de redéfinir une méthode déjà définie dans la super-classe. Il lui suffit de donner une nouvelle définition à cette méthode qui sera alors utilisée lorsque ses instances recevront le message correspondant.

Parfois, on ne veut pas complètement remplacer la méthode déjà définie mais seulement la compléter. C'est possible, en appelant la méthode de la superclasse dans la méthode redéfinie dans la sous-classe. Il suffit pour cela d'écrire un envoi de message dont le receveur est le mot-clé `super`, qui désigne alors le receveur courant mais vu comme une instance de la superclasse pour le besoin de la recherche de la méthode répondant à ce message.

Exemple 21 Considérons une classe `A` et une sous-classe `B` définies de la manière suivante :

```
public class A {

    public String montre() {
      return "1A" ;
    }

    public String montreToi() {
      return this.montre() ;
    }
}

public class B extends A {

    public String montre() {
      return "2B" ;
    }

    public String evite() {
      return super.montre() ;
    }
}
```

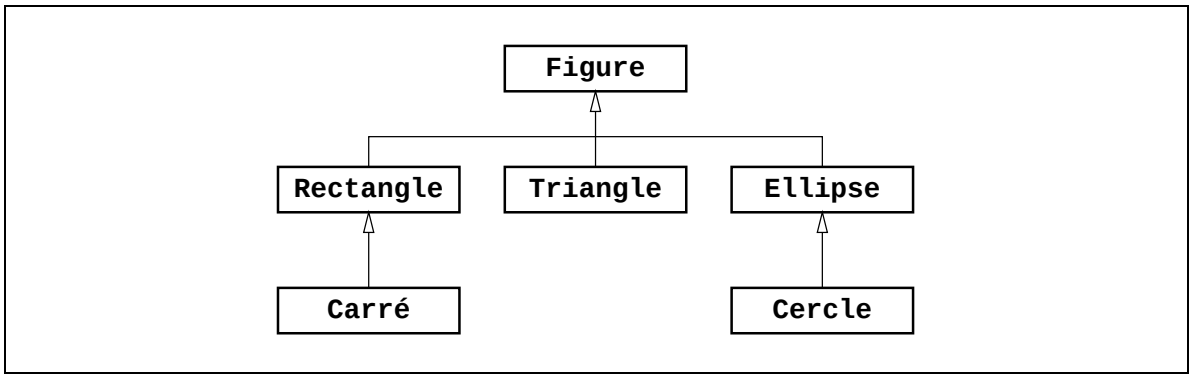


FIG. 3.8 – Taxonomie des figures géométriques

La séquence suivante illustre l'effet d'appels à `this` et à `super` :

`A a = new A();`	déclaration d'une variable `a` et affectation d'une instance de `A`.
`B b = new B();`	déclaration d'une variable `a` et affectation d'une instance de `A`.
`a.montre()` ⇒ `"1A"`	le message `montre` à une instance de `A` retourne la chaîne `"1A"`.
`b.montre()` ⇒ `"2B"`	le message `montre` à une instance de `B` retourne la chaîne `"2B"`.
`a.montreToi()` ⇒ `"1A"`	le message `montreToi` à une instance de `A` retourne la chaîne `"1A"` car la méthode `montreToi` appelle la méthode `montre` définie sur la classe `A`.
`b.montreToi()` ⇒ `"2B"`	le message `montreToi` à une instance de `B` retourne la chaîne `"2B"` car la méthode `montreToi` appelle la méthode `montre` définie sur la classe `B`.
`b.evite()` ⇒ `"1A"`	ici, le résultat est `"1A"` parce que l'appel à la méthode `montre` utilise le «receveur» `super`, ce qui lance la recherche de la méthode au niveau de la superclasse `A` de la classe `B` définissant la méthode `evite`, ce qui appelle donc la méthode `montre` de la classe `A`.

□

Exemple 22 Les classes représentant des figures géométriques peuvent être organisées selon la hiérarchie d'héritage de la figure 3.8. Au plus haut niveau, la classe `Figure` définit toutes les figures géométriques. Parmi les figures géométriques, les rectangles, les triangles et les ellipses sont définis par trois classes qui héritent directement de la classe `Figure`. La classe `Carré` est définit par héritage de la classe `Rectangle`, puisqu'un carré n'est qu'un rectangle particulier dont les quatre côtés sont égaux et parallèles deux à deux. De même, la classe `Cercle` est définie comme sous-classe de la classe `Ellipse` puisqu'un cercle est une ellipse dont les deux foyers coïncident.

En terme de réutilisation, la classe `Figure` définit les méthodes reconnues par toutes les figures géométriques. Une classe qui hérite de la classe `Figure` en hérite les méthodes et les variables d'instance. Il n'est donc pas nécessaire de réécrire toutes les méthodes dans les sous-classes, d'où la réutilisation du code existant. Il est par contre possible à une sous-classe de redéfinir une méthode. Selon le principe de polymorphisme présenté précédemment, les objets sont responsables de la sélection des méthodes à appliquer pour répondre à un message. Si la classe `Cercle` redéfinit la méthode `affiche` de la classe `Ellipse`, les instances de `Cercle` utiliseront cette nouvelle méthode pour répondre aux messages d'affichage. □

Les classes dans un programme sont donc organisées sous la forme d'une hiérarchie d'héritage. Dans la plupart des langages à objets, cette hiérarchie à une racine unique sous la forme d'une

classe le plus souvent appelée `Object`. La classe `Object` définit alors les comportements que doivent posséder tous les objets dans le langage. En Java, par exemple, la classe `Object` détermine que tout objet sait répondre au message `toString` en retournant une chaîne de caractères qui en donne une «forme» externe affichable sur un écran. Elle définit également la méthode booléenne `equals` qui permet de comparer deux objets pour leur égalité.

Héritage, polymorphisme et principe de substitution

L'héritage joue un rôle important vis-à-vis du polymorphisme, c'est-à-dire la possibilité pour une opération de prendre différentes formes selon les objets sur lesquels elles s'appliquent. Le fait de définir une méthode dans une super-classe implique que toutes ses sous-classes et les sous-classes de ces sous-classes vont hériter de cette méthode. On dit aussi que ces sous-classes doivent se *conformer* à l'interface de la super-classe dans le sens où leurs instances devront savoir répondre aux messages destinés à activer cette méthode. L'héritage assure donc le fait que tous ces objets se conforment à l'interface définie par la super-classe.

Le fait qu'une sous-classe peut redéfinir une méthode introduit le polymorphisme. En effet, la rédéfinition introduit une nouvelle forme pour l'opération qui est spécifique aux instances de la sous-classes ou à celles de ses propres sous-classes.

La somme de ces deux considérations donne naissance à un principe très puissant en programmation par objets, le *principe de substitution* qui s'énonce de la manière suivante :

> *Partout dans un programme où l'on attend une instance d'une classe `A`, il est possible de substituer une instance d'une sous-classe `B` de `A` sans qu'en résulte une erreur.*

Exemple 23 Reconsidérons notre exemple précédent d'une classe `A` et de sa sous-classe `B`. La séquence suivante illustre le principe de substitution, tout en faisant bien apparaître la recherche dynamique de la méthode appliquée :

`a = b ;`	affectation dans `a` d'une instance de la classe `B`, ce qui est légal selon le principe de substitution.
`a instanceof B` ⇒ `true`	l'objet dont l'identificateur est dans la variable `a` est bel et bien une instance de la classe B.
`a.montreToi()` ⇒ `"2B"`	on obtient le résultat de `montreToi` sur une instance de la classe `B`, ce qui illustre bien la sélection dynamique de la méthode `montre` qui dépend de la classe de l'objet affecté à `a`, et non au type déclaré pour `a`.

Notons cependant qu'il serait incorrect d'utiliser dans ce contexte une expression `a.evite()`, puisque les instances de `A` ne savent pas répondre au message `evite`. La vérification de type de Java n'autorise à utiliser que les méthodes connues du type attribué à la variable lors de sa déclaration. C'est parce que les instances de la sous-classe `B` savent aussi répondre aux messages connus des instances de `A` que le principe de substitution fonctionne. Mais il s'accompagne aussi d'une perte d'information, puisque l'instance de `B` placée dans une variable de type `A` ne peut qu'utiliser les méthodes connues aussi de `A`, et non les ajouts qui lui sont faits par rapport à `A`. □

Héritage et création des objets

L'héritage est utulisé pour étendre la description des objets faites par les superclasses. On vient de voir comment l'héritage influe sur la recherche des méthodes qui doivent être appliquées pour répondre aux envois de messages. Mais l'héritage influe également sur la description de l'état de l'objet par des variables d'instance. Chaque classe définit ses variables d'instance. La règle normale de l'héritage des variables d'instance dit qu'un objet instance d'une sous-classe `B` de la superclasse `A` possède toutes les variables définies par `A` plus les variables définies par `B`.

La création des objets supposent l'initialisation des varaibles d'instance de l'objet. En Java, par exemple, les constructeurs ou méthodes d'initialisation sont utilisés pour ce faire. Cependant, lors de la création d'une instance de la sous-classe, par exemple `B`, c'est un constructeur de la classe `B` qui est appelé pour initiliser le nouvel objet, et non un constructeur de la classe `A`. Ceci pose la question de savoir si on doit répéter dans le constructeur de `B` les initialisations des variables définies par `A` qui sont normalement dévolues aux constructeurs de cette dernière.

Cette répétition n'est en fait ni souhaitble, ni même toujours possible. Elle n'est pas souhaitable, car si on devait modifier la classe `A`, on ne veut pas devoir modifier les constructeurs de toutes les sous-classes de `A` pour tenir compte de changements sur les variables d'instance de `A`. Elle n'est pas toujours possible, car la visibilité des variables peut être de type «privée», c'est-à-dire que des variables de `A` peuvent être invisibles de `B`.

Pour répondre à ce problème, il est possible à un constructeur de la sous-classe d'utiliser un constructeur de la superclasse pour initialiser les variables définies par la superclasse. En Java, les constructeurs ont le nom de la classe et se distinguent par le nombre et les types de leurs paramètres. Pour appeler un constructeur de la superclasse dans le constructeur de la sous-classe, on utilise la syntaxe :

```
super(a1, ..., an) ;
```

qui désigne le constructeur parmi les superclasses qui a `n` paramètres des types de `a1`, ..., `an`. Il y a par contre une restriction importante de Java dans ce cas, à savoir que l'énocé d'appel au constructeur de la superclasse doit obligatoirement apparaître comme le premier énoncé dans le constructeur de la sous-classe.

Exemple 24 Considérez les classes `A` et `B` suivantes :

```
public class A {
    int i ;

    public    A(int init)     { i = init ; }
}

public class B {
    double x ;

    public    B(int init_i, double init_x) {
        super(init_i) ;        // appel du constructeur A(int)
        x = init_x ;           // initialisations locales
    }
```

```
}
```

Le constructeur de `A` initialise la variable `i`. Dans le constructeur de `B`, on reçoit en paramètre à la fois les paramètres nécessaires au constructeur de `A` (ici, `init_i`), et ceux nécessaires à la classe `B` (ici, `init_x`). Dans le corps du constructeur de `B`, on commence par appeler le constructeur de `A` par `super`, puis on fait les initialisations spécifiques à `B`. □

Du besoin de classes et de méthodes abstraites

La création de hiérarchie de classes est une activité qui requiert expérience et anticipation. On cherche à partager variables et méthodes de telle façon à minimiser les répétitions. On doit aussi introduire par les superclasses des concepts généraux utiles dans les programes, comme c'est le cas dans l'exemple des figures géométriques suivant.

Exemple 25 Revenons à nouveau sur l'exemple des formes géométriques. Si la classe `Figure` définit la méthode `affiche`, la déclaration d'un variable de type `Figure` :

```
Figure forme ;
```

établit que tout objet dont l'identificateur d'objet serait affecté à la variable `forme` devra se conformer à l'interface de la classe `Figure` en étant soit une instance de la classe `Figure` ou encore, par le principe de substitution, une instance d'une sous-classe de `Figure`. C'est ainsi que dans notre exemple de la section précédente, il devient possible de ranger dans `forme` une instance de n'importe laquelle des sous-classes de `Figure`, puis d'exécuter l'envoi de message `forme.affiche()` en étant sûr que le receveur possède bien une méthode `affiche` capable de répondre à ce message. □

Il arrive cependant que lorsqu'on factorise des méthodes dans une superclasse, on se retrouve dans la situation de définir une classe incomplète. Considérez par exemple deux classes `B` et `C` ayant des méthodes `m1` et `m2`. Supposons que les définitions de `m1` dans les deux classes soient exactement les mêmes, mais qu'elles appellent la méthode `m2` qui, elle, est différente dans les deux classes. Dans cette situation, on peut factoriser dans une superclasse `A` la méthode `m1` qui a la même définition dans les deux classes, mais pas la méthode `m2`.

Le problème, c'est que `A` où la méthode `m1` sera factorisée ne possèdera pas de méthode `m2`. La classe est donc incomplète, et Java par exemple se plaindra à la compilation de `A` que la méthode `m1` appelle une méthode `m2` qui n'existe pas dans `A`. En faite, une telle classe incomplète est appelée *classe abstraite*, et la méthode `m2` doit être définie dans `A` comme une *méthode abstraite*. Une classe abstraite est une classe incomplète dont on ne peut créer d'objets. Une méthode abstraite est une déclaration de méthode comportant la signature (nom, type de résultat, types de paramètres) mais pas de corps. Une telle déclaration dans `A` peut être vue comme l'intention de définir la méthode `m2` dans les sous-classes, et d'ailleurs Java l'interprètera comme une obligation pour les sous-classes conrètes de `A` de définir la méthode.

3.3 Programmation avec des classes et des objets

Programmer prend des sens différents et fait appel à des compétences différentes selon le niveau auquel on se place. Ces différences ont donné lieu à l'émergence de différents métiers, tel que nous allons le voir ci-après. Au niveau de détail le plus bas, programmer veut dire

développer une série de lignes de programme, voire quelques méthodes, pour résoudre un problème donné. Cet aspect, auquel les américains ont donné le nom de «*programming-in-the-small*» sera abordé plus longuement dans le prochain chapitre. Ce niveau requiert de la compétence *tactique* et une bonne connaissance des solutions à certaines familles de problèmes. Des méthodes formelles, dites axiomatiques, ont également été développées sur la base de la notion d'assertions que nous avons vue.

À l'autre extrémité du spectre, il faut savoir organiser les très grandes applications dans toute leur complexité inhérente à leur taille, leur nombre élevé de composants et au grand nombre d'interactions possibles entre ceux-ci. C'est un travail d'architecture qui demande une vision globalisante, stratégique, et un excellent jugement sur les grands compromis à réaliser entre des objectifs qui peuvent être contradictoires. C'est ce que les américains appellent «*programming-in-the-large*». Contrairement à l'architecture classique, cette discipline de l'informatique est encore peu codifiée, malgré des progrès importants depuis une ou deux décennies. Pour l'essentiel, elle repose sur une bonne intuition et sur une connaisance de catalogues de solutions stéréotypées.

Avec la programmation par objets émerge de façon fulgurante une discipline méconnue qui se situe un peu entre les deux premières et que l'on peut dénommer «*programming-in-the-medium*». L'objectif ici est de découper de façon modulaire de grandes fonctionnalités des systèmes en morceaux et sous-morceaux cohérents[9] et de définir les interactions entre les morceaux qui vont permettre d'obtenir le résultat à un certain calcul. C'est à ce niveau de programmation que la métaphore objet joue pleinement son rôle comme nous allons le voir maintenant.

Une fois les morceaux et leurs responsabilités respectives identifiés, il faut arriver à une spécification précise du rôle de chacune de ces entités. L'approche de la programmation contractuelle, que nous allons introduire au chapitre 6, a été développée pour répondre à ce besoin. Ensuite, le développement de chacune des classes fait appel à l'aspect «*programming-in-the-small*» pour écrire les méthodes ; certaines des compétences tactiques alors nécessaires seront abordées dans les prochains chapitres.

3.3.1 Analyse des problèmes et identification des classes

La métaphore de la simulation induit une méthode d'analyse des problèmes de programmation dans l'approche objet. Typiquement, un problème de programmation, ou dit autrement une application à écrire, est d'abord exprimé de façon très succinte. Cela peut se borner à une description du genre : nous souhaiterions obtenir un programme qui gère la location d'une flotte de véhicule aux particuliers. C'est le rôle de l'*analyste*, ou encore de ce qu'on appelle maintenant l'*architecte logiciel*, d'élaborer à partir d'une telle description et d'échanges avec le donneur d'ordres, l'architecture globale de l'application qui sera ensuite élaborée par les programmeurs.

La première tâche à réaliser consiste à identifier les grandes lignes du projet de façon à le diviser en parties relativement indépendantes dont la réalisation pourra être confiée à différents programmeurs de l'équipe (ou différentes équipes de programmeurs selon la taille du projet). Suivant l'approche objet, l'architecte logiciel a alors pour objectif l'identification des actions qui seront réalisées par l'application et l'attribution de ces actions à différentes entités identifiées dans le problème. Pour ce faire, on utilise souvent des techniques d'analyse qui s'apparente à l'écriture de scénarios de plus en plus élaborés. Dans ce scénario, les acteurs

[9]Le découpage peut bien sûr impliquer plus de deux niveaux.

seront les objets, les actions à réaliser sous la responsabilité de chacun de ces objets vont devenir les méthodes, et la séquence des actions apparaissant dans le scénario sous la forme de requêtes allant d'un objet à l'autre va devenir le fil d'exécution dans l'application finale.

Le point de départ consiste donc à identifier les principales classes de l'application. Ce peut être l'agence, les clients, les véhicules, les contrats, comme dans notre exemple d'agence de location de voitures. En général, les premières grandes divisions se présentent de façon assez évidentes. Mais attention, on peut être amener à revenir sur le découpage initial[10]. Ce découpage initial obtenu, on peut commencer à identifier le rôle de chacun de ces composants. C'est en général à ce moment que l'architecte décide avec le client quels seront les moyens d'interaction, ou la vision externe de l'application, qui doit servir à interagir avec elle pour lui fournir les données et pour en obtenir les résultats. On utilise de plus en plus les interfaces graphiques aujourd'hui, depuis leur popularisation dans les années '80.

Une fois le premier dégrossissage réalisé, l'architecte logiciel applique itérativement les techniques bien connues en ingéniérie qui consistent à diviser les entités plus grandes en en une composition d'entités plus petites. Le scénario est alors raffiné pour tenir compte des rôles respectifs de ces sous-composants, leurs méthodes et les requêtes qu'ils échangent. Ces étapes de raffinement se poursuivent jusqu'à obtenir des objets unitaires, cohérents, et de taille relativement modeste qu'un programmeur saura réaliser directement dans le langage de programmation choisi.

L'identification des composants et leur découpage en sous-composants est guidé par deux grands principes qui assure la modularité des applications à objets :

1. Un composant doit avoir un ensemble relativement petit et cohérent de responsabilités dans l'application.
2. Les composants doivent interargir le moins possible les uns avec les autres.

La réalisation de ces deux objectifs est clairement contradictoire. Si l'application contient un seul composant, il est clair que son interaction avec l'extérieur sera minimale, mais il est probable que son ensemble de responsabilités sera trop grand pour être défini simplement... Par contre, si le composant devient trop petit, il ne pourra rien faire sans utiliser d'autres composants d'où des interactions nombreuses et incessantes. Un équilibre doit être trouvé entre les deux termes de ce dilemme. La réalisation de cet équilibre est un gage de qualité du programme obtenu, et ... de compétence de l'architecte logiciel.

En cours de définition du scénario, on explicite généralement l'information que doit détenir l'objet, ou l'information qu'il doit représenter avec les traitements associées. Il devient alors possible de préciser quelles seront les données détenues par l'objet et qui formeront son état.

En résumé, l'analyse doit :

1. Préparer des scénarios.
2. Identifier les objets et leur rôle.
3. Définir les actions réalisées par les objets.
4. Identifier les données devant être détenues et mémorisées par les objets.

Cette analyse procède par itération et raffinement des objets en suivant ce processus plusieurs fois après avoir identifiés des sous-composants. Ce processus itératif revient inévitablement sur des décisions précédentes, car il est rare de prendre toujours de bonnes décisions sans avoir une image complète et précise de l'application. Il est donc important à chaque étape de bien anticiper les changements futurs. La préparation au changement se

[10]Cent fois sur le métier, remettez votre ouvrage...

réalise justement par l'observation précise des principes précédents, c'est-à-dire la réduction des interactions entre les objets et leur définition autour d'une taille et des responsabilités limités, cohérentes et précises.

Par exemple, dans l'application location de voiture, les grands composants initiaux sont l'agence, les véhicules, les clients, les contrats. Il serait inutilement complexe de confondre ici contrat et client ou contrat et véhicule. Lors de la phase de raffinement, on peut vouloir faire apparaître qu'il existe plusieurs sortes de véhicules et de contrats, ou encore que les agences peuvent faire partie d'un réseau, ou encore qu'un contrat se divise en une partie location de la voiture et une partie assurance qui est éventuellement sous-traitée à une société d'assurances.

3.3.2 Développement des classes

Lorsque tous les composants sont identifiés, avec leurs savoir-faire (méthodes) et leur état (variables d'instance), on peut passer à la phase de spécification avant celle de la définition dans le langage de programmation. À cette étape, il est utile d'appliquer ces principes que tout objet est conçu autour d'un invariant structurel et que toute méthode est développée autour de pré- et postconditions.

Invariant structurel : représentationnel et fonctionnel

À la fin du chapitre 2, la notion d'invariant structurel a été développée comme moyen d'exprimer les conditions qui doivent toujours être vérifiées par les valeurs des variables participant à la représentation interne d'une entité. Par définition, un objet représente une entité du monde de l'application à l'aide de ses variables d'instance. On applique donc naturellement la notion d'invariant structurel à la partie représentation de l'objet. Lors du développement de la classe, en parallèle avec la définition de la représentation, cette notion d'invariant permet de formaliser les conditions qui font que cette représentation est cohérente. Les invariants sont alors déclarés dans la définition même de la classe.

On distingue deux parties dans l'invariant structurel de l'objet : l'invariant structurel *fonctionnel* et l'invariant structurel *représentationnel*. Un objet peut vérifier un ensemble de conditions de fonctionnement importantes du point de vue de son utilisation. Ces conditions forment un invariant d'interface que nous appellerons *invariant structurel fonctionnel*, ou plus simplement *invariant fonctionnel*. L'invariant fonctionnel regroupe donc l'ensemble des conditions visibles de l'extérieur qui font qu'un objet est dans un état cohérent.

Exemple 26 Imaginons un objet qui représente un réservoir d'essence. On sait bien, et cela fait partie de l'utilisation de l'objet réservoir (de son interface), qu'un réservoir a une capacité, que son état donne la quantité d'essence contenue dans le réservoir et qu'il peut être plein, vide, ou partiellement rempli. L'invariant fonctionnel dans ce cas est que le réservoir ne peut jamais contenir plus d'essence que sa capacité. Cette condition fait partie intégrante de l'utilisation d'un réservoir. □

Par sa nature même, l'invariant fonctionnel est défini en terme de conditions vérifiables depuis l'extérieur de l'objet. Il doit donc pouvoir s'écrire en utilisant uniquement des méthodes faisant partie de l'interface de l'objet pour éviter que pour observer cet invariant, l'utilisateur de l'objet ne deviennent dépendant du choix de représentation interne à cet objet.

Exemple 27 Si on considère à nouveau l'exemple du rectangle[11], il va de soi qu'un rectangle bien défini a une surface strictement positive. Si le rectangle est représenté par deux points : un point supérieur gauche et un coin inférieur droit, la condition de définition cohérente impose que le point supérieur gauche reste effectivement à gauche et au-dessus du point inférieur droit. Mais cette expression de la condition dépend totalement du choix de représentation du rectangle. Si l'on choisit plutôt de représenter le rectangle par son point inférieur gauche, sa hauteur et sa largeur, l'invariant s'exprimera plutôt en imposant que les valeurs de hauteur et de largeur soient toutes les deux strictement positives. Dans ce cas, on s'assurera de passer par des méthodes (à définir le cas échéant) pour exprimer et vérifier l'invariant fonctionnel. Dans le cas présent, l'ajout d'une méthode `surfacePositive` permet de vérifier la condition de l'extérieur sans connaître les choix de représentation interne. □

Pour sa part, l'*invariant structurel représentationnel*, ou plus simplement *invariant représentationnel*, sert à exprimer des contraintes qui dépendent uniquement des choix de représentation sans implications sur la fonction visible de l'objet. Il s'exprime le plus souvent par des relations entre valeurs de variables d'instance.

Exemple 28 Considérez un petit bureau de poste de campagne où s'accumule le courrier (lettres recommandées et paquets) n'ayant pu être livré parce que le destinataire était absent lors du passage du facteur. Le courrier est rangé dans une des 26 cases, pour les lettres A à Z, où ils sont séparés en fonction de la première lettre du nom de famille du destinataire. Cependant, les paquets, peu nombreux au demeurant, trop gros pour entrer dans les cases sont simplement déposés dans une pièce adjacente bien déterminée. L'invariant structurel de cette organisation est :

> *le paquet est soit dans le casier portant l'initiale de son nom de famille ou encore dans la pièce adjacente et nulle part ailleurs.*

Le client venant chercher son paquet n'a pas à avoir connaissance que cet invariant est vérifié, puisque ce dernier n'a rien à voir avec le service rendu du point de vue fonctionnel externe. C'est par contre un invariant structurel de l'organisation choisie par le bureau de poste, car il s'agit d'une condition qui doit toujours être vérifiée pour que le préposé trouve à tous les coups le paquet du client. □

Les deux types d'invariants, fonctionnels et représentationnels, peuvent être perturbés lors de la modification de l'état de l'objet dans les méthodes appelées modificateurs. Dans la mesure où les variables d'instance sont affectées une par une, il arrive nécessairement que l'invariant soit moméntanément violé pendant que l'état de l'objet est en cours de modification. Si on translate un rectangle représenté par deux points, il faudra nécessairement modifier un des points avant l'autre. Entre la modification du premier et du second point, il est probable que l'invariant sera momentanément violé. La règle qui doit être appliquée ici est que l'invariant est vrai lorsque l'on débute l'exécution d'une méthode sur cet objet et il doit toujours être vrai à la fin de l'exécution de la méthode. Le programmeur doit prendre soin de cette obligation lors de l'écriture des méthodes.

Pré- et postconditions versus l'invariant structurel

Au chapitre 2, des pré- et postconditions ont été associées aux méthodes de manière à permettre à leur utilisateur de connaître les conditions devant être vraies pour que la méthode

[11] dont les côtés sont parallèles aux axes du plan cartésien.

1. Diviser le problème en sous-problèmes en identifiant les classes et les méthodes à partir d'une description narrative du problème.
2. Avant de définir les variables d'instances et d'écrire quelque code de méthode que ce soit, arriver à une description complète et consistente de l'interface de toutes les classes identifiées.
3. Sauf si une méthode n'est jamais appelée de l'extérieur, déclarer les méthodes comme publiques.
4. À partir de la description narrative du problème et des opérations définies sur chacune des classes, identifier les informations nécessaires à chaque objet et en tirer la définition des variables d'instances. À moins qu'il ne soit absolument nécessaire d'y accèder de l'extérieur, déclarer les variables d'instances comme privées.
5. Si des contraintes individuelles ou croisées apparaissent pour définir les valeurs admissibles des variables d'instance, les exprimer sous la forme d'un invariant structurel de la classe. Distinguer les conditions qui font partie intégrante du fonctionnement de l'objet de celles qui ne sont dépendantes que de la représentation. Dans l'invariant fonctionnel, masquer les choix de représentation en utilisant des méthodes pour le vérifier plutôt que directement les variables d'instance.
6. Si une méthode prend des paramètres et que des contraintes individuelles ou croisées apparaissent pour définir les valeurs des paramètres admissibles, définir les pré-conditions correspondantes sur les méthodes concernées. Si une méthode retourne un résultat et que des contraintes individuelles ou croisées entre ce résultat et les paramètres apparaissent, définir également la post-condition correspondante.
7. Si une méthode n'est appelable que dans certains états de l'objet, ajouter à sa précondition la condition correspondante.
8. Inclure dans la postcondition des méthodes qui modifient l'état de l'objet les conditions vérifiées sur la modification de l'état de l'objet.

FIG. 3.9 – Rappel des lignes directrices pour la programmation avec des classes.

puisse s'exécuter correctement, puis les conditions qui seront vérifiées après cette exécution.

Dans un objet, les méthodes agissent aussi sur la représentation interne de l'objet, dont les variables d'instance font partie des variables visibles depuis la méthode. Pour obtenir une vision complète, méthodes et représentation, il faut inclure le principe selon lequel l'invariant structurel est vérifié au départ (précondition implicite) et qu'il devra être respecté à l'issue de l'exécution de la méthode (postcondition implicite). La précondition d'une méthode contient donc deux parties : les conditions propres à la méthodes sur ses paramètres et l'invariant structurel (qui n'a pas besoin d'être répété). La précondition peut aussi spécifier que la méthode n'est appellable que dans certains états de l'objet en posant des conditions sur ses variables d'instance.

De même, la postcondition contient une partie propre concernant son résultat et une partie invariant structurel qui n'a pas à être répété. La postcondition peut aussi spécifier des conditions sur les transformations de l'état de l'objet. Les parties de la pré- et de la postcondition qui sont propres à la méthode sont associées à la définition de cette dernière, comme nous l'avons vu.

L'utilisation systématique des notions d'invariant structurel, de pré- et de postconditions font partie d'une approche appelée programmation contractuelle. La *programmation contractuelle* consiste à définir systématiquement sur toutes les méthodes de telles pré- et postconditions et à les interpréter comme un *contrat* entre l'appelant et la méthode qui s'énonce comme suit :

> *Si l'appelant fournit des paramètres réels et appelle la méthode dans un contexte où ceux-ci et les variables visibles ont des valeurs satisfaisant la précondition, alors la méthode s'engage à rendre un résultat et à modifier l'état des variables visibles de telle manière à ce que la postcondition soit vraie lorsque le contrôle retourne à l'appelant. De plus, si le receveur est dans un état respectant son invariant structurel lors de l'appel, la méthode s'engage à le laisser dans un état respectant aussi cet invariant à la fin de son exécution.*

Nous allons revenir plus longuement sur ces notions au chapitre 6.

Assertions et héritage

L'utilisation d'invariants, de pré- et de postconditions s'étend au cas des classes définies différentiellement par héritage. En fait, les assertions définies sur la superclasse sont combinées avec celles de la sous-classe pour donner les assertions effectives s'appliquant sur les différents éléments.

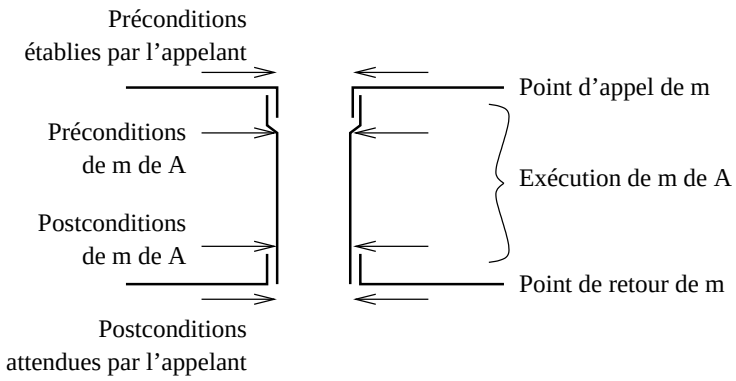
Dans le cas de l'invariant, il apparaît clairement que l'instance d'une sous-classe utilisant la représentation définie par la superclasse, l'invariant de cette dernière s'applique intégralement aux instances de la sous-classe. Il faut donc que l'invariant de la superclasse soit une conséquence logique de celui de la sous-classe. Supposons que ce osit le cas, quelle combinaison de l'invariant de la superclasse avec l'invariant déclaré dans la sous-classe permet d'éviter de répéter dans la sous-classe toutes les conditions de la superclasse ? En faisant la conjonction des deux, nous obtenons l'effet escompté. L'invariant effectif de la sous-classe est donc obtenu en faisant la conjonction de l'invariant de la superclasse et de celui défini dans la sous-classe. L'invariant défini dans la sous-classe n'a donc pas à répéter l'invariant de la superclasse ; il se contente de déterminer les relations devant exister entre les nouveaux éléments de représentation définis dans la sous-classes ou entre ces derniers et ceux définis dans la superclasse.

Pour le cas des pré- et postconditions, il faut tenir compte du principe de substitution pour comprendre la combinaison entre les préconditions d'un méthode héritée `m` et sa redéfinition dans la sous-classe. Soient `A` une classe et `B` sa sous-classe. Soit `m` une méthode définie dans `A` et redéfinie dans `B` dont les pré- et posconditions sont respectivement $pre^{\mathtt{A}}_{\mathtt{m}}$, $post^{\mathtt{A}}_{\mathtt{m}}$, $pre^{\mathtt{B}}_{\mathtt{m}}$ et $post^{\mathtt{B}}_{\mathtt{m}}$. Le principe de substitution nous dit que partout où une instance de `A` est attendue, on peut utiliser une instance de `B`. Supposons donc un appel `a.m(...)`, où `a` est une variable de type `A`.

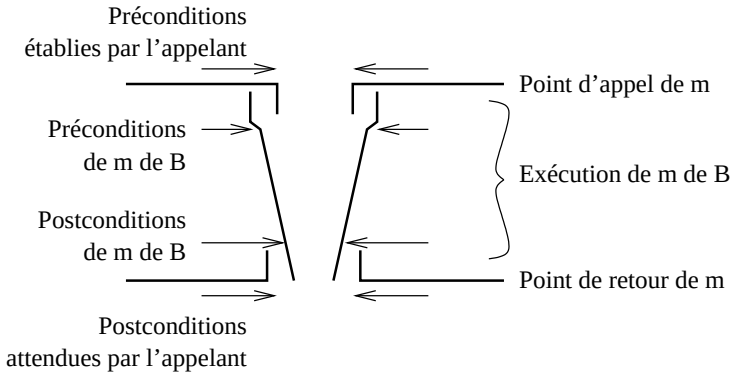
Lorsque le programmeur écrit cet appel, il va vérifier que la précondition $pre^{\mathtt{A}}_{\mathtt{m}}$ est observée et il va supposer qu'après l'appel la postcondition $post^{\mathtt{A}}_{\mathtt{m}}$ sera aussi vérifiée. Ne sachant pas nécessairement que `A` peut avoir une sous-classe, il ne peut vérrifier les assertions des sous-classes éventuelles. Si on place une instance de `B` dans la variable `a`, la méthode `m` de `B` sera appelée. Pour que l'appel se passe sans erreur, il faut donc que les relations suivantes existent :

$pre^{\mathtt{A}}_{\mathtt{m}} \Rightarrow pre^{\mathtt{B}}_{\mathtt{m}}$, et $post^{\mathtt{B}}_{\mathtt{m}} \Rightarrow post^{\mathtt{A}}_{\mathtt{m}}$

La précondition $pre^{\mathtt{B}}_{\mathtt{m}}$ doit donc être égale ou plus faible que $pre^{\mathtt{A}}_{\mathtt{m}}$, alors que la postcondition $post^{\mathtt{B}}_{\mathtt{m}}$ doit être égale ou plus forte que $post^{\mathtt{A}}_{\mathtt{m}}$. Si ces relations tiennent, quelle combinaison des



(a) Appel de la méthode `m` sur une instance de `A`



(a) Appel de la méthode `m` sur une instance de `B`

FIG. 3.10 – Illustration des relations entre pré- et postconditions lors de l'application du principe de substitution, par analogie avec un collecteur faisant passer un état de calcul avant l'appel vers un état de calcul après l'appel. Les pré- et postconditions établies lors de l'appel et attendues lors du retour ne changeant pas, la précondition de `m` sur `B` doit être égale ou plus faible à celle exigée par `m` sur `A`, alors que la postcondition de `m` sur `B` doit être plus forte que celle sur `A`.

pré- et postconditions nous donnera la flexibilité voulue afin d'éviter les répétitions dans les assertions ? Pour les préconditions, il faut faire la disjonction puisqu'alors :

$$pre^{\mathtt{A}}_{\mathtt{m}} \Rightarrow pre^{\mathtt{A}}_{\mathtt{m}} \vee pre^{\mathtt{B}}_{\mathtt{m}}$$

alors que pour les postconditions, il faut faire la conjonction puisqu'alors :

$$(post^{\mathtt{B}}_{\mathtt{m}} \Rightarrow post^{\mathtt{A}}_{\mathtt{m}}) \Rightarrow (post^{\mathtt{B}}_{\mathtt{m}} \Rightarrow post^{\mathtt{B}}_{\mathtt{m}} \wedge post^{\mathtt{A}}_{\mathtt{m}})$$

Les pré- et postconditions définies dans `B` n'ont donc pas à répéter les assertions de `A`, mais plutôt définir les conditions affaiblissantes pour la précondition et renforçantes pour la postcondition.

3.4 Type abstrait de données et classe

Au chapitre précédent, la notion de type abstrait de données a été développée comme suite naturelle du typage. Un type est constitué d'un ensemble de valeurs, d'opérations applicables à ces valeurs et d'une représentation qui définit comment les valeurs sont construites en terme de données primitives. Les choix de représentation des valeurs du type déterminent la mise en œuvre des opérations. Un type abstrait est un type qui établit une frontière entre l'utilisateur du type et son implanteur. Cette frontière masque à l'utilisateur les choix de représentation et donc la mise en œuvre précise des opérations pour ne lui laisser voir que la manière de créer des valeurs du type et d'appliquer les opérations sur les valeurs de ce type.

En programmation par objets, c'est la classe qui fournit le moyen de définir des types abstraits de données. Les objets sont considérés comme des valeurs d'un type de données défini par leur classe d'instantiation. Une classe définit le moyen de créer des valeurs du type qu'elle représente et aussi établit la liste des opérations applicables à ses instances, ce qui forme ce que l'on appelle son *interface* ou encore son *protocole*. Le programmeur qui implante la classe, lui, va faire les choix de représentation qui vont amener la déclaration des variables d'instance ; il va aussi écrire les méthodes en fonction des choix de représentation qu'il a faits.

La frontière apparaît entre l'utilisateur de la classe et l'implanteur dans la mesure où l'utilisateur ne voit pas ou ne peut pas remettre en cause les choix de représentation et de mise en œuvre faits par l'implanteur. En fait, la classe établit le principe selon lequel un objet regroupe un ensemble de données privées (invisibles de l'extérieur) et un ensemble de méthodes publiques ; la seule façon de faire une opération sur un objet est d'appeler l'une de ses méthodes. La classe est donc un outil d'encapsulation qui ne permet à l'utilisateur que la manipulation de ses instances via les opérations qu'elle définit.

Comme pour les types de données abstraits, cette notion d'encapsulation présente l'avantage de l'application du principe de la localité. Les données étant privées, la représentation n'est manipulée que par les méthodes de la classe de l'objet et la cohérence de ces données entre elles est par conséquent assurée par les méthodes. Cette cohérence est exprimée par l'invariant structurel de l'objet. On peut modifier la représentation sans changer les clients d'un objet. Les seules modifications nécessaires suite à un changement de représentation pour une classe se limitent aux méthodes de cette classe.

3.5 Éléments de savoir-faire : utilisation des objets en Java

Au chapitre précédent, nous avons introduit la déclaration de classes racines en Java, ces classes qui permettent de créer des objets qui ont pour rôle de lancer l'exécution d'un programme via leur méthode `main`. La plupart des classes en Java ne sont pas des classes racines ; elles décrivent plutôt des objets qui seront créés et utilisés pour exécuter un certain calcul. Nous allons ici illustrer la création d'une telle classe en Java, de même que nous allons introduire certains outils complémentaires de cette création pour la documentation et la contractualisation.

3.5.1 Les objets polynômes de degré 2

Plaçons-nous dans le cadre d'un hypothétique programme de manipulation de polynômes. En particulier, supposons que ce programme fasse intervenir des polynômes particuliers, des polynômes de degré 2. Un polynôme de degré 2 dont tous les coefficients sont nuls sauf

éventuellement ceux des termes de degré 2, 1 et 0. On peut les écrire sous la forme :

$$ax^2 + bx + c$$

où a, b et c sont les coefficients des termes de degré 2, 1 et 0 respectivement. Comme pour tout polynôme, on doit pouvoir additionner et soustraire des polynômes de degré 2. On sait aussi que les polynômes de degré 2 ont ceci de particulier qu'on connaît une formule analytique simple pour trouver leurs racines. Cette formule, que nous avons vu à plusieurs reprises, implique le calcul d'une racine carrée dont l'argument est appelé le discriminant. Si le discriminant est positif, le polynôme a deux racines, s'il est nul, le polynôme a une seule racine, et s'il est négatif, le polynôme n'a pas de racines (ou des racines complexes).

Pour manipuler informatiquement de tels polynômes, on va devoir les représenter. Pour cela, nous définissons une classe `Poly2` qui va décrire la forme des objets représentant des polynômes de degré 2. En ce qui concerne la représentation, trois nombres réels caractérisent complètement un polynôme de degré deux : ses trois coefficients. Il suffit donc de trois variables capables de contenir chacune une valeur réelle.

Pour les opérations, on doit pouvoir créer des objets à partir de la classe `Poly2` ; cela suppose que la classe définisse des *méthodes d'initialisation* des objets lors de leur création. Ensuite, on veut pouvoir changer les valeurs des coefficients des polynômes. De telles méthodes qui changent l'état d'un objet sont appelées des *modificateurs*. Au coeur des savoir-faire de l'objet polynôme, nous voulons avoir des opérations pour calculer le discriminant, pour calculer les deux racines, pour additionner et pour soustraire des polynômes. Enfin, il est utile dans les traces de programme d'être capable d'afficher le contenu d'un objet ; nous allons donc définir une opération en ce sens pour les polynômes de degré 2.

3.5.2 Déclaration des variables de l'objet

Dans sa forme la plus simple, la déclaration des variables d'un objet prend une forme similaire à la déclaration des variables locales aux méthodes vue au chapitre précédent, c'est-à-dire un nom de type, suivi d'une liste de noms de variables et terminée par un point-virgule :

```
double a, b, c ;
```

Dans la déclaration d'une variable d'un objet, on fait intervenir un autre concept qui est celui de la visibilité de la variable en question. Normalement, en programmation par objets, on insiste pour que les variables de l'objet ne soient visibles que par les méthodes définies par la classe de l'objet lorsqu'elles sont appelées sur l'objet en question. Un objet polynôme, disons `p1` possèderait donc trois variables contenant les valeurs de ses coefficients, et ces variables ne seraient accessibles que par les procédures de la classe `Poly2` lorsqu'elles sont appelées sur l'objet `p1`.

En pratique, cette approche est trop restrictive. Il est parfois utile de permettre un accès aux variables d'un objet depuis une méthode s'exécutant sur un autre objet. Pour rendre accessible une variable à partir d'un autre objet, on déclare la variable avec un *modificateur de visibilité*[12] :

```
<modificateur-de-visibilité> <type> <liste-de-variables> ;
```

[12] Nous allons utiliser pour décrire la syntaxe de Java une notation empruntée aux grammaires de langage. Tout ce qui porte un nom placé entre les signes < et > doit être remplacé par des éléments du langage Java correspondant pour devenir un énoncé Java utilisable dans un programme.

```
public class    Poly2
{
    public static final Poly2   ZERO = new Poly2() ;

    private double coef2 ;
    private double coef1 ;
    private double coef0 ;

    public              Poly2() {
        coef2 = 0.0 ; coef1 = 0.0 ; coef0 = 0.0 ;
    } // ------------------------------------------------- Poly2()

    public              Poly2(double c2, double c1, double c0) {
        coef2 = c2 ; coef1 = c1 ; coef0 = c0 ;
    } // ------------------------------------------------- Poly2()

    public void         changeCoef2 (double nouvCoef) {
        coef2 = nouvCoef ;
    } // ------------------------------------------------- changeCoef2()

    public void         changeCoef1 (double nouvCoef) {
        coef1 = nouvCoef ;
    } // ------------------------------------------------- changeCoef1()

    public void         changeCoef0 (double nouvCoef) {
        coef0 = nouvCoef ;
    } // ------------------------------------------------- changeCoef0()

    public double       evalue(double  x) {
        return (coef2 * x + coef1) * x + coef0 ;
    } // ------------------------------------------------- evalue()

    public double       calculeDiscriminant() {
        return coef1 * coef1 - 4.0 * coef2 * coef0 ;
    } // ------------------------------------------------ calculeDiscriminant()

    public double       racine1() {
        return (-coef1 + Math.sqrt(calculeDiscriminant()))/ (2.0 * coef2) ;
    } // ------------------------------------------------- racine1()

    public double       racine2() {
        return (-coef1 - Math.sqrt(calculeDiscriminant()))/ (2.0 * coef2) ;
    } // ------------------------------------------------- racine2()

    public Poly2        additionne(Poly2 autrePoly) {
        return new Poly2(coef2 + autrePoly.coef2, coef1 + autrePoly.coef1,
                         coef0 + autrePoly.coef0) ;
    } // ------------------------------------------------- additionne()

    public Poly2        soustrait(Poly2 autrePoly) {
        return new Poly2(coef2 - autrePoly.coef2, coef1 - autrePoly.coef1,
                         coef0 - autrePoly.coef0) ;
    } // ------------------------------------------------- soustrait()

    public Poly2        inverseAdditif() {
        return Zero.soustrait(this) ;
    } // ------------------------------------------------- inverseAdditif()

    public String       toString() {
        return "[" + coef2 + ", " + coef1 + ", " + coef0 + "]" ;
    } // ------------------------------------------------- toString()

} // ---------------------------------------- classe Poly2
```

FIG. 3.11 – Classe `Poly2`

Accessible à	Visibilité des variables et méthodes			
	`public`	`protected`	*package*	`private`
Même classe	oui	oui	oui	oui
Classe dans le même «*package*»	oui	oui	oui	non
Sous-classe dans un autre «*package*»	oui	oui	non	non
Autre classe, dans un autre «*package*»	oui	non	non	non

FIG. 3.12 – Modificateurs de visibilité de Java

Les quatre modificateurs de visibilité de Java sont : `public`, `protected`, `package` et `private`. Ces quatre modificateurs impliquent plus ou moins de libéralité dans l'accessibilité de la variable depuis un autre objet. Un exemple de déclaration est :

```
private double coef2 ;
```

Cet énoncé déclare une variable de nom `coef2` de type `double` et dont le modificateur de visibilité est `private`.

En Java, l'autre objet qui souhaite accéder une variable d'un objet `p1`, par exemple, peut faire partie de l'une des quatre catégories suivantes :

1. Un objet créé à partir de la même classe que celle de `p1`, c'est-à-dire `Poly2`.
2. Un objet créé à partir d'une autre classe que c1 mais qui appartient au même *package* que la classes de `p1`. Les classes en Java sont divisées en groupes de classes conceptuellement reliées qui forment alors ce que Java appelle un *package*.
3. Un objet créé à partir d'une sous-classe de la classe de `p1` mais qui est dans un autre *package* que celui de `Poly2`.
4. Enfin, un objet créé à partir d'une autre classe appartenant à un autre *package* que celui de `Poly2`.

Le tableau de la figure 3.12 résume les droits d'accès aux variables d'un objet selon la catégorie d'appartenance de l'objet depuis lequel l'accès est tenté, et le modificateur de visibilité utilisé dans la déclaration de la variable.[13] Reprenons notre exemple. Les trois variables de la classe `Poly2` sont déclarées `private`. Cela veut dire que seuls les objets créés à partir de la classe `Poly2` pourrons accéder à ces variables.

Les notions de visibilité pour les variables s'appliquent aussi aux méthodes. La déclaration d'une méthode peut donc faire intervenir un modificateur de visibilité. Les modificateurs sont les mêmes que pour les variables, et ils ont exactement la même signification. Une méthode est généralement publique, pour permettre aux autres objets d'activer l'objet qui la possède. Il est par contre possible de définir des méthodes privées (resp. protégées, ou de *package*) dont la visibilité sera aussi limitée qu'une variable privée (resp. protégées, ou de *package*).

En Java, la forme enrichie de cette déclaration par rapport au chapitre précédent est :

```
<modificateur-de-visibilité> <type-résultat> <nom>(<liste-paramètres-formels>)
{
   <corps>
}
```

[13] Remarquons qu'il n'y a pas à proprement parler de modificateur pour le cas *package*, ce cas étant obtenu par défaut lorsqu'il n'y a pas de modificateur de visibilité dans la déclaration.

Qualificatifs `static` et `final`

Le qualificatif `static` devant une déclaration de variable indique que la variable en question ne sera pas contenue dans chaque instance de la classe mais plutôt dans la classe elle-même. Cette variable devient alors une variable dite *de classe*, qui est partagée par l'ensemble des instances de la classe. Le qualificatif `final` indique que la valeur de la variable ainsi qualifiée ne peut plus être changée après son initialisation. On dit alors que l'on a défini une *constante*. Dans la classe `Poly2`, `ZERO` est une constante partagée par toutes les instances de la classe et visible de l'extérieur. On peut donc référer à cette constante par la notation pointée :

```
Poly2.ZERO
```

qui s'évalue à la valeur de la constante, c'est-à-dire dans le cas présent l'objet représentant le polynôme zéro.

Les qualificatifs `static` et `final` peuvent aussi s'appliquer sur des déclarations de méthodes. Une méthode statique est une méthode dite de classe qui peut donc être appelée sur la classe plutôt que sur une instance de la classe. La méthode `main` est statique car elle peut être appelée directement sur la classe (racine) qui la définit, sans avoir à créer d'instances de cette classe. Pour les méthodes, le qualificatif `final` a un sens qui sera développé plus loin, lorsque le concept d'héritage sera introduit.

3.5.3 Initialiseurs, modificateurs et accesseurs

L'exemple de la classe `Poly2` à la figure 3.11 illustre bien la déclaration de méthodes vue au chapitre précédent. Plus généralement, on distingue trois grandes familles de méthodes en Java :

Les méthodes d'initialisation : ces méthodes sont utilisées pour initialiser les objets lors de leur création.

Les modificateurs : ces méthodes, comme leur nom l'indique, peuvent modifier l'état de l'objet sur lequel elles sont activées.

Les accesseurs : ces méthodes ne font pas de modifications de l'état de l'objet.

Les méthodes d'initialisation jouent un rôle très particulier dans Java. Elles répondent donc à une déclaration particulière, imposée par le langage, un peu comme les méthodes `main`, mais pour des raisons différentes. Lorsqu'on crée un objet à partir d'une classe comme `Poly2`, cet objet va posséder les variables définies par cette classe : `coef2`, `coef1`, et `coef0`. Parce qu'il est désirable que tout objet respecte son invariant de structure dès sa création, Java impose que la classe définisse des méthodes qui vont servir à l'initialisation des objets au moment de leur création. C'est le rôle d'une des méthodes d'initialisation de la classe appelée éventuellement avec des arguments.

Pour des raisons de régularité du protocole de création des objets, les méthodes d'initialisation de Java :

- ne retournent jamais de résultat, et donc n'ont pas besoin de type de retour dans leur déclaration,
- portent toujours le nom de la classe, et
- diffèrent dans leurs listes d'arguments soit en nombre, soit en types.

Pour `Poly2`, nous proposons deux méthodes d'initialisation : une méthode sans argument qui va mettre à 0 tous les coefficients de l'objet, et ainsi créer un nouvel objet représentant le

polynôme zéro, et une méthode à trois arguments par laquelle on peut passer trois valeurs de coefficients lors de la création de l'objet, et donc définir un polynôme particulier.

Les modificateurs possèdent au moins un énoncé d'affectation à une des variables d'instance dans leur corps et peuvent, au besoin, retourner un résultat. Lorsqu'une méthode ne retourne pas de résultat, on l'indique à Java en utilisant un type particulier comme type de retour : le type `void`. La classe `Poly2` propose trois méthodes `changeCoef2`, `changeCoef1`, et `changeCoef0` qui permettent de modifier un à un la valeur des coefficients de l'objet polynôme. Ces trois méthodes sont donc des modificateurs ne retournant pas de résultat et déclarées avec un type de retour `void`.

Les accesseurs sont ainsi appelés parce qu'ils ne contiennent aucun énoncé d'affectation ; l'appelant et l'appelé ont donc la garantie que l'état du receveur ne sera pas modifié par leur exécution. Les méthodes `evalue`, `calculeDiscriminant`, `racine1`, `racine2`, `additionne`, `soustrait` et `toString` sont des accesseurs pour la classe `Poly2`. Les méthodes `additionne` et `soustrait` sont intéressantes. Elles sont des accesseurs car elles ont pour résultat un nouvel objet polynôme ; elles ne modifient donc pas leurs deux arguments.

La méthode `toString` illustre l'utilisation des chaînes de caractères en Java. Cette méthode a pour rôle de retourner une chaîne de caractère représentant au mieux l'objet. Elle est utilisée pour des besoins d'affichage à l'écran. Son nom anglais vient du fait qu'il s'agit d'un comportement défini par la classe `Object` en Java et auquel tous les objets doivent se conformer. En définissant cette méthode dans `Poly2`, on redéfinit en fait la méthode standard de la classe `Object` pour en introduire une qui es spécifique aux objets polynômes de degré 2.

Son corps produit une chaîne en utilisant la concaténation. Rappelons que l'opérateur '+' sur les chaînes a pour signification la concaténation des chaînes, avec conversion de type si l'un des arguments n'est pas une chaîne de caractères. Si on applique la concaténation entre une chaîne et une valeur d'un autre type, cette valeur est d'abord convertie en chaîne avant la concaténation. Ainsi, si la variable `valeur` est entière et vaut 2, l'expression Java :

```
"Le résultat est : " + valeur
```

convertit le contenu de la variable `valeur`, 2, en chaîne de caractères, c'est-à-dire `"2"`, puis concatène les deux chaines pour donner `"Le résultat est : 2"`.

3.5.4 Création, référence, accès et envoi de message

La création d'un objet en Java se fait en utilisant l'expression primitive `new`. Comme nous l'avons décrit précédemment, la création implique également l'initialisation de l'objet créé. L'utilisation de l'expression primitive `new` est illustrée dans les méthodes `additionne` et `soustrait` de la classe `Poly2`. La forme générale en est :

```
new <nom-classe>(<liste-paramètres-initialisation>)
```

La création d'un objet implique l'allocation de l'espace nécessaire pour contenir les valeurs de ses variables d'instance, l'initialisation de ces variables par une méthode d'initialisation et l'attribution d'un identificateur d'objet unique. Rappelons que la sélection de la méthode d'initialisation se fait par la liste des paramètres. La classe `Poly2` offre deux méthodes d'initialisation : une méthode sans paramètre et une méthode à trois paramètres de type `double`.

La primitive `new` de Java prend la forme d'une expression, et à ce titre elle doit avoir un résultat : l'*identificateur d'objet unique* qui va servir par la suite à référencer cet objet.

Identificateur d'objet ici est un bien grand mot : il s'agit en fait d'un nombre binaire unique forgé de façon interne par Java. Le résultat de l'expression `new` est donc l'identificateur de l'objet créé, et son type est la classe de laquelle l'objet est instance.

L'identificateur d'objet est utilisé en Java dans les expressions permettant d'accéder aux variables de l'objet et dans les expression permettant d'appeler les méthodes de l'objet. Comme toute autre valeur, un identificateur d'objet peut être placé dans une variable. Si on crée un objet de type `Poly2`, on peut mettre son identificateur dans un variable de type `Poly2` :

```
Poly2 p1 ;

p1 = new Poly2() ;
```

La variable `p1` contient alors un identificateur d'objet de type `Poly2` dont les trois coefficients ont pour valeur 0. Un objet ayant le droit d'accéder aux valeurs d'un polynôme qui possède cette variable `p1` utilise une *notation pointée* pour désigner la variable dont il veut la valeur, comme dans l'expression :

```
p1.coef2
```

Pour activer une méthode sur `p1`, il utilise également la notation pointée, comme dans l'énoncé :

```
p1.changeCoef1(10) ;
```

L'exécution de cet énoncé va activer la méthode `changeCoef1` sur l'objet `p1` et se solder par la modification de la variable `coef1` de l'objet référencé par `p1`.

Une méthode dont le type de retour est `void` est assimilable à une procédure et son appel est considéré comme un énoncé (qui ne retourne pas de résultat mais fait des effets de bords sur l'état). Par contre, une méthode qui a un type de retour différent de `void` doit plutôt être vue comme une fonction et donc être utilisée dans une expression. Par exemple, dans la classe `Poly2`, la méthode `discriminant` retourne un réel et peut être utilisée dans une expression relationnelle comme :

```
p1.discriminant() >= 0.0
```

Cette expression envoie le message `discriminant()` au receveur `p1`. L'activation de la méthode `discriminant` sur l'objet `p1` va avoir pour effet de calculer le discriminant du polynôme de degré 2 représenté par `p1` et d'en retourner la valeur. Cette valeur est alors utilisée dans l'expression relationnelle, et donc comparée à 0.0.

La pseudo-variable `this`

Il est parfois utile de pouvoir référer dans une méthode à l'objet receveur du message que la méthode est en train de traiter. Java permet de référencer le receveur en utilisant la pseudo-variable `this`. On dit que `this` est une pseudo-variable dans la mesure où on l'utilise comme une autre variable dans une expression, mais il n'est pas nécessaire de déclarer la varaible `thos` ; elle existe par défaut. De plus, il est interdit d'utiliser `this` comme cible d'une affectation. La méthode `inverseAdditif` de la classe `Poly2` illustre l'utilisation de cette pseudo-variable.

3.5.5 L'héritage en Java

En Java, la syntaxe permettant de déclarer une classe comme étant une sous-classe d'une autre classe utilise le mot-clé `extends` suivi du nom de la superclasse dans l'entête de la déclaration de la sous-classe. Par exemple, on peut définir une classe `Figure` :

```
class Figure {
   ...
   public void trace() {...}
   public abstract double perimetre() ;
   ...
}
```

à partir de laquelle il est possible de définir par extension la sous-classe `Ellipse` et une sous-classe d'`Ellipse` appelée `Cercle` :

```
class Ellipse extends Figure {
   ...
   public void trace() {...}
   public double perimetre() {...}
   ...
}

class Cercle extends Ellipse {
   ...
   public void trace() {...}
   public double perimetre() {...}
   ...
}
```

La classe `Ellipse` est alors sous-classe de la classe `Figure`, ce qui induit l'héritage par la première des variables d'instance et des méthodes de la seconde. Le lien d'héritage entre classes crée ce que l'on appelle la hiérarchie d'héritage du programme. En Java, cette hiérarchie possède une unique racine qui est la classe `Object` ; une classe qui est déclarée sans la clause `extends` est par défaut sous-classe de la classe `Object`. La classe `Object` joue donc un rôle important en définissant les comportements appliquables à tous les objets.

Pour redéfinir une méthode de la classe `Figure`, la classe `Ellipse` n'a qu'à proposer une nouvelle définition de cette méthode en utilisant le même nom, le même nombre et les mêmes types de paramètres formels ainsi que le même type de résultat. Un exemple en est donné ici par la méthode `trace` définie par `Figure`, mais redéfinie par `Ellipse` puis par `Cercle`.

Parce que les classe `Ellipse` et `Cercle` sont sous-classes de la classe `Figure`, le principe de substitution s'applique et il est possible de faire les déclarations, instantiations et affectations suivantes :

```
Figure f1, f2 ;

f1 = new Ellipse() ;
f2 = new Cercle() ;
```

Les variables `f1` et `f2` sont déclarées de type `Figure`, mais puisque partout où un objet de type `Figure` est attendu le principe de substitution nous permet d'y fournir un objet de type `Ellipse` ou `Cercle`, ces deux affectations sont parfaitement légales en Java. Cependant, si on envoie le message `trace` aux objets contenus dans `f1` et `f2`, les deux n'opterons pas pour le même comportement pour répondre à ce message. La méthode employée pour répondre à ce message dépend de l'objet receveur (d'où la notion de liaison tardive). Le message `f1.trace()` va entraîner l'exécution de la méthode `trace` de la classe `Ellipse`, parce que l'objet contenu dans `f1` est une instance de la classe `Ellipse`. Par contre, le message `f2.trace()` entraînera pour sa part l'exécution de la méthode `trace` de la classe `Cercle` parce que l'objet contenu dans `f2` est instance de cette dernière.

L'inconvénient de déclarer les variables `f1` et `f2` de type `Figure` pour y placer des objets de type `Ellipse` ou `Cercle` est par contre que seuls les messages reconnus par `Figure` peuvent être envoyés aux objets contenus dans `f1` et `f2`. Cette contrainte permet à Java d'assurer que le message sera bien reconnu par le receveur à l'exécution. En effet, si on envoyait un message reconnu par la classe `Cercle` à `f2`, tout se passerait bien,mais si on l'envoyait à `f1`, ce message ne serait pas reconnu puisque le contenu de `f1` est une instance d'`Ellipse`.

L'héritage et la redéfinition de méthode induit aussi la liaison tardive d'un message à une méthode par l'utilisation du mot-clé `this` comme receveur. Considérons une méthode `m` de la classe `Ellipse` contenant un envoi de message de la forme :

```
this.trace() ;
```

La méthode employée pour répondre à ce message dépend également de l'objet qui exécute cette méthode `m`. Si l'objet exécutant `m` est une instance de la classe `Ellipse`, comme celui contenu dans `f1`, alors c'est la méthode `trace` de la classe `Ellipse` qui s'applique. Si par contre l'objet exécutant `m` est une instance de la classe `Cercle`, comme celui contenu dans `f2`, c'est la méthode `trace` de `Cercle` qui sera appliquée.

Le mot-clé `super` peut être utilisé dans une méthode de la classe `Cercle`, dont la méthode `trace`, pour appeler une méthode définie dans la classe `Ellipse`, comme par exemple `trace`. Ainsi, le message `super.trace()` placé dans la méthode `trace` de la classe `Cercle` entraînera l'exécution de la méthode `trace` de la classe `Ellipse`. Le mot-clé `super` permet ainsi de ne pas simplement redéfinir mais plutôt de compléter une définition de méthode en rappelant la définition précedente puis en ajoutant ce qui est spécifique à la redéfinition.

Une superclasse peut aussi imposer à toutes ses sous-classes de définir une certaine méthode, sans nécessairement en donner elle-même une définition. C'est le cas dans notre exemple pour la méthode `perimetre` de la classe `Figure` qui est déclarée avec le modificateur `abstract`. Ce modificateur a trois conséquences :

1. La classe `Figure` n'a pas à définir le corps de cette méthode ; elle se contente de dire par cette déclaration que toute figure doit savoir retourner son périmètre lorsqu'on lui envoie le message `perimetre`.
2. Il est interdit de créer un objet instance de la classe `Figure`, puisque ce dernier ne saurait pas répondre au message `perimetre`. La méthode correspondante étant abstraite, il n'y aurait pas de corps à exécuter le cas échéant. N'étant pas instantiable, on dit que la classe `Figure` est *abstraite*.
3. Toute sous-classe de la classe `Figure` doit inclure une définition complète de la méthode `perimetre`, sous peine d'être elle-même à nouveau abstraite.

Enfin, nonobstant la règle générale, il est possible en Java d'interdire la redéfinition d'une méthode par les sous-classes d'une classe. Pour cela, il suffit d'utiliser le qualifieur `final` lors de la définition de la méthode dans la superclasse. Java refusera de compiler une sous-classe qui tente de redéfinir une méthode qualifiée de finale dans une de ses superclasses. Il est souvent tentant pour le programmeur d'utiliser le qualifieur `final` pour se «prémunir» contre les éventuelles sous-classes, mais en réalité cecla va à l'encontre de la philosophie objet qui est fondée sur la réutilisation de classes existantes, réutilisation qui suppose presque toujours des ajustements par rapport à la classe réutilisée. Il ne faut donc utiliser un qualifieur `final` que dans des cas de force majeure.

3.5.6 Documentation embarquée avec *Javadoc*

Comme plusieurs autres langages récents (Eiffel, Perl), le langage Java prévoit un mécanisme de documentation technique de ses programmes. Ce mécanisme basé sur une technique d'embarquement s'appuie sur :

1. une structuration des textes de programmes (sources) documentés selon une syntaxe précise,
2. un outil d'extraction automatique appelé `javadoc`.

Commentaires et documentation embarquée

Le langage Java sait gérer deux types de commentaires :

1. les commentaires de fin de ligne introduits par '`//`' qui s'étendent jusqu'à la fin de la ligne, et
2. les commentaires sur plusieurs lignes délimités par les signes `/*` et `*/`.

Par exemple, on peut écrire[14] :

```
/*
 *   Ceci est un bloc de commentaires
 *   sur plusieurs lignes
 */
int i ;                    // Ceci est un commentaire de fin de ligne
```

Les commentaires dans un programme peuvent servir plusieurs finalités. Certains commentaires cherchent à révéler l'intention du programmeur qui autrement serait implicite dans la séquence d'opérations apparaissant dans le programme. Ce genre de commentaires, dit d'implantation, s'adressent au programmeur qui veut comprendre la mise en œuvre d'une classe, la plupart du temps dans le but de la retoucher.

D'autres commentaires expliquent comment utiliser les méthodes et s'adressent à l'utilisateur de la classe. Ce second type de commentaires se répète généralement dans le programme et dans la documentation d'utilisation classique qui est produite en parallèle avec le développement des programmes. Pour éviter de répéter deux fois la même information d'utilisation, ce qui est souvent sujet à erreur, oubli, ou encore peut ne pas être en parfaite synchronisation lors de l'évolution du programme, la documentation d'utilisation est laissée dans les programmes

[14] Sans etre rendu obligatoire par la syntaxe, l'alignement d'étoiles à gauche du commentaire est vivement recommandé par les règles de style.

sous forme de commentaires d'où elle est extraite par des outils dits d'extraction qui vont ensuite mettre en forme la documentation d'utilisation à partir de commentaires spécifiques mis dans le programme.

En Java, ces commentaires de documentation sont une variété particulière de commentaires qui commencent par le signe `/**`. Le format de ces commentaires est défini par une norme qui permet ensuite une extraction automatique avec un outil appelé `javadoc`. De manière générale, tout commentaire Javadoc accompagne une déclaration d'un élément du programme : une classe, une constante, une variable, ou une méthode. La structure standard d'une déclaration documentée en Javadoc est la suivante :

1. un commentaire de documentation d'interface, suivi
2. du code de la déclaration, suivi
3. d'un commentaire sur les choix et stratégies d'implantation (non-extractible).

Par exemple, la déclaration documentée d'une méthode se présente comme suit :

```
// ------------------------------------------------------
/**
 * calculeRacine1   : calcule la première racine
 *<P>
 *     Opération valide uniquement si le discriminant est positif ou nul
 *</P>
 *
 * @return double   : valeur de la première racine
 *
 *<P><STRONG>Contrat</STRONG>
 *
 * @pre     calculeDiscriminant() >= 0.0
 * @post    Math.abs(evalue(return)) < 0.0000001
 */
public double       calculeRacine1(
    )
{
    return (-coef1 + Math.sqrt(calculeDiscriminant()))/ (2.0 * coef2) ;
}
// ---------------------------------------- calculeRacine1
```

L'utilitaire `javadoc` extrait les commentaires de documentation et les informations utiles sur déclarations et compose des pages d'information en format HTML (à raison d'une page par classe). La documentation peut alors être parcourue avec un navigateur internet.

Syntaxe

Il y a trois types de documentation qui correspondent aux éléments du langage que le commentaire de documentation précède : classe, variable, méthode. Chaque niveau de documentation doit apparaître juste avant la déclaration qu'il documente ; la structure générale d'une classe documentée en Javadoc est donc :

```
/**   Documentation de classe          */
public class        docTest {

    /**   Documentation de variable     */
    public int      i ;

    ...
```

Étiquettes	Niveau syntaxique	Rôle
`@author`	classe	identification de l'auteur
`@deprecated`	méthode	indique qu'une méthode ne doit plus etre utilisée
`@exception`	méthode	documentation des exceptions
`@param`	méthode	décrit le rôle d'un argument d'appel
`@return`	méthode	décrit le résultat retourné par la méthode
`@see`	classe, variable, méthode	établit un lien hypertexte vers un paragraphe ou une autre page (la syntaxe du lien est la meme qu'en HTML)
`@since`	classe	indique une date de mise à jour
`@version`	classe	donne l'information de versionnement de la classe

FIG. 3.13 – Liste des étiquettes Javadoc

```
    /**   Documentation de méthode      */
    public void     f () {...}
}
```

Par défaut, l'outil `javadoc` n'extrait que les informations d'accès `public` et `protected`, mais des options[15] permettent de modifier ce comportement.

Structuration interne

Les commentaires de documentation peuvent être structurés par deux voies complémentaires :

1. des étiquettes de documentation prévus par Javadoc.
2. des balises HTML pour la mise en forme,

Les possibilités de mise en page (que ce soit de forme ou de fond) restent cependant très faibles et difficiles à contrôler. L'objectif n'est pas de faire du livre d'art !

Étiquettes de documentation

Les étiquettes de documentation sont des mots qui commencent par le caractère '@' ; ils sont le plus souvent spécifiques pour un niveau de documentation (classe, variable, méthode). La figure 3.13 donne la liste des étiquettes `javadoc`, le niveau syntaxique auquel elles s'appliquent et un explication bréve de l'information ainsi étiquetée.

HTML embarqué

Toute balise HTML placée dans le texte d'un commentaire de documentation est recopiée telle quelle dans la page HTML extraite. Les balises les plus utilisées dans cet usage servent à placer du texte tel quel :

[15] de la ligne de commande.

```
/**
 *   <PRE>
 *      exemple de code
 *   </PRE>
 */
```

ou à gérer des structures de listes :

```
/**
 *   <OL>
 *   <LI> premier item,
 *   <LI> second item,
 *   <LI> troisième ...
 *   </OL>
 */
```

Pour ceux qui connaîtraient HTML, certaine balises comme les commandes de sectionnement (`<H1>`, `<H2>`, ...) ou le tracé de ligne (`<HR>`) ne doivent pas être utilisées car `javadoc` s'en sert pour structurer les documents ; il y aurait donc conflit d'utilisation sur ces fonctionnalités.

Commande `javadoc`

L'usage de la commande `javadoc` est détaillé dans sa page de manuel (faire '`man javadoc`'). Vous êtes priés de vous y référer pour tout complément d'information. L'argument de `javadoc` est un nom de *package* (répertoire) ou un ensemble de sources (fichiers '`.java`'). Les classes dont on extrait la documentation doivent avoir été compilées (recherche des noms de variables et méthodes, héritage, etc).

La commande `javadoc` produit une page HTML par classe, mais aussi une page de liens pour le *package* et une table d'index alphabétique pour tous les noms de variables et de méthodes.

Usage des commentaires

Les trois types de commentaires doivent etre employés avec discernement de manière à contribuer à l'organisation et à la lisibilité des programmes.

Les commentaires de documentation ('`/** ... */`') doivent précéder directement les déclarations des éléments documentés. Ils doivent contenir les informations nécessaire à l'utilisation de l'élément décrit (documentation d'interface).

Les blocs de commentaires ('`/* ... */`') sont utilisés dans deux cas :

1. En début de fichier, comme entête d'identification normalisée du fichier source.
2. Juste après une ligne de déclaration d'élément (classe, variable ou méthode) pour en préciser les modalités d'implantation.

Les commentaires de fin de ligne ('`// ...`') doivent attirer l'attention sur l'utilisation d'une technique de programmation délicate et dont l'opacité risquerait de nuire à la compréhension

des programmes.[16] En aucun cas les commentaires doivent se substituer à l'écriture de programmes clairs, utilisant des noms de variables significatifs. Nous utilisons aussi les commentaires de fin de ligne pour faire ressortir les fins de structures syntaxiques (`}`), de manière à fournir des repères visuels au lecteur du programme.

Comme on peut le constater, cette approche privilégie la documentation par anticipation : le commentaire explicatif est toujours positionné dans le source *avant* le code qu'il documente. Cette disposition indique clairement au développeur qu'il doit rédiger ces commentaires avant le programme lui-même.

3.6 Exercices

3.6.1. Considérez la classe `Poly2` définie à la figure 3.11 et écrivez une séquence d'énoncés qui créent les polynômes $2x^2 + 3x + 6$ et $5x^2 - x - 4$, obtient le polynôme résultant de l'addition des deux premiers et imprime les deux racines de ce polynôme.

3.6.2. Définir une classe `Complexe` représentant les nombres complexes. Votre classe devrait contenir des constructeurs, des accesseurs aux parties réelle r et imaginaire i du nombre, et enfin une méthode `toString` donnant une représentation en chaîne de caractères du nombre selon le format (r, i).

3.6.3. Modifiez la classe `Poly2` de manière à ce que les racines du polynôme soient des nombres complexes en utilisant la classe `Complexe` de l'exercice précédent.

3.6.4. Ajoutez à la classe `Complexe` les opérations d'addition, de soustraction et de multiplication.

3.6.5. Introduisez dans la classe `Rectangle` deux méthodes `modifieHauteur` et `modifieLargeur`, en prenant soin de bien définir les contrats pour ces deux méthodes.

3.6.6. Écrivez une classe `Carre` héritant de la classe `Rectangle`. Vos carrés doivent répondre exactement aux mêmes messages que les instances de la classe `Rectangle`, y compris les deux méthodes de l'exercice précédent. Prenez soin aux assertions pour restreindre la représentation et les opérations sur les rectangles à donner des carrés.

3.6.7. Écrivez la classe `Tortue` représentant une tortue logo, en prenant soin de définir l'invariant structurel et les pré- et postconditions, sachant que la tortue doit évoluer dans un écran de 800 en hauteur et 600 en largeur, et que les quatre caps possibles sont nord (90.0), sud (270), est (0.0) et ouest (180.0). Votre classe `Tortue` devra contenir les méthodes :

1. une méthode d'initialisation sans arguments qui place la tortue au milieu de l'écran, cap au nord ;
2. une méthode d'initialisation à trois arguments spécifiant un position en x, une position en y et un cap ;
3. `avance` prenant un nombre de pas en paramètre et faisant avancer la tortue du nombre de pas prescrit dans la direction de son cap courant ;
4. `recule` prenant un nombre de pas en paramètre et faisant reculer la tortue du nombre de pas prescrit dans la direction inverse de son cap courant ;

[16] En particulier, il est inutile de répéter sous forme de commentaire ce qui est évident par la lecture des énoncés du programme. Un commentaire du genre :

```
x = x + 1 ;              // on range le résultat de x + 1 dans x
```

est non seulement totalement inutile mais il distrait l'attention du programmeur cherchant à comprendre votre programme.

5. `droite` faisant tourner la tortue de 90 degrés vers la droite ;
6. `gauche` faisant tourner la tortue de 90 degrés vers la gauche ; et,
7. `toString` retournant une chaîne de caractères représentant la tortue sous la forme `"[Tortue à (400, 300), cap à 90.0]"`.

3.6.8. Écrivez une classe `ReineTortue` représentant une tortue logo qui est capable de se déplacer non seulement dans les quatres directions nord, sud est et ouest, mais aussi selon les directions diagonales nord-est, nord-ouest, sud-est et sud-ouest, à la manière d'une reine au jeu d'échec. Cette classe définit les mémes méthodes que la classe `Tortue`, mais les méthodes `gauche` et `droite` font tourner la tortue de 45 degrés plutôt que 90. Prenez soin de définir l'invariant structurel de la classe ainsi que les pré- et post-conditions des méthodes.

3.6.9. Considérez les classes `Tortue` et `ReineTortue` précédente. Peut-on les définir différentiellement en utilisant l'héritage ? Si oui, sachant que l'invariant de la superclasse s'applique à la sous-classe, que la précondition d'une méthode redéfinie dans la sous-classe doit être égale ou plus faible que la méthode de la superclasse, et que la postcondition doit être égale ou plus forte, laquelle des deux classes `Tortue` et `ReineTortue` devrait être la superclasse de l'autre ?

3.6.10. Une classe abstraite en Java est une classe dont la définition est incomplète par ce que certaines de ses méthodes dites abstraites n'ont pas d'implantation, c'est-à-dire pas de corps mais seulement une signature (un type de retour, un nom et une liste de paramètres formels). Une classe abstraite ne peut en conséquent avoir d'instance. On utilise les classes abstraites pour définir une interface commune à un ensemble de sous-classes concrètes. Pour devenir concrètes, les sous-classes doivent en effet donner une définition complète de toutes les méthodes abstraites dont elles héritent. Le nom de la classe abstraite pourra être utilisé comme type de variables qui pourront contenir des instances de n'importe laquelle des sous-classes concrètes. L'utilisation de ces variables dans des appels de méthodes abstraites sera correctement typé, car l'assurance est donnée qu'une instance d'une sous-classe concrète aura une méthode correspondante.
Syntaxiquement, une classe abstraite est définie en faisant apparaître le mot-clé `abstract` avant le mot-clé `class` :

```
<qualificatif> abstract class     <nom>
{
   <corps de la classe>
}
```

Une méthode abstraite est définie en donnant ses qualificatifs, son type de résultat, son nom et sa liste de paramètres formels entre parenthèses suivi du point-virgule terminant l'énoncé de déclaration :

```
<qualificatif> <type>    <nom>(<paramètres formels>) ;
```

Les nombres complexes peuvent être représentés en coordonnées cartésiennes $x + iy$ et en coordonnées polaires $\rho e^{i\theta}$. La conversion entre les deux représentations se fait en utilisant les identités suivantes :

$$\begin{aligned} \rho &= \sqrt{x^2 + y^2} \\ \theta &= atan(y/x) \\ x &= \rho\cos(\theta) \\ y &= \rho\sin(\theta) \end{aligned}$$

Modifiez la classe `Complexe` pour devenir la classe abstraite des deux sous-classes concrètes `Cartesien` et `Polaire`. (Note : les méthodes de classe `Math.atan(double x)`, `Math.cos(double x)` et `Math.sin(double x)` peuvent être utilisées pour calculer l'arctangente, le cosinus et le sinus. Ces classes utilisent des angles en radians.)

Chapitre 4

Flôt de contrôle

Outre les données, un programme, nous l'avons vu, est un ensemble ordonné et formalisé des opérations nécessaires et suffisantes pour obtenir un résultat. En programmation impérative, le texte de programme contient des énoncés organisés selon un certain ordre. L'exécution du programme consiste en l'exécution de ses énoncés sur des données dites d'entrée, ce qui produira des résultats ou des données dites de sortie. Les énoncés vus jusqu'ici, affectation, séquence, appel de méthode (envoi de message), donnent peu de possibilités d'adapter l'exécution d'un programme aux données d'entrée ; un programme qui ne serait constitué que de ces types d'énoncés exécuterait toujours la même séquence d'énoncés, peu importe les données intiales qui lui seraient fournies.

L'objectif du présent chapitre est d'introduire les énoncés permettant de contrôler la séquence des opérations réalisées, ou *flôt de contrôle*, en fonction des données. Par aiguillage sur une séquence d'énoncés ou une autre, par répétition d'une certaine séquence d'énoncés, par exemple, l'exécution d'un programme se traduira par l'exécution de séquences d'énoncés différentes selon les données d'entrée. Comme nous l'avons déjà vu, la séquence complète des énoncés effectivement exécutés est appelée *fil d'exécution*.

4.1 La séquence

La composition séquentielle d'énoncés est l'un des moyens offerts en programmation impérative et par objets pour construire des programmes plus grands à partir de segments de programmes plus petits. Soient $\texttt{E}_1$ et $\texttt{E}_2$ deux énoncés, alors $\texttt{E}_1$ `;` $\texttt{E}_2$ est aussi un énoncé obtenu par composition séquentielle des énoncés $\texttt{E}_1$ et $\texttt{E}_2$. L'opérateur de composition '`;`' est utilisé pour joindre deux énoncés.[1] Cet énoncé composé est exécuté en exécutant d'abord $\texttt{E}_1$ puis $\texttt{E}_2$. Dans cet énoncé composé, on distingue trois points d'observation, où l'état mémoire du système peut être observé : le point initial précédant $\texttt{E}_1$, le point intermédiaire entre les deux énoncés, et le point final après $\texttt{E}_2$.[2]

Exemple 29 La séquence d'énoncés suivante fait partie d'un segment de programme cher-

[1] Nous allons voir dans la partie «Éléments de savoir-faire» que la syntaxe de Java utilise le point-virgule comme un terminateur d'énoncés plutôt que comme un opérateur de composition.

[2] Rappelons que le concept de point d'observation sert à suivre l'exécution d'un programme en l'instrumentant par des énoncés d'impression qui permettent de connaître l'état de tout ou partie des variables. Il sert également à raisonner sur les programmes en l'annotant par des assertions, ou expressions logiques, décrivant les ensembles des états possibles (ou acceptables) en ces points.

chant à accumuler dans une variable `s` la somme des entiers 1 à `i` :

```
s = s + i ;
i = i + 1 ;
```

□

Raisonnement sur les séquences avec les assertions

En tant qu'énoncés, chacun des énoncés d'une séquence agit sur l'état mémoire du système, en prenant un état précédent et en le transformant en un état suivant. Nous avons utilisé les assertions posées sur les points d'observation pour décrire les états mémoire et raisonner sur l'effet des énoncés. Si l'énoncé $\mathtt{E}_1$ respecte les assertions :

$\{pre_{\mathtt{E}_1}\}$
$\mathtt{E}_1$;
$\{post_{\mathtt{E}_1}\}$

ce qui veut dire que si l'énoncé $\mathtt{E}_1$ est exécuté dans un état respectant l'assertion $\{pre_{\mathtt{E}_1}\}$, alors il termine son exécution en produisant un état respectant l'assertion $\{post_{\mathtt{E}_1}\}$. Et si l'énoncé $\mathtt{E}_2$ respecte les assertions :

$\{pre_{\mathtt{E}_2}\}$
$\mathtt{E}_2$;
$\{post_{\mathtt{E}_2}\}$

alors la composition $\mathtt{E}_1$; $\mathtt{E}_2$; nécessite de s'assurer que l'état intermédiaire produit par $\mathtt{E}_1$, qui respecte sa postcondition $\{post_{\mathtt{E}_1}\}$, respecte également la précondition de l'énoncé $\mathtt{E}_2$, c'est-à-dire $\{pre_{\mathtt{E}_2}\}$.

Bien sûr, l'état au point d'observation intermédiaire respectera les deux assertions si elles sont les mêmes, c'est-à-dire si $\{post_{\mathtt{E}_1}\}$ est la même assertion que $\{pre_{\mathtt{E}_2}\}$. Mais cette condition est trop forte. En fait, si on utilise des connaissances élémentaires de logique, un état e qui respectent une assertion A_1, respectera aussi une assertion A_2 si on peut montrer que A_2 découle de A_1, c'est-à-dire si on peut montrer que $A_1 \Rightarrow A_2$.

Exemple 30 Reprenons notre exemple précédent. L'énoncé `s = s + i` respecte les assertions suivantes[3] :

$\{A_1 : \mathtt{i} \geq 0 \wedge \mathtt{s} + \mathtt{i} = 0 + 1 + 2 + \ldots + \mathtt{i}\}$

```
s = s + i ;
```

$\{A_2 : \mathtt{i} \geq 0 \wedge \mathtt{s} = 0 + 1 + 2 + \ldots + \mathtt{i}\}$

On obtient la précondition A_1 de la postcondition A_2 en substituant simplement `s + i` à `s`. L'énoncé `i = i + 1` pour sa part respecte les assertions suivantes :

$\{A_3 : \mathtt{i} \geq 0 \wedge \mathtt{s} = 0 + 1 + 2 + \ldots + \mathtt{i}\}$

```
i = i + 1 ;
```

$\{A_4 : \mathtt{i} > 0 \wedge \mathtt{s} = 0 + 1 + 2 + \ldots + (\mathtt{i} - 1)\}$

On comprend mieux le passage de la précondition à la postcondition si on introduit une assertion supplémentaire en précondition : $A'_3 : \{\mathtt{i} + 1 > 0 \wedge \mathtt{s} = 0 + 1 + 2 + \ldots + (\mathtt{i} + 1 - 1)\}$.

[3]Ces assertions sont choisies en fonction du problème que l'on cherche à résoudre, c'est-à-dire accumuler dans `s` la somme des entiers 1 à `i`. Nous introduisons également un étiquetage des assertions pour une référence plus facile dans le texte.

A'_3 est une conséquence immédiate de A_3. Or, si on applique la règle de transformation vue à la section 2.1, on arrive à A'_3 par substitution de `i` + 1 à `i` dans A_4.

Pour composer les deux énoncés, il faut s'assurer que A_3 découle de A_2 or, A_3 est égale à A_2. □

L'«opérateur» de composition ' ;' peut servir à composer d'autres séquences d'énoncés, comme par exemple la composition de `E1 ;` avec `E2 ; E3 ;` pour donner `E1 ; (E2 ; E3) ;`. Opérationnellement, l'exécution de cette nouvelle séquence d'énoncés débute par l'exécution de `E1 ;` suivie de l'exécution de `E2 ; E3 ;`.

La composition d'énoncés est associative, c'est-à-dire que la composition de `E1 ;` avec `E2 ; E3` donne le même résultat que la composition de `E1 ; E2 ;` avec `E3 ;`. En résumé, on a

$$\texttt{E1 ; (E2 ; E3 ;)} \equiv \texttt{(E1 ; E2 ;) E3 ;}$$

Ce résultat peut être démontré en utilisant les assertions sur les états intermédiaires des deux énoncés composés.

Comme il a été noté à la section 2.1, les assertions intermédiaires dans un énoncé séquentiel servent à raisonner sur l'exécution de la séquence elle-même. Une fois cette exécution maîtrisée, on peut se contenter de la précondition initiale et de la postcondition finale pour raisonner sur la séquence vue comme un seul énoncé. On dit alors que l'on abstrait le calcul réalisé par la séquence par son effet global de transformation de l'état initial de la séquence en un état final.

4.2 Rupture de séquence et flôt de contrôle dirigé par la syntaxe

La séquence permet de construire des programmes simples où les énoncés exécutés sont toujours les mêmes, peu importe la situation dans laquelle le programme est exécuté, et sont en nombre fini et connu à l'avance. Des programmes qui auraient de telles limitations seraient de relativement peu d'utilité en pratique. Si on se reporte aux instructions de la machine physique, des situations aussi simples que le calcul de la valeur absolue d'un entier relatif nécessitent de faire des opérations différentes selon les données présentées en entrée au programme, et dans ce cas particulier selon que l'entier est négatif ou positif. De même, le calcul de la somme des n premiers entiers nécessite l'exécution d'un nombre d'instructions différent selon la valeur de n.

La rupture de séquence devient nécessaire lorsque :

1. Certaines instructions doivent être exécutées seulement dans certains cas.
   Exemple 31 Situations courantes d'exécution conditionnelle :
   - *Si le moule n'est pas enduit de téflon, le graisser.*
   - *Ajouter du sel au besoin.*

   □

2. Certaines instructions doivent être exécutées plusieurs fois de suite jusqu'à obtenir un certain état.
   Exemple 32 Situations courantes d'exécution répétitive :
   - *Appuyer sur le bouton autant de fois que nécessaire pour obtenir la couleur désirée.*
   - *Brasser jusqu'à obtenir une consistance lisse.*

   □

3. Certains groupes d'instructions peuvent être exécutés à la demande, comme une sous-tâche (appel de méthode).

   Exemple 33 Situations courantes d'exécution de sous-tâches :
   - *Préparer une sauce béchamel selon la recette de la page 52.*
   - *Avant tout, installez l'appareil selon la procédure 'Installation' de votre manuel d'utilisation.*

□

Une rupture de séquence est aussi produite par l'appel de méthode qui interrompt la séquence d'énoncés en cours d'exécution pour aller exécuter la méthode, pour revenir ensuite continuer l'exécution de la séquence l'endroit suivant immédiatement l'appel de méthode.

Les instructions des processeurs physiques comprennent des instructions de branchement conditionnel et inconditionnel. Ces instructions, suppléées par quelques instructions de gestion des registres, permettent d'écrire les ruptures de séquence nécessaires pour produire l'exécution conditionnelle ou répétitive d'une séquence d'instructions. Elles exhibent cependant une facheuse permissivité en admettant la production de n'importe quel fil d'exécution pour un programme. Rien n'empêche effectivement d'utiliser les instructions de branchement pour sauter un peu partout dans le programme, ce qui s'est soldé dans certains programmes par un flôt de contrôle assimilé à une platée de spaghettis ; suivre le flôt de contrôle dans ces programmes peut se comparer à suivre des yeux un spaghetti à travers une assiette remplie de pâtes ! Une autre comparaison plus moderne est celle des histoires *dont vous êtes le héros* qui proposent des choix de poursuite de l'histoire qui renvoient le lecteur en différents points du livre en fonction de leur décision sur la suite à donner à l'histoire ; le chemin suivi par la séquence de lignes lues dans ces histoires peut devenir fort complexe.

La compréhension des programmes nécessite une discipline dans le flôt de contrôle. Cette idée a été défendue par un courant de pensée favorable à la structuration des programmes dans les années '70 qui a pris pour nom *programmation structurée*. L'idée mise en avant par cette approche consistait à avoir un flôt de contrôle le plus séquentiel possible, puisque c'est le plus simple à comprendre. Certes, il faut des ruptures de séquences, mais celles-ci sont produites par des énoncés spécifiques qui respectent tous un principe de base important :

Tout énoncé a un et un seul point d'entrée de même qu'un et un seul point de sortie.

C'est-à-dire que le flôt de contrôle arrive à un énoncé toujours par un unique point d'entrée et que si cet énoncé produit une rupture de séquence en dirigeant le flôt de contrôle dans des directions divergentes, alors ces directions divergentes vont nécessairement se refondre dans un unique flôt de contrôle avant de franchir le point d'observation final de l'énoncé.

La figure 4.1 illustre un flôt de contrôle de type *spaghetti* pa rapport à un flôt de contrôle dirigé par la syntaxe. Dans le premier cas (a), le flôt de contrôle peut suivre des directions qui s'entremêlent et sont difficiles à suivre. Dans le deuxième cas (b), des blocs successifs se joignent pour former un flôt de contrôle globalement séquentiel bien que localement divergent. Outre le fait de proscrire le phénomène *spaghetti*, ce principe permet de raisonner sur les énoncés de rupture de séquence en identifiant des points d'observation clairs que l'on peut instrumenter pour produire des traces ou encore annoter par des assertions. Chaque énoncé a un point d'observation initial et un point d'observation final ; il possède généralement aussi des points d'observation intermédiaires. Ces points d'observation intermédiaires sont utiles pour comprendre le détail de l'effet d'un énoncé de rupture de séquence. Une fois cet effet bien maîtrisé cependant, pour comprendre l'effet global de l'énoncé, il suffit de raisonner sur son effet de transformation d'un état initial en un état final.

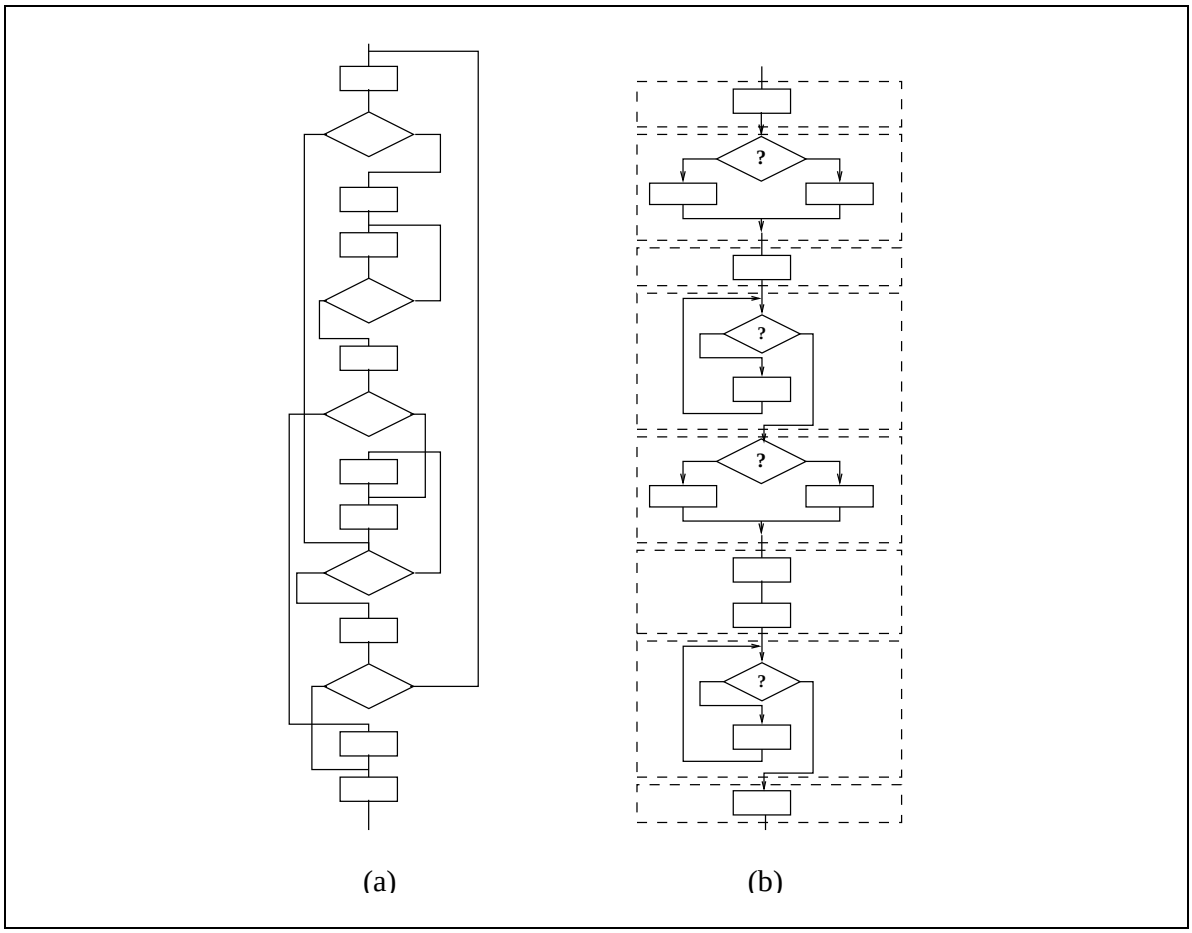


FIG. 4.1 – Flôt de contrôle de type *spaghetti* (a) en comparaison d'un flôt de contrôle dirigé par la syntaxe (b) globalement séquentiel bien que localement divergent.

L'utilisation d'énoncés, donc de constructions syntaxiques, pour structurer le flôt le contrôle d'un programme a donné naissance au concept appelé *flôt de contrôle dirigé par la syntaxe*. L'intérêt d'un flôt dirigé par la syntaxe est la production de fils d"exécution plus simple et plus discipliné, donc plus facile à lire et à comprendre.

4.3 Énoncés conditionnels

L'énoncé conditionnel le plus courant est l'énoncé d'alternative qui se présente sous une syntaxe ressemblant à :

$$si\ \texttt{C}\ alors\ \texttt{E}_1\ sinon\ \texttt{E}_2$$

où la partie `C` est une expression booléenne et les énoncés (possiblement composés) $\texttt{E}_1$ et $\texttt{E}_2$ sont respectivement exécutées selon que la condition `C` s'évalue à vrai ou à faux.

En Java, le patron syntaxique générale pour l'énoncé d'alternative est :

```
if <exp-condition> {
  <énoncé-si-vrai>
} else {
  <énoncé-si-faux>
```

```
}
```

Exemple 34 L'énoncé d'alternative suivant calcule la partie réelle et la partie imaginaire, si elle existe, d'une racine de l'équation du second degré $\mathtt{a}x^2 + \mathtt{b}x + \mathtt{c}$:

```
disc = b*b - 4*a*c ;
if (disc >= 0.0) {
  r = (-b + Math.sqrt(disc))/(2*a) ;
  i = 0.0 ;
} else {
  r = -b/(2*a) ;
  i = Math.sqrt(-disc)/(2*a) ;
}
```

□

L'énoncé d'alternative possède six points d'observation :

1. le point d'observation initial, avant l'énoncé,
2. le point d'observation en début de partie «alors»,
3. le point d'observation en fin de partie «alors»,
4. le point d'observation en début de partie «sinon»,
5. le point d'observation en fin de partie «sinon», et
6. le point d'observation final, après l'énoncé.

La figure 4.2 illustre graphiquement les différents chemins pouvant être suivis lors de l'exécution d'un énoncé d'alternative. La partie condition est illustrée par un losange. La branche de gauche à partir du losange illustre les énoncés exécutés si la condition s'évalue à vrai, alors que la branche de droite illustre ceux qui sont exécutés si la condition s'évalue à faux. En tant qu'énoncés possiblement composés, les parties «alors» et «sinon» peuvent évidemment comporter elles-mêmes des points d'observation (initiaux et finaux ainsi que des points d'observation intermédiaires).

Il est possible d'utiliser ces points d'observation pour produire une trace de l'exécution d'un énoncé d'alternative. Dans un programme Java, l'introduction d'énoncés d'impression en ces points d'observation fera produire cette race lors de l'exécution.

Exemple 35 Reprenons l'énoncé d'alternative précédent calculant les parties réelles et imaginaires d'une équation du second degré. La version instrumentée de cet énoncé est :

```
System.out.println("init        : a = " + a + ", b = " + b + ", c = " + c) ;
disc = b*b - 4*a*c ;
System.out.println("début       : a = " + a + ", b = " + b + ", c = " + c +
                   ", disc = " + disc) ;
if (disc >= 0.0) {
    System.out.println("début-alors : a = " + a + ", b = " + b + ", c = " + c +
                       ", disc = " + disc) ;
    r = (-b + Math.sqrt(disc))/(2*a) ;
    i = 0.0 ;
    System.out.println("fin-alors   : a = " + a + ", b = " + b + ", c = " + c +
                       ", disc = " + disc + ", r = " + r + ", i = " + i) ;
} else {
    System.out.println("début-sinon : a = " + a + ", b = " + b + ", c = " + c +
                       ", disc = " + disc) ;
    r = -b/(2*a) ;
    i = Math.sqrt(-disc)/(2*a) ;
    System.out.println("fin-sinon   : a = " + a + ", b = " + b + ", c = " + c +
```

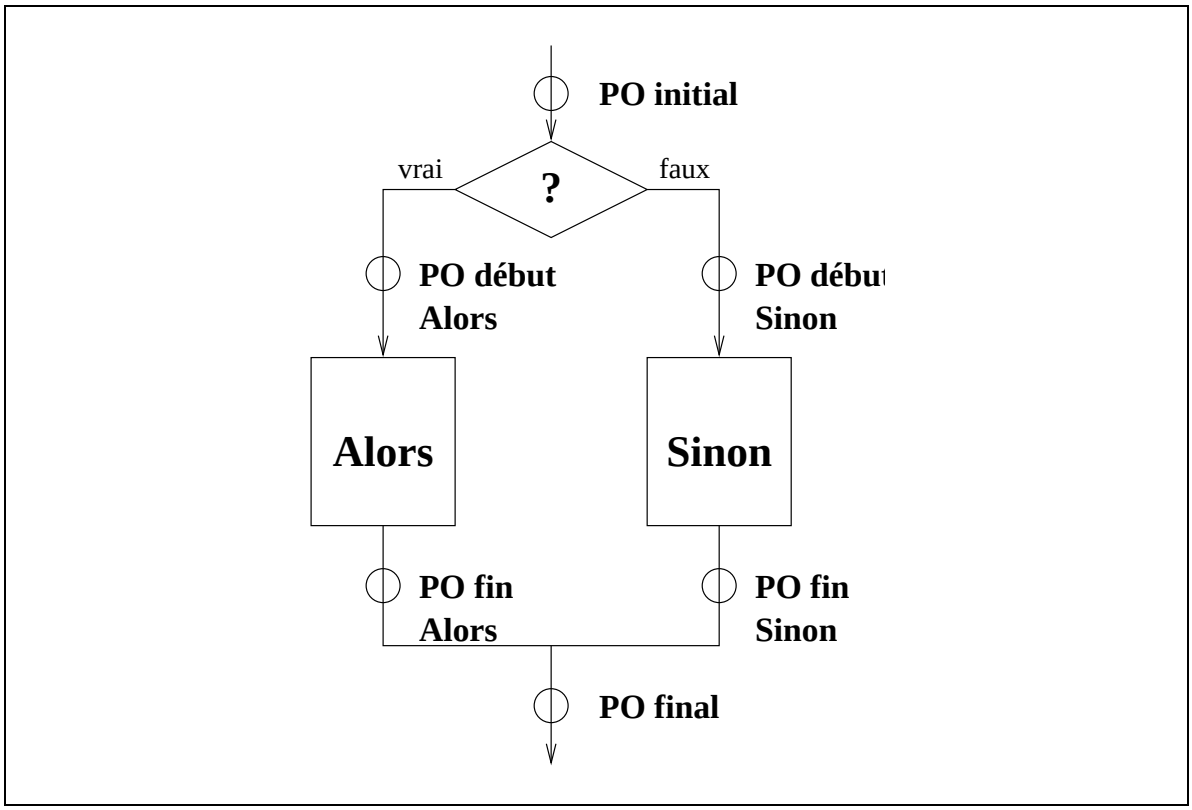


FIG. 4.2 – Points d'observation de l'alternative

```
                        ", disc = " + disc +  ", r = " + r + ", i = " + i) ;
}
System.out.println("fin         : a = " + a + ", b = " + b + ", c = " + c +
                   ", disc = " + disc + ", r = " + r + ", i = " + i) ;
```

L'exécution de cette portion de programme sur les valeurs `a` = 2.0, `b` = 7.0 et `c` = 3.0 donne la trace :

```
init        : a = 2.0, b = 7.0, c = 3.0
début       : a = 2.0, b = 7.0, c = 3.0, disc = 25.0
début-alors : a = 2.0, b = 7.0, c = 3.0, disc = 25.0
fin-alors   : a = 2.0, b = 7.0, c = 3.0, disc = 25.0, r = -0.5, i = 0.0
fin         : a = 2.0, b = 7.0, c = 3.0, disc = 25.0, r = -0.5, i = 0.0
```

et sur les valeurs `a` = 2.0, `b` = 2.0 et `c` = 3.0, elle donne la trace :

```
init        : a = 2.0, b = 2.0, c = 3.0
début       : a = 2.0, b = 2.0, c = 3.0, disc = -20.0
début-sinon : a = 2.0, b = 2.0, c = 3.0, disc = -20.0
fin-sinon   : a = 2.0, b = 2.0, c = 3.0, disc = -20.0, r = -0.5, i = 1.1180339887498949
fin         : a = 2.0, b = 2.0, c = 3.0, disc = -20.0, r = -0.5, i = 1.1180339887498949
```

□

Raisonnement par assertions sur les alternatives

Le raisonnement sur l'énoncé d'alternative utilise des assertions également posées aux six points d'observation de l'énoncé :

$\{pre\}$
`if C {`
    $\{début\text{-}alors\}$
    $\mathtt{E}_1$
    $\{fin\text{-}alors\}$
`} else {`
    $\{début\text{-}sinon\}$
    $\mathtt{E}_2$
    $\{fin\text{-}sinon\}$
$\{post\}$

Par définition de l'énoncé d'alternative, des relations logiques élémentaires existent entre les différentes assertions sur les points d'observation adjacents. Considérons les différents cas :

- Assertions $\{pre\}$ et $\{début\text{-}alors\}$: entre ces deux assertions, la condition a été exécutée et on sait que si on parvient au point d'observation du début de la partie «alors», la condition était vraie. L'assertion $\{début\text{-}alors\}$ est donc l'assertion initiale augmentée de la condition qui est vraie, c'est-à-dire

$$\{pre\} \wedge \mathtt{C} \Rightarrow \{début\text{-}alors\}$$

- Assertions $\{pre\}$ et $\{début\text{-}sinon\}$: de manière similaire au cas précédent, on obtient que

$$\{pre\} \wedge \neg\, \mathtt{C} \Rightarrow \{début\text{-}sinon\}$$

- Assertions $\{fin\text{-}alors\}$, $\{fin\text{-}sinon\}$ et $\{post\}$: lorsque le flôt de contrôle parvient au point d'observation final, on peut avoir exécuté soit la partie «alors», soit la partie «sinon». L'assertion finale doit donc découlé de la disjonction des assertions de fin de partie «alors» et de fin de partie «sinon», c'est-à-dire

$$\{fin\text{-}alors\} \vee \{fin\text{-}sinon\} \Rightarrow \{post\}$$

De même que pour la composition séquentielle, la composition d'une condition `C` et de deux énoncés $\mathtt{E}_1$ et $\mathtt{E}_2$ en un énoncé d'alternative pose des conditions entre les assertions des points d'observation de l'énoncé d'alternative et celles obtenues aux points d'observation initiaux et finaux de $\mathtt{E}_1$ et $\mathtt{E}_2$. En particulier, si $\mathtt{E}_1$ et $\mathtt{E}_2$ sont annotés d'assertions $\{pre_{\mathtt{E}_1}\}$, $\{post_{\mathtt{E}_1}\}$, $\{pre_{\mathtt{E}_2}\}$, et $\{post_{\mathtt{E}_2}\}$, alors les relations suivantes doivent exister entre les assertions sur les points d'observation de l'alternative et ces assertions sur les énoncés constitutifs :

- $\{début\text{-}alors\} \Rightarrow \{pre_{\mathtt{E}_1}\}$.
- $\{post_{\mathtt{E}_1}\} \Rightarrow \{fin\text{-}alors\}$.
- $\{début\text{-}sinon\} \Rightarrow \{pre_{\mathtt{E}_2}\}$.
- $\{post_{\mathtt{E}_2}\} \Rightarrow \{fin\text{-}sinon\}$.

Si on replace toutes ces relations dans leur contexte, on obtient les deux relations :

$$\{pre\} \wedge \mathtt{C} \Rightarrow \{début\text{-}alors\} \Rightarrow \{pre_{\mathtt{E}_1}\}\ \mathtt{E}_1\ \{post_{\mathtt{E}_1}\} \Rightarrow \{fin\text{-}alors\} \Rightarrow \{post\}$$

et :

$$\{pre\} \wedge \neg\mathtt{C} \Rightarrow \{début\text{-}sinon\} \Rightarrow \{pre_{\mathtt{E}_2}\}\ \mathtt{E}_2\ \{post_{\mathtt{E}_2}\} \Rightarrow \{fin\text{-}sinon\} \Rightarrow \{post\}$$

ce qui donne l'ensemble des contraintes posées sur les états aux points d'observation, mais

qui peut aussi se résumer dans la règle de transformation de l'alternative qui s'énonce de la manière suivante :

Si $\{pre \wedge \mathtt{C}\}\ \mathtt{E}_1\ \{post\}$ et $\{pre \wedge \neg\mathtt{C}\}\ \mathtt{E}_2\ \{post\}$
alors $\{pre\}$ `if C { E1 } else { E2 }` $\{post\}$.

Exemple 36 L'énoncé d'alternative annoté suivant modifie une variable `x` pour qu'elle contienne la valeur absolue de sa valeur initiale :

$\{\mathtt{x} = X\}$
```
if (x < 0) {
```
$\{\mathtt{x} = X \wedge \mathtt{x} < 0\}$
```
    x = -x ;
```
$\{\mathtt{x} = -X \wedge \mathtt{x} > 0\}$
```
} else {
```
$\{\mathtt{x} = X \wedge \mathtt{x} \geq 0\}$
```
    ;
```
$\{\mathtt{x} = X \wedge \mathtt{x} \geq 0\}$
```
}
```
$\{(\mathtt{x} = -X \wedge \mathtt{x} > 0) \vee (\mathtt{x} = X \wedge \mathtt{x} \geq 0)\}$

La propriété $\mathtt{x} = abs(X)$, où *abs* dénote la valeur absolue de son argument, découle immédiatement de l'assertion finale. □

Du point de vue du flôt de contrôle et de l'état mémoire, on constate que l'énoncé d'alternative permet de séparer l'ensemble des états possibles à son point d'observation initial en deux sous-ensembles : le sous-ensemble des états pour lesquels la condition de l'énoncé est vraie et le sous-ensemble des états pour lesquels cette condition est fausse. Un traitement approprié peut alors être appliqué à chacun des deux cas pour produire deux ensembles d'états modifiés. Puis, l'exécution ramène le flôt de contrôle au point d'observation final. Cette jointure après deux séquences d'énoncés différentes fait l'union des états possibles à la fin de la partie «alors» avec ceux possibles à la fin de la partie «sinon». D'un point de vue assertionnel, l'union se reflète par le «ou» entre les assertions de fin des deux parties.

L'énoncé d'alternative a donc pour rôle d'introduire une information supplémentaire dans le flôt de contrôle, à savoir la véracité ou la fausseté de la condition de manière à appliquer un traitement approprié à chaque cas. Par contre, la jointure vers l'unique point de sortie de l'énoncé fait perdre de l'information en réunissant des ensembles d'états. L'important si on veut avoir un énoncé d'alternative utile pour le programme consiste à produire des états possédant des propriétés communes, à savoir un résultat utilisable pour la suite du programme. Ceci se manifeste par des parties d'assertions communes entre les deux assertions de fin de partie «alors» et «sinon» de façon à ce que leur réunion par le «ou» logique permette d'extraire ce résultat commun.

Exemple 37 Dans notre exemple précédent, le fait qu'après l'énoncé d'alternative l'assertion $\mathtt{x} = abs(X)$ soit vraie est déjà une conséquence de l'assertion de fin de partie «alors» et de l'assertion de fin de partie «sinon». En effet,

$$\{\mathtt{x} = -X \wedge \mathtt{x} > 0\} \Rightarrow \{\mathtt{x} = abs(X)\}$$

$$\{\mathtt{x} = X \wedge \mathtt{x} \geq 0\} \Rightarrow \{\mathtt{x} = abs(X)\}$$

```
{vrai}
if (x >= y) {
    {x ≥ y}
    if (x >= z) {
       {x ≥ y ∧ x ≥ z}
       m = x ;
       {x ≥ y ∧ x ≥ z ∧ m = x} {m = max(x,y,z)}
    } else {
       {x ≥ y ∧ x < z}
       m = z ;
       {x ≥ y ∧ x < z ∧ m = z} {m = max(x,y,z)}
    }
    {m = max(x,y,z)}
} else {
    {x < y}
    if (y >= z) {
       {x < y ∧ y ≥ z}
       m = y ;
       {x < y ∧ y ≥ z ∧ m = y} {m = max(x,y,z)}
    } else {
       {x < y ∧ y < z}
       m = z ;
       {x < y ∧ y < z ∧ m = z} {m = max(x,y,z)}
    }
    {m = max(x,y,z)}
}
{m = max(x,y,z)}
```

FIG. 4.3 – Calcul du maximum de trois valeurs avec les assertions sur les points d'observation.

donc

$$(\{\mathtt{x} = -X \wedge \mathtt{x} > 0\} \vee \{\mathtt{x} = X \wedge \mathtt{x} \geq 0\}) \;\Rightarrow\; \{\mathtt{x} = abs(X) \vee \mathtt{x} = abs(X)\} \;\Rightarrow\; \{\mathtt{x} = abs(X)\}$$

□

Imbrication des conditionnelles

Les parties «alors» et «sinon» dans un énoncé d'alternative sont elles-mêmes des énoncés. À ce titre, elles peuvent être des énoncés d'alternatives également, comme par exemple :

```
if <condition 1> {
   if <condition 2> {
      <énoncé 1>
   } else {
      <énoncé 2>
   }
} else {
   if <condition 3> {
      <énoncé 3>
   } else {
      <énoncé 4>
   }
}
```

On dit alors que les énoncés d'alternative sont imbriqués, et on peut bien sûr produire autant de niveaux d'imbrication que l'on veut.

Exemple 38 L'imbrication d'alternatives suivante range dans la variable `m` la valeur maximale entre celles des variables `x`, `y` et `z` :

```
if (x >= y) {
   if (x >= z) {
      m = x ;
   } else {
      m = z ;
   }
} else {
   if (y >= z) {
      m = y ;
   } else {
      m = z ;
   }
}
```

Pour se convaincre que cet énoncé produit bien le résultat attendu, la figure 4.3 en montre l'annotation avec les assertions à chaque point d'observation, où max dénote le maximum de ses trois arguments. □

L'imbrication des énoncés d'alternative peut rapidement produire, malgré les contraintes imposés par le flôt de contrôle dirigé par la syntaxe, des programmes relativement opaque. À chaque fois que c'est possible, une bonne pratique pour ramener la complexité du flôt sous un seuil raisonnable consiste à n'imbriquer les énoncés que dans les parties «sinon». La forme générale de ces imbrications contrôlées est :

```
if (<condition 1>) {
   <énoncé 1>
} else if (<condition 2>) {
   <énoncé 2>
...
} else if (<condition n-1>) {
   <énoncé n-1>
} else {
   <énoncé n>
}
```

Le grand avantage de discipliner ainsi l'imbrication des alternatives est qu'au lieu de produire une arborescence de division de l'ensemble des états possibles, on produit plutôt une séquence linéaire de sous-ensembles qui sont traités l'un après l'autre par les parties «alors» de chacun des énoncés d'alternative, et par la partie «sinon> du dernier énoncé. Cette linéarisation du traitement est généralement plus facile à comprendre que l'arborescence produite par une imbrication incontrôlée.

Exemple 39 Considérons le problème de déterminer si une année est bissextile ou non. Plus précisément, écrivons un énoncé qui range dans une variable `estBissextile` la valeur booléenne `vrai` si l'année contenue dans la variable `annee` est bissextile, et `faux` sinon. Par définition, une année qui est un multiple de 4 est bissextile, sauf si elle est multiple de 100 auquel cas elle n'est bissextile que si elle est aussi multiple de 400. Une première possibilité consiste à imbriquer les énoncés d'alternative de la manière suivante :

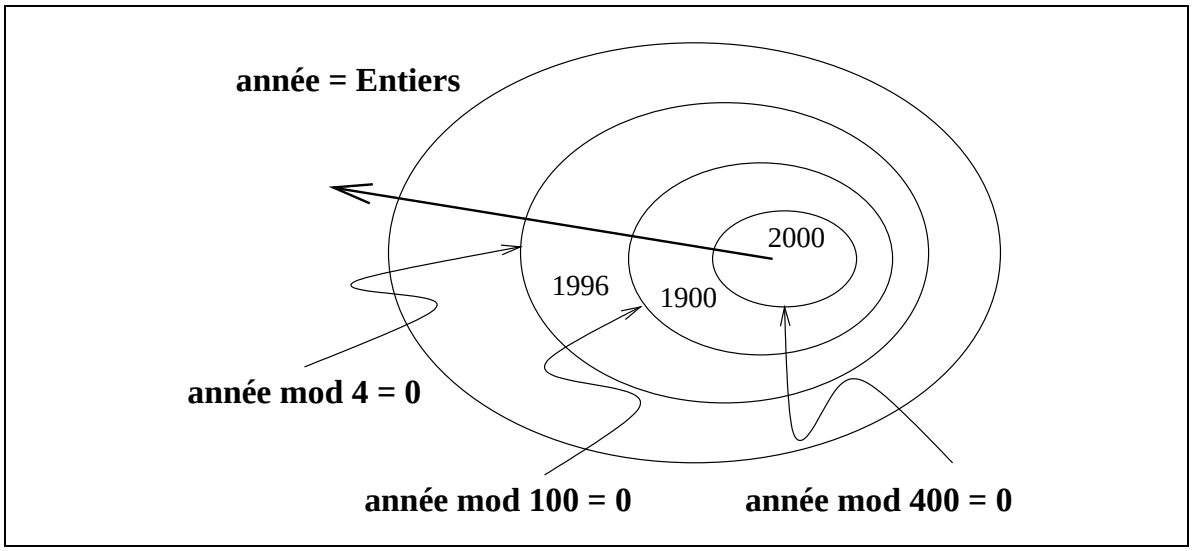


FIG. 4.4 – Découpage des années pour décider si elles sont bissextiles

```
if (annee % 4) == 0 {
    if (annee % 100) == 0 {
       if (annee % 400) == 0 {
          {annee mod 4 = 0 ∧ annee mod 100 = 0 ∧ annee mod 400 = 0}
          estBissextile = vrai ;
       } else {
          {annee mod 4 = 0 ∧ annee mod 100 = 0 ∧ annee mod 400 ≠ 0}
          estBissextile = faux ;
       }
    } else {
       {annee mod 4 = 0 ∧ annee mod 100 ≠ 0}
       estBissextile = vrai ;
    }
} else {
    {annee mod 4 ≠ 0}
    estBissextile = faux ;
}
```

Cette imbrication est relativement difficile à suivre. Les assertions peuvent aider à se rassurer sur le bien-fondé de cet énoncé, mais entre deux programmes corrects, on doit la plupart du temps privilégier le programme le plus simple et le plus clair. La difficulté ici vient de ce que les conditions ne sont pas mutuellement exclusives. Peut-on linéariser les différents cas et utiliser une imbrication uniquement dans les parties «sinon» ? Considérons les ensembles de valeurs d'année en fonction des différents cas. La figure 4.4 montre que ces différents ensembles sont inclus les uns dans les autres dans l'ordre : $\{\texttt{annee} \mid \texttt{annee}\, mod\, 400 = 0\} \subset \{\texttt{annee} \mid \texttt{annee}\, mod\, 100 = 0\} \subset \{\texttt{annee} \mid \texttt{annee}\, mod\, 4 = 0\} \subset \{\texttt{annee} \mid \texttt{annee} \in I\!N\}$.

En terme de traitement des différents cas, on veut obtenir le résultat suivant en fonction du sous-ensemble des entiers dans lequel se trouve la valeur de la variable `annee` :

- $\{\texttt{annee} \mid \texttt{annee}\, mod\, 400 = 0\}$: l'année est bissextile.
- $\{\texttt{annee} \mid \texttt{annee}\, mod\, 400 \neq 0 \wedge \texttt{annee}\, mod\, 100 = 0\}$: l'année n'est pas bissextile.
- $\{\texttt{annee} \mid \texttt{annee}\, mod\, 400 \neq 0 \wedge \texttt{annee}\, mod\, 100 \neq 0 \wedge \texttt{annee}\, mod\, 4 = 0\}$: l'année est bissextile.
- $\{\texttt{annee} \mid \texttt{annee}\, mod\, 400 \neq 0 \wedge \texttt{annee}\, mod\, 100 \neq 0 \wedge \texttt{annee}\, mod\, 4 \neq 0\}$: l'année n'est pas bissextile.

De cette catégorisation, on tire un premier énoncé d'alternative imbriqué par les parties «sinon» :

```
if ((année % 400) == 0) {
   estBissextile = true ;
} else if ((année % 400) != 0 && (année % 100) == 0) {
   estBissextile = false ;
} else if ((année % 400) != 0 && (année % 100) != 0 && (année % 4) = 0) {
   estBissextile = true ;
} else {
   estBissextile = false ;
}
```

Mais il est possible de simplifier les conditions successives en notant, comme cela devient évident par les assertions, que ce qui a été vérifié auparavant n'a pas besoin d'être revérifié à nouveau :

```
if ((annee % 400) == 0) {
   {annee mod 400 = 0}
   estBissextile = true ;
} else if ((annee % 100) == 0) {
   {annee mod 400 ≠ 0 ∧ annee mod 100 = 0}
   estBissextile = false ;
} else if ((année % 4) == 0) {
   {annee mod 400 ≠ 0 ∧ annee mod 100 ≠ 0 ∧ annee mod 4 = 0}
   estBissextile = true ;
} else {
   {annee mod 400 ≠ 0 ∧ annee mod 100 ≠ 0 ∧ annee mod 4 ≠ 0}
   estBissextile = false ;
}
```

□

Le genre de simplification que nous avons fait sur l'exemple des années bissextiles utilise l'ordre d'exécution de l'énoncé d'alternative pour ne pas réévaluer plusieurs fois les mêmes conditions, ce qui peut avoir un effet positif su la performance du programme si la condition demande beaucoup de calcul. Mais alors le programme devient dépendant de l'ordre dans lequel les conditions sont placées dans l'imbrication. Il est alors judicieux de l'indiquer par un commentaire.

En résumé, le savoir-faire à développer pour écrire des énoncés d'alternative corrects se résume par les éléments suivants :

1. Déterminer l'objectif : l'information nécessaire pour la suite (état final).
2. Déterminer les différentes alternatives et les conditions associées.
3. Exprimer les conditions sous forme d'expressions logiques (prédicatives).
4. Identifier les cas de recouvrement dans les conditions.
5. Ordonner les cas de recouvrement de façon correcte (évite de calculer plusieurs fois la même chose).
6. Exprimer explicitement les conditions «complémentaires» pour vérifier que les énoncés de la partie sinon leur correspondent bien.
7. Écrire le code permettant dans chaque cas d'atteindre l'objectif de départ.

4.4 Énoncés de répétition

Les énoncés de répétition, ou d'itération, permettent d'exécuter plusieurs fois un énoncé ou une séquence d'énoncés. Ils existent en deux grandes familles :

1. Énoncés de répétition comptée : ces énoncés exécutent n fois un énoncé (éventuellement composé), pour un certain n déterminé au moment où l'itération débute.

2. Énoncés de répétition conditionnelle : ces énoncés exécutent un certain nombre de fois un énoncé (éventuellement composé), tant qu'une condition est vraie.

Parce que les énoncés de répétition conditionnelle sont les plus généraux, nous allons examiner ici que ces derniers. Les autres énoncés de répétition de Java seront présentés à la section 4.6. La forme syntaxique de la répétition conditionnelle qui exécute un énoncé tant qu'une condition est vraie est la suivante :

```
while (<condition>) {
    <énoncé>
}
```

La partie `<condition>` est une expression booléenne, alors que la partie `<énoncé>` est un énoncé quelconque, éventuellement composé. L'exécution de cet énoncé de répétition conditionnelle va d'abord évaluer la partie `<condition>`. Si cette expression s'évalue à vrai, alors on exécute la partie `<énoncé>`. Ensuite, on réévalue la partie `<condition>`, et si elle s'évalue à vrai, le corps de l'itération est exécuté à nouveau. Et ainsi de suite tant que la partie `<condition>` s'évalue à vrai. Dès que la partie `<condition>` s'évalue à faux, l'énoncé de répétition se termine et on poursuit l'exécution avec l'énoncé suivant dans le programme.

Exemple 40 Considérons l'exemple de l'accumulation dans la variable `s` de la somme des entiers de 1 à n. On définit la répétition conditionnelle suivante pour réaliser cette somme :

```
s = 0 ;
i = 1 ;
while (i <= n) {
   s = s + i ;
   i = i + 1 ;
}
```

On commence par initialiser les variables `s` et `i` à 0 et 1 respectivement, puis la répétition va s'exécuter tant que la valeur de `i` est inférieure ou égale à `n`. Chaque itération augmente la valeur de ls somme `s` de la valeur courante de `i`, puis incrémente `i` de 1 pour passer à l'entier suivant. □

Raisonnement sur les répétitions

Les points d'observation sur l'énoncé '`while`' sont au nombre de cinq :

1. le point d'observation initial,
2. le point d'observation d'avant la condition,
3. le point d'observation du début du corps de la répétition,
4. le point d'observation de la fin du corps de la répétition, et
5. le point d'observation final.

Ces points d'observations sont illustrés, de même que les fils d'exécution qui peuvent être produits par l'énoncé de répétition, à la figure 4.5. Chacun de ces points d'observation peut faire l'objet d'instrumentations par des énoncés d'impression pour comprendre l'exécution de cet énoncé.

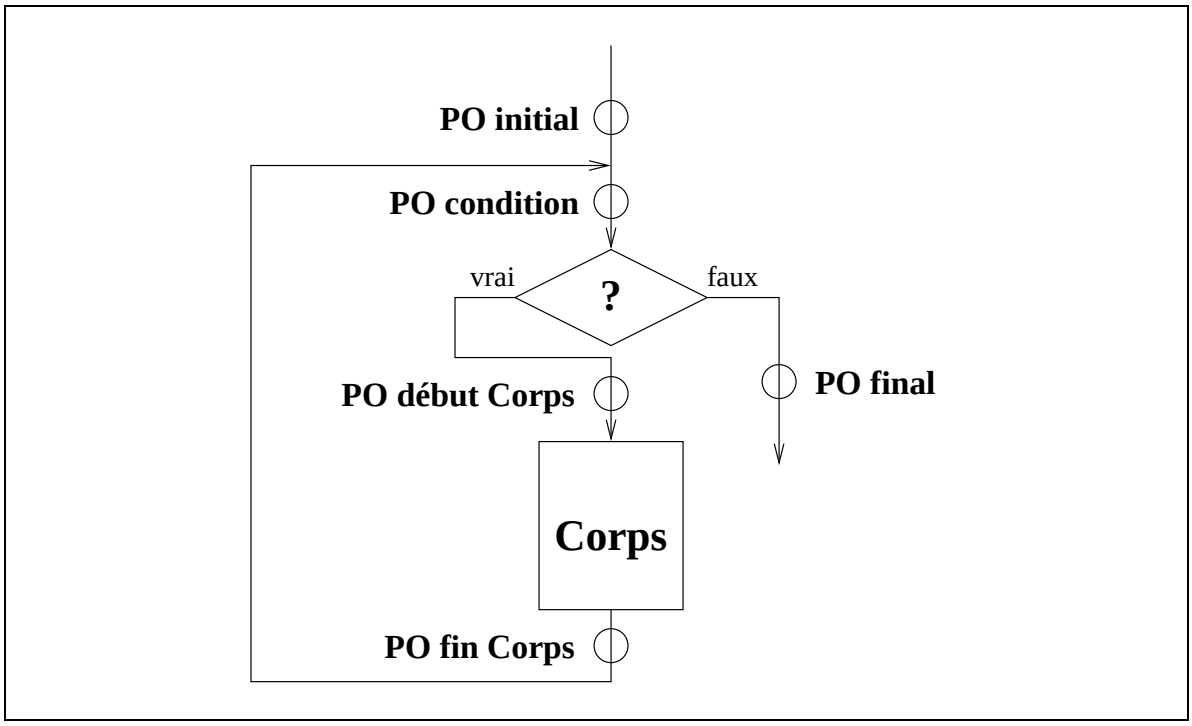


FIG. 4.5 – Points d'observation de l'itération

	1	2	3	4	5	3	4	5	3	4	5	3	4	5	3	4	5	3	6
i	*ni*	1	1	1	2	2	2	3	3	3	4	4	4	5	5	5	6	6	6
s	0	0	0	1	1	1	3	3	3	6	6	6	10	10	10	15	15	15	15

FIG. 4.6 – Trace de la répétition calculant la somme des entiers de 1 à n.

Exemple 41 Considérons à nouveau l'exemple de la répétition calculant la somme des entiers 1 à n et numérotons les énoncés :

```
1.   s = 0 ;
2.   i = 1 ;
3.   while (i <= n) {
4.      s = s + i ;
5.      i = i + 1 ;
6.   }
```

La figure 4.6 présente la trace de cette séquence d'énoncés en posant `n` = 5. L'axe horizontal développe le fil d'exécution obtenu, alors que l'axe vertical donne les valeurs prises par les variables `i` et `s` après l'exécution de l'énoncé dont le numéro apparaît en haut de la colonne. □

Chacun de ces points d'observation peut aussi faire l'objet d'annotations par des assertions pour raisonner sur l'effet de cet énoncé. L'annotation d'un énoncé tant que produit donc les assertions suivantes :

{*pre*}
`while` {*avant-condition*} `C {`
    {*début-corps*}
    `E`
    {*fin-corps*}
`}`
{*post*}

Comme dans le cas de l'énoncé d'alternative, les assertions aux différents points d'observation doivent aussi respecter certaines relations :

- $\{pre\} \Rightarrow \{avant\text{-}condition\}$
- $\{avant\text{-}condition\} \wedge \neg \mathtt{C} \Rightarrow \{post\}$
- $\{avant\text{-}condition\} \wedge \mathtt{C} \Rightarrow \{début\text{-}corps\}$
- $\{fin\text{-}corps\} \Rightarrow \{avant\text{-}condition\}$

De même, si l'énoncé `E` est annoté d'assertions $\mathtt{E}_{pre}$ et $\mathtt{E}_{post}$, alors la composition d'une condition `C` et d'un énoncé `E` nécessite que les assertions de `E` et les assertions des points d'observation de la répétition respecte les conditions suivantes :

- $\{début\text{-}corps\} \Rightarrow \mathtt{E}_{pre}$, et
- $\mathtt{E}_{post} \Rightarrow \{fin\text{-}corps\}$.

Par transitivité de l'implication, on en déduit également que :

- $\{pre\} \wedge \neg \mathtt{C} \Rightarrow \{post\}$
- $\{fin\text{-}corps\} \wedge \neg \mathtt{C} \Rightarrow \{post\}$
- $\{pre\} \wedge \mathtt{C} \Rightarrow \{début\text{-}corps\}$
- $\{fin\text{-}corps\} \wedge \mathtt{C} \Rightarrow \{début\text{-}corps\}$

Les deux premières relations nous indiquent que l'ensemble des états possibles à la fin de la répétition est obtenue de l'union des états possibles à la fin du corps de l'énoncé de répétition et de ceux possibles à l'entrée de l'énoncé, si ces états sont tels que la condition de l'énoncé est fausse. L'information (assertion) qui sera avérée à la fin de l'énoncé sera commune aux assertions $\{pre\}$, $\{fin\text{-}corps\}$ et $\neg \mathtt{C}$. Les deux dernières relations nous indiquent que l'assertion de début de corps de la répétition, qui est vraie à chaque passage dans le corps de la répétition, est une conséquence logique des assertions $\{pre\}$, $\{fin\text{-}corps\}$ et `C`.

L'examen de ces relations nous indique qu'il y a une information commune à toutes ces assertions qui est vraie en tout point d'observation de l'énoncé de répétition. Cette information commune est appelée *invariant* P de la répétition. Cet invariant joue un rôle essentiel dans la compréhension de l'exécution de l'itération et nous permet de résumer de la manière suivante la règle de transformation de la répétition '`while`' :

$$\text{Si } \{P \wedge \mathtt{C}\}\ \mathtt{E}\ \{P\}$$
$$\text{alors } \{P\}\ \texttt{while (C) \{ E \}}\ \{P \wedge \neg \mathtt{C}\}.$$

Exemple 42 Considérons à nouveau la séquence d'énoncés qui calcule la somme des entiers de 1 à n. L'étude attentive de la trace de cette séquence d'énoncés qui apparaît à la figure 4.6 montre qu'effectivement une relation entre les valeurs de `i` et `s` est vérifiée après l'exécution de l'énoncé 2, après chaque exécution des énoncés 3 et 5, et enfin après la répétition. En fait, à chaque fois, `s` contient la somme des entiers de 0 à `i - 1`. Cette relation est l'invariant de la répétition.

Nous avons vu plus tôt dans ce chapitre que la séquence d'énoncés `i = i + 1 ; s = s + i ;` observe les assertions suivantes :

$\{A_1 : \mathtt{i} \geq 0 \wedge \mathtt{s} = 0 + 1 + 2 + \ldots + (\mathtt{i} - 1)\}$

```
s = s + i ;
i = i + 1 ;
```

$\{A_2 : \mathtt{i} > 0 \wedge \mathtt{s} = 0 + 1 + 2 + \ldots + (\mathtt{i} - 1)\}$

Considérons l'assertion suivante comme invariant de la répétition : $\{P : \mathtt{n} \geq 0 \wedge 0 \leq \mathtt{i} \leq (\mathtt{n} + 1) \wedge \mathtt{s} = 0 + 1 + 2 + \ldots + (\mathtt{i} - 1)\}$. On peut alors annoter la séquence précédente en s'aidant de cet invariant :

```
s = 0 ;
i = 1 ;
```
$\{pre : \mathtt{i} = 1 \wedge P\}$
```
while {ac : P} (i <= n) {
```
$\quad \{dc : \mathtt{i} \leq \mathtt{n} \wedge P\} \Rightarrow \{\mathtt{n} \geq 0 \wedge \mathtt{i} \leq \mathtt{n} \wedge A_1\}$
```
    s = s + i ;
    i = i + 1 ;
```
$\quad \{\mathtt{n} \geq 0 \wedge \mathtt{i} \leq (\mathtt{n} + 1) \wedge A_2\}$
$\quad \{fc : \mathtt{n} \geq 0 \wedge 0 < \mathtt{i} \leq (\mathtt{n} + 1) \wedge \mathtt{s} = 0 + 1 + 2 + \ldots + (\mathtt{i} - 1)\} \Rightarrow \{P\}$
```
}
```
$\{post : \mathtt{i} > \mathtt{n} \wedge P\} \Rightarrow \{\mathtt{n} \geq 0 \wedge \mathtt{i} = \mathtt{n} + 1 \wedge \mathtt{s} = 0 + 1 + 2 + \ldots + \mathtt{i} - 1\}$
$\{R : \mathtt{s} = 0 + 1 + 2 + \ldots + \mathtt{n}\}$

Cette séquence est vraie si $\mathtt{n} \geq 0$, condition préservée tout au long de la séquence et que nous avons par conséquent omis pour ne pas surcharger les assertions. P est trivialement vraie au point d'observation initial puisque `i` contient la valeur 1 et `s` contient 0, alors que la somme de 0 à 0 peut être considérée comme étant égale à 0. P est vraie également au point d'observation avant la condition car elle découle à la fois de l'assertion *pre* et de l'assertion *fc*. P est également vraie au début du corps de la répétition puisque rien n'a été fait par le programme qui pourrait en modifier la véracité si elle était vraie avant la condition. Enfin, pour la même raison P est aussi vraie au point d'observation final, puisqu'elle était vraie avant la condition et que si la condition est fausse, on passe au point d'observation final sans avoir fait quoique ce soit qui puisse changer sa véracité.

Le seul point qu'il faut encore vérifier est que si P est vraie au début du corps, alors elle est aussi vraie à la fin du corps. Pour cela, on vérifie que l'assertion *dc* implique bien l'assertion A_1, ce qui est le cas trivialement. Par le raisonnement appliqué à la séquence `i = i + 1 ; s = s + i` plus tôt, on en déduit que l'assertion A_2 est aussi vraie après la séquence. Il reste donc à vérifier que P découle de A_2, ce qui est le cas puisque *fc* découle de A_2 et du fait que si $\mathtt{i} < \mathtt{n}$ au début du corps (*dc*), l'incrémentation de 1 produit au pire une valeur égale à `n`, c'est-à-dire $\mathtt{i} \leq \mathtt{n}$. P découle bien de *fc* car si $0 < \mathtt{i} \leq \mathtt{n}$ alors nécessairement $0 \leq \mathtt{i} \leq \mathtt{n}$.

L'invariant étant démontré, on peut s'en servir pour montrer que la répétition fait bien le travail attendu en raisonnant à l'aide de la règle de transformation précédente. L'assertion R exprime ce qui est attendu de la répétition : `s` contient la somme de 0 à `n`. L'assertion $\{post = \neg(\mathtt{i} \leq \mathtt{n}) \wedge P\}$ nous donne tout ce qu'il faut puisque $\{post : \mathtt{i} > \mathtt{n} \wedge 0 \leq \mathtt{i} \leq \mathtt{n} + 1 \wedge \mathtt{n} \geq 0 \wedge \mathtt{s} = 0 + 1 + 2 + \ldots + \mathtt{i} - 1\} \Rightarrow \{\mathtt{i} = \mathtt{n} + 1 \wedge \mathtt{n} \geq 0 \wedge \mathtt{s} = 0 + 1 + 2 + \ldots + \mathtt{i} - 1\}$, ce qui implique immédiatement R. □

L'utilisation d'un énoncé de répétition dans un programme est toujours un moyen de calculer un résultat par approximations successives. Dans notre exemple de calcul de la somme des entiers de 0 à n, la somme est approximée par $\sum_{i=0}^{k} i$ pour un certain $0 \leq k \leq n$, comme cela est illustré à la figure 4.7. À chaque exécution du corps de la répétition, cette approximation

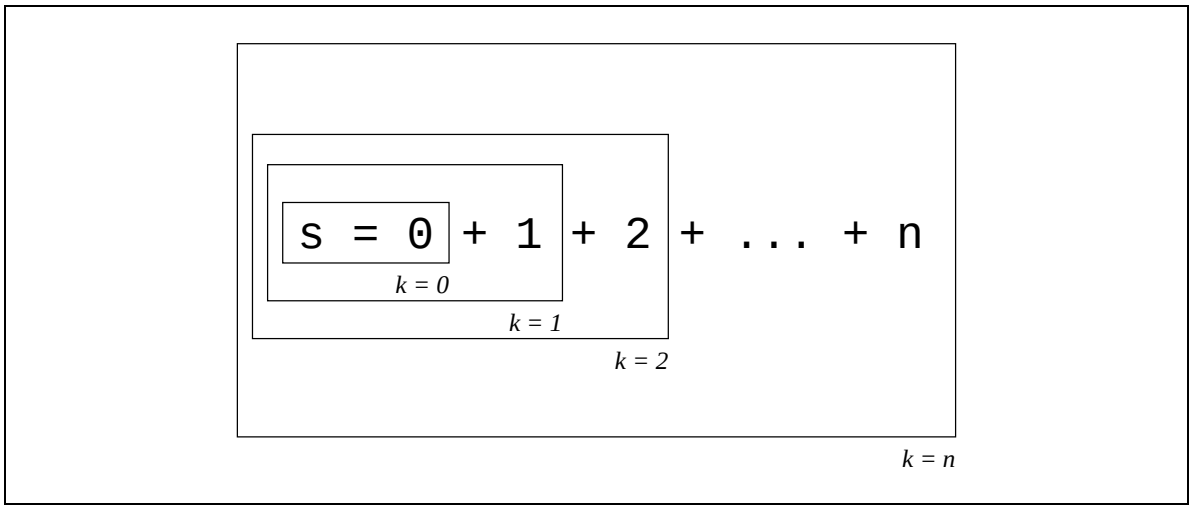


FIG. 4.7 – Itération comme un calcul par approximations successives.

s'améliore dans la mesure où la valeur de k se rapproche de n. La répétition se termine lorsque l'approximation n'est plus une approximation mais devient égale au résultat à calculer.[4]

En résumé, les étapes à suivre pour écrire des itérations correctes peuvent se résumer par :

1. Déterminez l'objectif de la répétition, à savoir la condition que devra respecter l'état mémoire après la répétition pour que son travail soit réalisé.
2. Exprimez le calcul à faire sous forme d'approximations successives en explicitant les étapes du calcul.
3. Déterminez comment passer d'une étape du calcul à la suivante, et tirez-en le corps de l'énoncé de répétition.
4. Déterminez la condition à remplir pour que l'approximation obtenue soit considérée comme le résultat de l'itération. Cette condition va devenir la condition de fin de la répétition. À partir de la condition de fin de la répétition, exprimez la condition de poursuite si l'énoncé utilisé est de type '`while`'.
5. Déterminez les conditions initiales à respecter au début de la répétition et tirez-en les énoncés d'initialisations nécessaires qui viendront juste avant l'énoncé de répétition. Vérifiez que le résultat sera correct même dans les cas limites où on ne passe pas du tout par le corps de la répétition.
6. Vérifiez que les énoncés du corps de la répétition permettent bien de progresser vers l'objectif final à chaque passage dans la répétition, i.e. que l'on modifie quelque chose utilisé dans la condition.

Exemple 43 Si on reprend l'exemple précédent, l'objectif de la répétition est d'accumuler dans une variable `s` la somme des entiers de 0 à n, pour $n \geq 0$, où on suppose que la valeur n est contenue dans une variable `n`. Les approximations successives seront construires en ajoutant un par un les entiers de 0 à n à la somme ; une étape de calcul consiste donc à avoir dans la variable `s` la somme des entiers de 0 à $i-1$, la valeur i étant contenue dans une variable `i`. Passer à l'étape suivante va consister à passer à l'entier suivant en ajoutant la valeur de

[4]Ou tenu pour tel, puisque dans certains cas, on se contente d'un résultat approché. Pensez par exemple au calcul d'une racine carrée. Le résultat précis étant bien souvent impossible à calculer sur ordinateur, on se contente d'un résultat qui s'approche du plus près possible de la solution.

la variable `i` à la somme cumulée dans `s` et puis en incrémentant de 1 la variable `i`. Dans ce contexte, la condition à remplir pour que la variable `s` contienne la somme des entiers de 1 à n est que la valeur de la variable `i` atteigne $n+1$; la condition de poursuite de la répétition est donc que la valeur de `i` soit inférieure ou égale à la valeur de la variable `n`. Les conditions initiales à respecter, outre l'hypothèse sur `n`, sont que `i` vaille 1 et `s` 0. Le corps de la répétition permet de progresser vers la fin de la répétition puisque qu'à chaque exécution du corps de la répétition, la valeur de la variable `i` est incrémentée de 1. Or, la valeur de `n` reste inchangée dans le corps de la répétition et `i` vaut initialement 1. La valeur de `i` doit donc forcément dépasser celle de `n` et donc forcer la sortie de la répétition. □

Le guide précédent vise à éviter les erreurs les plus courantes dans l'écriture des répétitions, c'est-à-dire une condition de fin de répétition incorrecte qui fait qu'on exécute une fois de trop le corps de la répétition (ou une fois en moins que nécessaire), traitement incorrect des cas limites[5], ou encore non terminaison parce que la condition de sortie de la répétition n'est jamais vérifiée (la répétition s'exécute un nombre «infini» de fois).

4.5 L'appel de méthode ou envoi de message

L'appel de méthode, appelé aussi envoi de message en programmation par objets, est aussi un énoncé de rupture de séquence inconditionnel dans la mesure où son exécution suppose d'interrompre le cours normal de l'exécution séquentielle pour aller exécuter les énoncés du corps de la méthode appelée avant de revenir au point de l'appel pour poursuivre le programme. La syntaxe de l'appel de méthode est :

```
<receveur>.<nom>(<expression>, ..., <expression>)
```

où la partie `<receveur>` est une expression dont le résultat doit être un identificateur d'objet qui va désigner le receveur du message, la partie `<nom>` est un identificateur correspondant au nom de la méthode qui va devoir être exécutée, et les parties `<expression>` sont des expressions quelconques qui vont rendre comme résultats les paramètres réels à passer à la méthode exécutée.

Deux points sont cruciaux dans la compréhension de l'appel de méthode :
- Comment se fait le lien entre l'appel de méthode et la méthode effectivement exécutée ?
- Comment se passent les informations entre le contexte de l'appel et le contexte de la méthode appelée, que ce soit lors du passage des paramètres à la méthode ou lors du retour du résultat ?

Le lien entre l'appel de méthode et la méthode exécutée a été abordé au chapitre précédent (§3.1). Deux éléments entrent en ligne de compte dans ce lien : l'objet receveur ainsi que le nombre et les types des paramètres utilisés dans l'appel. Une classe peut définir plusieurs méthodes de même nom si chacune d'entre elles a une liste de paramètres formels différente. Ce phénomène a été illustré en particulier par les méthodes d'initialisation des classes. Lors d'un envoi de message, le système détermine de quelle classe est l'objet receveur, et recherche dans cette classe la méthode qui correspond à l'appel, c'est-à-dire celle qui possède le même nom ainsi que le même nombre et les mêmes types de paramètres. S'il ne trouve pas dans la classe d'instantiation de l'objet, il poursuit sa recherche avec la superclasse. La vérification de type faite par le compilateur Java assure qu'une méthode sera toujours trouvée pour répondre au message.

[5] Dans notre exemple de calcul de la somme, est-ce que le résultat est correct si $n = 0$?

Une fois réalisée cette sélection de la méthode à exécuter, il faut passer les paramètres réels à la méthode, puis exécuter le corps de cette méthode, et enfin récupérer le résultat éventuel. Pour bien comprendre le passage des paramètres, il faut voir les paramètres formels de la méthode comme des variables locales à la méthode dont les valeurs vont être fournies par les expressions évaluées lors de l'appel. Lors de la définition d'une méthode, les paramètres formels permettent donc de désigner des réceptacles (qui seront des emplacements mémoire) servant à recevoir les valeurs des paramètres réels lors de l'exécution de la méthode.

L'évaluation des expressions fournissant les paramètres réels se fait dans le contexte de l'appel de la méthode. Une fois obtenues, ces valeurs des paramètres réels vont être copiées dans le contexte de la méthode appelée, c'est-à-dire dans les réceptacles correspondants aux paramètres formels de la méthode.

Exemple 44 Considérons la classe `Poly2` du chapitre précédent, et définissons une méthode `additionneNFois` qui prend deux paramètres : un poynôme et le nombre de fois que ce polynôme doit être additionné au receveur :

```
public Poly2       additionneNFois(
    Poly2 autrePoly,     // polynôme à additionner
    int n                // nombre d'additions à faire
    )
{
   if (n == 0) {
      return this ;
   } else {
      return this.additionne(autrePoly).additionneNFois(autrePoly, n - 1) ;
   }
}
```

Considérons maintenant l'appel de cette méthode dans le contexte suivant :

```
Poly2 p1 = new Poly2(1.0, 1.0, 1.0) ;
int i = 9 ;

p1 = (new Poly2()).additionneNFois(p1.inverseAdditif(), i + 1) ;
```

Lors de l'exécution, l'expression `(new Poly2())` va être évaluée. Cela va entraîner la création d'un nouvel objet polynôme dont tous les coefficients seront égaux à 0.0. Le résultat de l'expression sera l'identificateur de l'objet créé. À cet objet, on envoie le message `additionneNFois(p1.inverseAdditif(), i + 1)`. La sélection de la méthode à exécuter va se porter sur la méthode définie ci-haut. Il faut ensuite évaluer les paramètres formels. Il y a deux expression à évaluer pour cela : `p1.inverseAdditif()` et `i + 1`.

L'expression `p1.inverseAdditif()` aura pour résultat un polynôme représentant l'inverse additif du polynôme dont l'identificateur d'objet est dans la variable `p1`. Comme cet objet polynôme représente $x^2 + x + 1$, le résultat sera un objet polynôme représentant $-x^2 - x - 1$ dont l'identificateur d'objet est, disons, `o1`. L'expression `i + 1` s'évalue à 10 puisque la variable `i` contient la valeur 9.

Avant d'exécuter le corps de la méthode `additionneNFois`, les valeurs des deux paramètres, soient l'identificateur d'objet `o1` et la valeur 10 sont copiées dans les paramètres formels `autrePoly` et `n`. Le corps est ensuite exécuté, et l'objet polynôme qui résultera du calcul représentera le polynôme $-10x^2 - 10x - 10$. L'identificateur d'objet de ce polynôme, disons `o2`, sera retourné comme résultat de la méthode.

Au retour, on revient au point où l'exécution était rendue dans le contexte de l'appel. Dans notre exemple, le programme se poursuit par l'affectation du résultat de l'appel de la méthode `additionneNFois` à la variable `p1`. Le résultat étant l'identificateur d'objet `o2`, celui-ci est rangé dans l'emplacement mémoire correspondant à la variable `p1`. □

Passage par valeur versus passage par référence

La discussion précédente simplifie quelque peu la question du passage des paramètres à une méthode. La description qui vient d'être faite laisse entendre que les paramètres sont passés par «copie» des valeurs calculées dans le contexte appelant vers les réceptacles fournis par les paramètres formels dans le contexte appelé (la méthode). Il s'agit bel et bien de ce que l'on appelle un *mode de passage des paramètres* qui, s'il est le plus courant, n'est pas le seul.

En effet, la nature de la variable impérative, comme on l'a vu, fait que l'on peut en modifier le contenu par affectation. La prise en compte des affectations dans le contexte du passage des paramètres pose quelques problèmes. Considérons-le sous un angle pratique. La séquence suivante réalise l'échange des valeurs de deux variables `x` et `y` :

```
t = x ;
x = y ;
y = t
```

Dans cet échange, la variable `t` joue un rôle de réceptacle temporaire pour la valeur initiale de la variable `x` de façon à ne pas la perdre lors de l'affectation à `x` et la ranger dans `y`. Considérons la possibilité de réaliser l'échange des valeurs de deux variables entières par une méthode `echange` d'une hypothétique classe `Manip` et qui aurait la forme suivante :

```
public static void      echange(int i, int j) {
   int t ;

   t = i ; i = j ; j = t ;
}
```

Que se passe-t-il lors de l'appel à cette méthode ? Supposons la portion de programme suivante :

```
int x, y ;

x = 10 ; y = 20 ;
Manip.echange(x, y) ;
System.out.println("x = " + x + " et y = " + y) ;
```

Qu'est-ce qui sera imprimé ? Un essai en Java montrera que le résultat sera l'impression de `"x = 10 et y = 20"`. L'explication en est la suivante. Lors de l'appel à `echange`, les deux paramètres réels, les expressions '`x`' et '`y`' sont évaluées et retournent respectivement 10 et 20. Ces deux valeurs sont copiées dans les paramètres formels `i` et `j` de la méthode `echange`. L'échange des valeurs des paramètres formels `i` et `j` est bel et bien réalisée, mais ce sont les valeurs des paramètres formels qui sont échangées, et non celle des deux variables `x` et `y` puisque ces deux dernières sont inconnues dans le contexte de la méthode `echange`.

Plusieurs langages de programmation impérative proposent un mécanisme de passage de paramètres dit *par référence*. L'idée de ce mode de passage, qui n'est valable que si le paramètre réel est un nom de variable, est de copier dans le contexte appelé non pas le contenu de la variable (sa valeur), mais plutôt l'adresse de l'emplacement mémoire qui contient la valeur de cette variable. La paramètre formel dans la méthode devient alors une espèce d'alias pour la variable du contexte appelant puisqu'il désigne alors le même emplacement mémoire. Dans notre exemple d'échange des valeurs de deux variables, si les deux paramètres de la méthode `echange` sont passés par référence, alors on comprend que les affectations à `i` et `j` dans le corps de la méthode modifient directement le contenu des emplacements mémoire des variables utilisées comme paramètres réels.

Comment ces notions sont-elles prises en compte en programmation par objets ? En fait, la plupart des langages à objets offrent uniquement le mode de passage des paramètres par valeur. C'est le cas de Java, par exemple. Pourtant une forme de passage par référence est réalisée par l'utilisation des identificateurs d'objets. Un identificateur d'objet désigne en effet un objet auquel ont été alloués un certain nombre d'espaces mémoire. Lorsqu'un objet est passé en paramètre à une méthode, on a vu que c'est son identificateur qui est copié dans le paramètre réel de la méthode. Si un message impliquant un modificateur (méthode) lui est envoyé ou si un accès à une variable publique est fait pour lui affecter une nouvelle valeur, c'est le contenu de l'objet qui sera modifié. Au retour de la méthode, l'objet aura été modifié de manière irrémédiable et visible de tous. Le passage d'identificateurs d'objets à des méthodes réalise donc l'équivalent d'un passage par référence.

Exemple 45 Considérons deux classes appelées `Paire` et `Manip` dont le but est purement illustratif :

```
public Class    Paire
{
    public int x ;
    public int y ;

    public static     Paire(int initx, int inity) {
        x = initx ; y = inity ;
    } // ------------------------------------------ Paire()

} // ---------------------------------------------- classe Paire

public class    Manip
{
    public static void    echange(Paire p)
    {
        int t ;

        t = p.x ; p.x = p.y ; p.y = t ;
    } // ------------------------------------------ echange()

    public static void    main(String[] args)
    {
        Paire pp = new Paire(10, 20) ;

        echange(pp) ;
        System.out.println("x = " + pp.x + " et y = " + pp.y) ;
    } // ------------------------------------------ main()

} // ---------------------------------------------- classe Manip
```

L'exécution de la méthode `main` de la classe racine `Manip` va provoquer la création d'un objet `Paire` dont les valeurs des variables `x` et `y` sont 10 et 20 et dont l'identificateur d'objet, disons `o1`, est rangé dans la variable `pp`. Lors de l'appel de la méthode `echange`, l'identificateur

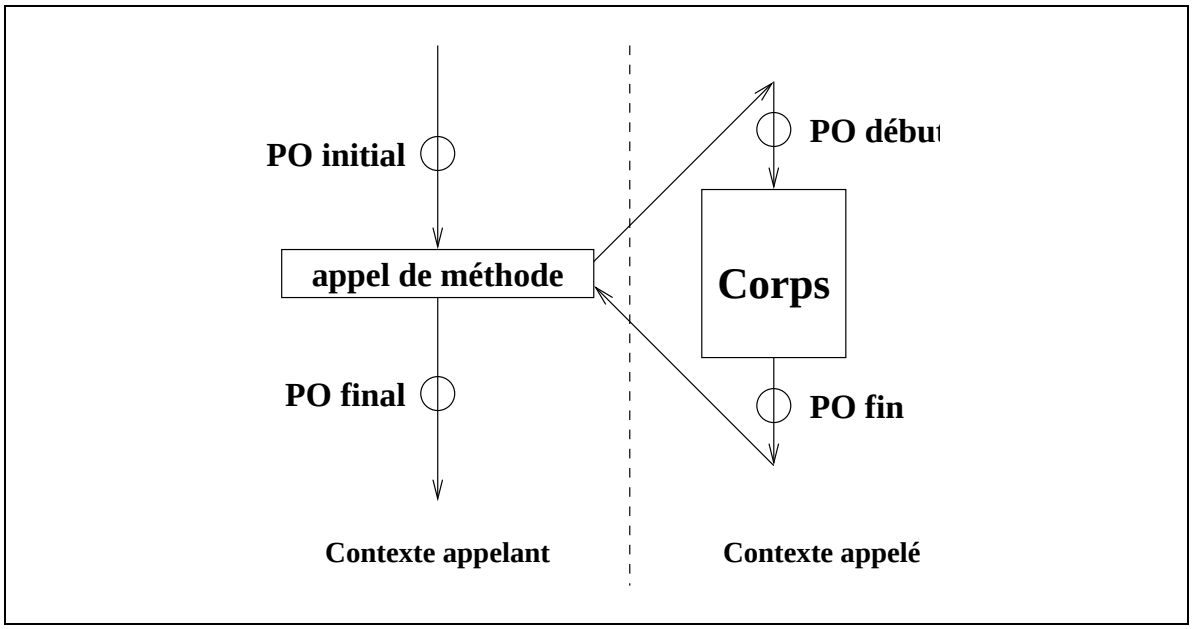


FIG. 4.8 – Points d'observation de la méthode

d'objet `o1` va être copié dans le paramètre formel `p`, comme il se doit en passage par valeur. Cependant, ce sont bel et bien les variables `x` et `y` de l'objet désigné par `o1` qui vont êter modifiées par les affectations dans le corps de la méthode `echange`. L'impression des valeurs après l'exécution de la méthode `echange` va donc donner `"x = 20 et y = 10"`. □

Raisonnement et traces sur l'appel de méthode

La rupture de séquence engendrée par un appel de méthode est relativement simple. Il s'agit de d'interrompre la séquence d'énoncés en cours d'exécution pour aller exécuter le corps de la méthode appelée, pour revenir ensuite au fil premier de l'exécution. Quatre points d'observation peuvent être distingués :

1. Le point d'observation précédant immédiatement l'appel de la méthode.
2. Le point d'observation au début de l'exécution de la méthode.
3. Le point d'observation à la fin de l'exécution de la méthode.
4. Le point d'observation suivant immédiatement l'appel de la méthode.

Ces quatre points d'observations sont illustrés à la figure 4.8. Ils se séparent en deux groupes : le premier et le dernier qui se situent dans le contexte de l'appel de la méthode ainsi que les second et troisième qui se situent dans le contexte de la méthode appelée. Comme dans les cas précédents, ces points d'observation peuvent servir soit à l'instrumentation pour connaître l'état mémoire lors d'une exécution du programme, ou encore à l'annotation avec des assertions pour le raisonnement général sur les exécutions possibles du programme.

Exemple 46 Considérons la méthode `racines` de la classe `ExempleRacines` (chapitre 2, p. 44) :

```
public class ExempleRacines
{
  double r1, r2 ;
```

```
1.1  void racines(double a, double b, double c)
     {
1.2      double t1, t2 ;

1.3      t1 = Math.sqrt(b*b - 4*a*c) ;
1.4      t2 = 2*a ;
1.5      r1 = (-b + t1)/t2 ;
1.6      r2 = (-b - t1)/t2 ;
     }

1    public static void main(String[] args)
2    {
         ...
3        racines(1.0, 5.0, 4.0) ;
4        ...
     }
}
```

Traçons son exécution.

	r1	r2	a	b	c	t1	t2
3	*ni*	*ni*					
1.1	*ni*	*ni*	1.0	5.0	4.0	*ni*	*ni*
1.2	*ni*	*ni*	1.0	5.0	4.0	*ni*	*ni*
1.3	*ni*	*ni*	1.0	5.0	4.0	3.0	*ni*
1.4	*ni*	*ni*	1.0	5.0	4.0	3.0	2.0
1.5	-1.0	*ni*	1.0	5.0	4.0	3.0	2.0
1.6	-1.0	-4.0	1.0	5.0	4.0	3.0	2.0
4	-1.0	-4.0					

□

Cet exemple de trace rend compte du fait que le contexte appelant et le contexte appelé ne sont pas les mêmes. De fait, les variables du contexte appelé n'existe pas avant l'appel. Par exemple, si `r1` est non-initialisé avant l'appel, ce que nous notons `ni`, alors que le paramètre `a` et la variable locale `t1` n'existe pas, ce que nous montrons par l'absence de valeur. Il est également important de se rappeler que les méthodes de Java sont récursives. Chaque appel récursif possède donc ses propres variables locales et paramètres. Une trace d'une méthode récursive fait donc apparaître indépendamment les valeurs de ces variables.

L'annotation d'un appel de méthode doit aussi tenir compte du passage du contexte appelant au contexte appelé, et donc de la transmission des paramètres. Considérons d'abord l'appel d'une méthode sans résultat. Intuitivement, on s'attend à ce que le passage du contexte appelant au contexte appelé se fasse par substitution des paramètres réels aux paramètres formels dans les assertions du contexte appelant. Cette idée nous donne une première règle de transformation pour l'appel de méthode. Soit pf les paramètres formels et pr les paramètres réels, et notons A^{pf}_{pr} l'assertion obtenue en remplaçant dans A les paramètres réels par les paramètres formels correspondant. Soit `p` une procédure de corps `E` et de paramètres formels $f_1, ...f_k$, alors :

$$\text{Si } \{A^{f_1,...,f_k}_{r_1,...,r_k}\}\ \texttt{E}\ \{C^{f_1,...,f_k}_{r_1,...,r_k}\} \text{ alors}$$

$$\{A\}\quad \texttt{p(}\ r_1,...,r_k\ \texttt{)}\quad \{C\}$$

Exemple 47 Considérons la classe `Temperature` suivante qui définit des objets représentant une température à la fois en Celsius et en Farenheit :

```
public class Temperature
{
  double celsius, farenheit ;

  public void     initialise(double c)
  {
      celsius = c ;
      farenheit = 9.0*c/5.0 + 32.0 ;
  }
}
```

Nous voulons montrer que :

$\{A : \texttt{x} = 25.0\}$
`initialise(x) ;`
$\{C : \texttt{farenheit} = 77.0 \wedge \texttt{celsius} = 25.0\}$

Par la règle de déduction précédente, il suffit de montrer que :

$\{A^{\texttt{c}}_{\texttt{x}} : \texttt{c} = 25.0\}$
`celsius = c ;`
`farenheit = 9.0*c/5.0 + 32.0 ;`
$\{C^{\texttt{c}}_{\texttt{x}} : \texttt{farenheit} = 77.0 \wedge \texttt{celsius} = 25.0\}$

L'application de la règle de transformation de l'affectation sur $\{C^{\texttt{c}}_{\texttt{x}}\}$ substitue `9.0*c/5.0 + 32.0` à `farenheit`, ce qui donne $\{9.0 \times \texttt{c}/5.0 + 32.0 = 77.0 \wedge \texttt{celsius} = 30.0\}$. Après simplification, on obtient $\{\texttt{c} = 25.0 \wedge \texttt{celsius} = 25.0\}$. En appliquant à nouveau la règle de l'affectation pour traiter '`celsius = c ;`', on substitue `c` à `celsius` pour trouver $\{\texttt{c} = 25.0 \wedge \texttt{c} = 25.0\}$, ce qui est une conséquence logique de $\{A^{\texttt{c}}_{\texttt{x}}\}$. □

L'ennui avec cette première règle est qu'il n'est tenu aucun compte des pré- et postconditions sur la méthode. Idéalement, nous souhaiterions déterminer une fois pour toute les ces pré- et postconditions à partir du corps de la méthode, puis être en mesure de raisonner sur les appels à la méthode sans avoir à raisonner à chaque fois sur son corps. Soit $\{P\}$ et $\{Q\}$ respectivement la pré- et la postcondition de la méthode, la question qui se pose est de relier $\{A\}$, l'antécédent de l'appel de méthode, à $\{P\}$ et $\{C\}$, le conséquent de l'appel de méthode, à $\{Q\}$.

Rappelons que les pré- et postconditions d'une méthode, étant définie dans le contexte de cette dernière, utilise les paramètres formels et des variables externes à la méthode. Pour les replacer dans le contexte appelant, il faut intuitivement remplacer les paramètres formels par les paramètres réels. De plus, on s'attend aussi intuitivement à ce que la postcondition nous permette de démontrer le conséquent. Si $\{P\}$ et $\{Q\}$ sont les pré- et postcondition de la méthode, il a été démontré en fait la règle suivante :

$$\{P^{r_1,...,r_k}_{f_1,...,f_k} \wedge (Q^{r_1,...,r_k}_{f_1,...,f_k} \Rightarrow C)\} \quad \texttt{p(}\ r_1, ..., r_k\ \texttt{)} \quad \{C\}$$

C'est-à-dire que l'antécédent de cette règle est une condition suffisante pour établir la véracité du conséquent après l'appel de la méthode.

Exemple 48 Poursuivons avec notre exemple précédent, et posons comme pré- et postcondition sur la méthode `initialise` les assertions suivantes :

$\{P : \texttt{c} = X\}$
$\{Q : \texttt{farenheit} = 9.0 \times X/5.0 + 32.0 \wedge \texttt{celsius} = X\}$

Pour obtenir le conséquent $\{C : \texttt{farenheit} = 77.0 \wedge \texttt{celsius} = 25.0\}$, il faut donc montrer avoir l'antécédent $\{P^{\texttt{x}}_{\texttt{c}} \wedge (Q^{\texttt{x}}_{\texttt{c}} \Rightarrow C)\}$, ce qui revient à :

$\{\texttt{x} = X \wedge ((\texttt{farenheit} = 9.0 \times X/5.0 + 32.0 \wedge \texttt{celsius} = X) \Rightarrow (\texttt{farenheit} = 77.0 \wedge \texttt{celsius} = 25.0)\}$

Or, si $X = 25.0$, l'implication est vraie. L'assertion est donc une conséquence logique de l'antécédent $\{A : \texttt{x} = 25.0\}$. □

En pratique, il peut être compliqué de raisonner sur l'implication apparaissant dans l'antécédent de cette dernière règle de transformation. Heureusement, il est possible de simplifier cette règle lorsque le conséquent est composé de la postcondition et d'un certain nombre d'autres conjonctions sur des variables qui ne sont pas touchées par la méthode, c'est-à-dire si $C = Q \wedge I$, où I ne contient pas de références aux variables modifiées par la méthode. Dans ce cas, la règle suivante s'applique :

$$\{P^{r_1,...,r_k}_{f_1,...,f_k} \wedge I\} \quad \texttt{p(}\ r_1, ..., r_k\ \texttt{)} \quad \{Q^{r_1,...,r_k}_{f_1,...,f_k} \wedge I\}$$

C'est-à-dire qu'il suffit en quelque sorte de s'assurer que la précondition est vraie avant l'appel pour la «remplacer» par la postcondition dans le conséquent, en laissant le reste de l'antécédent intouché. Cette approche nous permet donc d'utiliser les pré- et postconditions sur les méthodes comme des moyens de raisonner simplement sur les appels de méthodes sans avoir à se référer au corps de la méthode elle-même.

Exemple 49 Revenons une troisième fois sur l'exemple des températures. Pour appliquer cette troisième règle, il faut transformer le conséquent pour qu'il ait la forme $C = Q \wedge I$. On le formule donc comme suit : $\{C : \texttt{farenheit} = 9.0 \times X/5.0 + 32.0 \wedge \texttt{celsius} = X \wedge X = 25.0\}$
À partir de là, on a I valant $X = 25.0$, et l'antécédent $\{P^{\texttt{x}}_{\texttt{c}} \wedge I\}$ est $\{\texttt{x} = X \wedge X = 25.0\}$, ce qui est une conséquence logique directe de l'antécédent $\{A : \texttt{x} = 25.0\}$. □

Pour les méthodes qui retournent un résultat, les postconditions peuvent utiliser la pseudo-variable `return` pour désigner la valeur du résultat. La difficulté qui se présente ici est que le contexte appelant utilise souvent le résultat de l'appel d'une méthode sans le mettre dans une variable, ce qui empêche de le désigner dans une assertion. Pour remédier à ce problème, une solution relativement simple consiste à «prétendre» que la méthode possède un paramètre supplémentaire qui va contenir le résultat et considérer cette méthode comme une procédure. Le paramètre supplémentaire peut alors se traiter comme les autres paramètres en faisant partie de la substitution.

Exemple 50 Considérons la méthode `sqrt(double x)` de la classe `Math` de Java qui calcule la racine carrée de son paramètre. La précondition de cette méthode s'exprime en fonction de son paramètre formel : $\texttt{x} \geq 0.0$ alors que sa postcondition s'exprime en terme de son paramètre formel et du résultat calculé : $\texttt{return} \times \texttt{return} = \texttt{x}$.

L'appel de cette méthode peut se faire dans le contexte du calcul des racines d'un polynôme de degré 2 où nous avons remplacé les appels de méthodes retournant un résultat par un appel procédural «équivalent» (mais qui n'est pas du Java, attention) :

```
...
calculeDiscriminant(a, b, c, disc) ;
Math.sqrt(disc, r) ;
r1 = (-b + r)/(2.0*a) ;
...
```

Le conséquent de l'appel à `Math.sqrt`, obtenu par le traitement de l'affectation à `r1` prendra vraisemblablement la forme $\{\mathtt{r} \times \mathtt{r} = disc \wedge \mathtt{a} \neq 0.0\}$ (pour tenir compte de la division par `2 * a`, qui apportera sa contrainte). Ici, on a que $Q^{\mathtt{r},\,\mathtt{disc}}_{\mathtt{return},\,\mathtt{x}} = \mathtt{r} \times \mathtt{r} = \mathtt{disc}$ et $I = \mathtt{a} \neq 0.0$. L'antécédent est donc $\{P^{\mathtt{disc}}_{\mathtt{x}} \wedge I\}$, ce qui donne $\{\mathtt{disc} \geq 0.0 \wedge \mathtt{a} \neq 0.0\}$. C'est bien la précondition attendue. En poursuivant, l'appel à `calculeDiscriminant` introduit la substitution de $\mathtt{b}^2 - 4\mathtt{ac}$ pour `disc`, d'où la précondition $\{\mathtt{b}^2 - 4\mathtt{ac} \geq 0.0 \wedge \mathtt{a} \neq 0.0\}$, ce qui correspond bien à l'antécédent nécessaire pour que le calcul de la racine réelle se passe correctement. □

4.6 Élément de savoir-faire : énoncés de flôt de contrôle en Java

4.6.1 Énoncés conditionnels en Java

Le langage Java, comme la plupart des langages, propose plusieurs énoncés conditionnels. Le principal est l'énoncé d'alternative avec partie «sinon» de la forme :

```
if ( <condition> ) {
    <énoncé-si-vrai>
} else {
    <énoncé-si-faux>
}                              //  if ( <condition> )
```

La partie `<condition>` est une expression dont le résultat doit être de type `boolean`. L'éxécution d'un tel énoncé se fait comme suit. Si la partie `<condition>` s'évalue à la valeur booléenne `true`, l'énoncé `<énoncé-alors>` est exécuté ; si par contre la partie `<condition>` s'évalue à `false`, c'est l'énoncé `<énoncé-sinon>` qui est exécuté. L'exécution du programme se poursuit ensuite avec l'énoncé suivant l'alternative. La partie «sinon» est optionnelle, c'est-à-dire qu'il existe un énoncé d'alternative sans partie «sinon». Les imbrications sont bien sûr possibles.

Exemple 51 Énoncé d'alternative avec imbrication :

```
if ( y > x ) {
    if ( x == 3 ) {
        ....
    }                          //  if ( x == 3 )
    ....
} else {
    ....
}                              //  if ( y > x )
```

□

L'expression d'alternative, comme son nom l'indique, n'est pas un énoncé, mais il mérite une mention ici dans la mesure où il produit une rupture de flôt de contrôle.[6] Sa forme syntaxique utilise l'opérateur ternaire '`_ ? _ : _`' :

```
<condition> ? <expression-alors> : <expression-sinon>
```

La partie `<condition>` est une expression dont le résultat est de type `boolean`, alors que les deux parties `<expression>` sont des expressions quelconques mais qui retournent des résultats de types compatibles. L'exécution de cette expression débute par l'évaluation de la partie `<condition>`. Si elle s'évalue à `true`, il y a exécution de la partie `<expression-alors>` et retour de son résultat comme celui de l'expression entière. Si par contre elle s'évalue à `false`, il y a plutôt exécution de la partie `<expression-sinon>` et retour de son résultat comme celui de l'expression entière. Cette expression doit s'utiliser avec modération, car elle peut mener à des programmes relativement opaques. Un exemple d'utilisation est l'affectation conditionnelle de valeur à une variable :

```
smaller = ( x < y ) ? x : y ;
```

L'énoncé de filtrage par cas de Java est lénoncé `switch` dont la forme syntaxique est :

```
switch (<varOuExpr>) {
     case <val1> :
        .... ;
        break ;
     case <val2> :
        .... ;
        break ;
     ....
     default :
        .... ;
}                                  // switch (<varOuExpr>)
```

La sélection ne peut se faire que sur une valeur qui est du type `byte`, `char`, `short` ou `int`. L'exécution de l'énoncé débute par l'évaluation de la partie `<varOuExpr>` suivie de la sélection qui consiste en une recherche séquentielle de la valeur obtenue parmi les `case`. La partie `case` identifiée, l'exécution se poursuit avec le premier énoncé suivant le `case` où se retrouve la valeur filtrée, et avec tous les énoncés jusqu'à la fin du `switch`, à moins qu'un énoncé `break` ne force la sortie de l'énoncé. La plupart du temps, chaque partie introduite par un `case` se termine par un énoncé `break`. La partie «`default :`» est exécutée si la valeur sur laquelle se fait le filtrage ne correspond à aucune des valeurs énumérées dans les «`case ... :`» ; cette partie est optionnelle. Plusieurs `case` peuvent se suivre, sans énoncés intercalés.

Exemple 52 Énoncé `switch` séparant les chiffres (0 à 9) entre chiffres pairs et impairs :

```
switch ( x ) {
   case 0 :
   case 2 :
```

[6] C'est le `if` de Scheme !

```
      case 4 :
      case 6 :
      case 8 :
         System.out.println ("Le chiffre " + x + " est pair") ;
      default :
         System.out.println ("Le chiffre " + x + " est impair") ;
   } // ---------------------------------------- switch ( x )
```

□

4.6.2 Énoncés de répétition en Java

Les énoncés de répétition, ou d'itération, existent en deux grandes familles :

1. Énoncés de répétition comptée : ces énoncés exécutent n fois un énoncé (éventuellement composé), pour un certain n déterminé au moment où l'itération débute.
2. Énoncés de répétition conditionnelle : ces énoncés exécutent un certain nombre de fois un énoncé (éventuellement composé), tant qu'une condition est vraie.

La forme syntaxique de la répétition comptée de Java est de la forme suivante[7] :

```
for (<init> ; <condition> ; <incr>)  {
    <énoncé>
}                          // for (<init> ; <condition> ;...
```

La partie `<init>` contient les énoncés d'initialisation de la répétition. La partie `<condition>` est une expression s'évaluant à un résultat de type `boolean`. La partie `<incr>` contient les énoncés faisant progresser la répétition. L'exécution de l'énoncé débute par l'exécution de la partie `<init>`. Ensuite, il y a évaluation de la partie `<condition>` ; si elle s'évalue à `true`, la partie `énoncé` est exécutée une fois, puis il y a exécution de la partie `<incr>` une fois et retour à l'évaluation de la partie `<condition>`. Si la partie `<condition>` s'évalue à `false`, la répétition est terminée et le programme se poursuit après la répétition.

Exemple 53 Accumulation dans la variable `s` de la somme des entiers de 1 à n, où la valeur n est rangée dans la variable `n` :

```
s = 0 ;
for ( i = 1 ; i <= n ; i++ ) {
    s = s + i ;
}                          // for ( i = 1 ; i <= n ; i++ )
```

□

Java propose également un énoncé de répétition conditionnelle de type `tant que -- faire` appelé `while`, comme nous l'avons vu au cours à la section 4.4. Un énoncé de répétition comptée de la forme :

[7] En réalité, cet énoncé ne se limite pas à la répétition comptée mais peut servir à toutes formes de répétition. Il est cependant d'usage de ne l'utiliser que dans le cas d'une répétition avec une initialisation et une incrémentation claire et simple.

```
for (<init> ; <condCont> ; <incr>)  {
    <énoncé>
}                              // for (<init> ; <condCont> ; ...
```

peut se transformer très facilement en une répétition conditionnelle en adoptant le schéma suivant :

```
<init> ;
while (<condCont>) {
   <énoncé> ;
   <incr> ;
}                              // while (<condCont>)
```

Java propose finalement un énoncé de type `faire -- tant que` dont la forme syntaxique est :

```
do {                           // while ( <condition> )
    <énoncé>
} while ( <condition> ) ;
```

La partie `<condition>` est une expression dont le résultat doit être de type `boolean`. L'exécution de l'énoncé débute par l'exécution de la partie `<énoncé>` une fois. Elle se poursuit ensuite par l'évaluation de la partie `<condition>` ; si elle s'évalue à `true`, on revient à l'exécution de la partie `<énoncé>`, mais si elle s'évalue à `false`, on poursuit avec l'énoncé suivant l'itération.

Cet énoncé permet d'exécuter au moins une fois le corps de la répétition. On ne doit cependant pas le confondre avec un énoncé `faire - jusqu'à` qui s'arrête lorsque la condition devient vraie. On peut généralement remplacer une répétition de type '`do ... while`' par une répétition de type '`while`' en utilisant le schéma suivant :

```
<énoncé> ;
while (<condition>) {
   <énoncé>
}                              // while (<condition>)
```

Les trois formes d'énoncés de répétition sont donc pour l'essentiel des variantes de l'énoncé '`while`'. Les langages de programmation proposent ces variantes pour permettre une écriture plus claire de certaines répétitions, mais aussi parce que la répétition comptée peut généralement être mise en œuvre de façon plus efficace que ne peut l'être sa transformation en énoncé '`while`'. Pour comprendre l'exécution des énoncés de répétition, nous avons profité de ces équivalences pour nous concentrer sur l'énoncé '`while`'.

4.6.3 Maîtrise de la répétition

Deux autres exemples de répétitions sont maintenant développés pour aider à mieux comprendre l'utilisation des assertions et de l'invariant pour raisonner sur la répétition.

Calcul du quotient et du reste

Le premier exemple est celui du calcul du quotient et du reste. Soient deux variables `x` et `y` contenant des entiers, il s'agit de calculer le quotient et le reste de la division entière de `x` par `y` et de les ranger dans les variables `q` et `r`.

L'idée générale de la solution que nous allons mettre en œuvre consiste à poser `r = x` au départ et à soustraire `y` de `r` tant que le résultat n'est pas négatif. Ce faisant, il faut compter le nombre de soustractions effectuées pour obtenir le quotient. La dernière valeur de `r` à la fin de la répétition, c'est-à-dire avant de devenir négative, sera le reste de la division. Comment coder cela sous forme d'un algorithme ? Un première proposition d'algorithme de la division entière donne :

```
r = x ;
q = 0 ;
while (r >= y) {
    r = r - y ;
    q = q + 1 ;
}
```

Quel est l'invariant de cette répétition ? L'objectif de la répétition est d'obtenir des valeurs de `q` et `r` telles que $\mathtt{x} = \mathtt{y} \times \mathtt{q} + \mathtt{r}$. Cette formule logique peut fort bien servir d'invariant. En effet, elle est vrai au point d'observation initial de la répétition puisqu'en ce point $\mathtt{r} = \mathtt{x}$ et $\mathtt{q} = 0$. Peut importe la valeur de `y`, $\mathtt{q} \times \mathtt{y}$ donnera 0, ce qui permet donc de vérifier l'invariant.

Supposons que cette formule est vraie au début du corps de la répétition. On sait déjà que ce sera le cas lors du premier passage dans le corps de la répétition puisqu'elle est vraie au point d'observation initial. Montrons qu'elle est vraie aussi à la fin du corps de la répétition :

```
r = r - y ;
q = q + 1 ;
```
$\{\mathtt{x} = \mathtt{y} \times \mathtt{q} + \mathtt{r}\}$

Le raisonnement sur l'affectation `q = q + 1` nous permet de déduire que la formule $\{\mathtt{x} = \mathtt{y} \times (\mathtt{q} + 1) + \mathtt{r}\}$ en substituant $\mathtt{q} + 1$ à `q` dans l'invariant. Si on prend cette formule comme assertion finale de l'affectation `r = r - y`, la substitution de $\mathtt{r} - \mathtt{y}$ à `r` dans cette formule donne l'assertion initiale $\{\mathtt{x} = \mathtt{y} \times (\mathtt{q}+1) + (\mathtt{r} - \mathtt{y})\}$. Une simple manipulation algébrique nous permet d'obtenir à nouveau la formule $\{\mathtt{x} = \mathtt{y} \times \mathtt{q} + \mathtt{r}\}$.

La formule $\{\mathtt{x} = \mathtt{y} \times \mathtt{q} + \mathtt{r}\}$ parait donc bien être un invariant de la répétition précédente. Cette répétition va-t-elle bien s'arrêter ? La condition de fin de la répétition est la négation de `r >= y`, donc `r < y`. Puisqu'on soustrait la valeur de `y` de `r` à chaque itération, `r` diminue jusqu'à devenir inférieur à `y` à condition que $\mathtt{y} > 0$. Cette condition est aussi nécessaire pour que la division elle-même ait un sens.

L'algorithme est-il correct ? Voyons cela avec un état initial respectant $\mathtt{x} = -5 \wedge \mathtt{y} = 2$. L'instrumentation du programme donne la trace suivante :

	`r`	`q`	`r >= y`	$\mathtt{x} = \mathtt{y} \times \mathtt{q} + \mathtt{r}$
PO initial	-5	0	faux	vrai
PO condition	-5	0	faux	vrai
PO final	-5	0	faux	vrai

Donc, l'invariant est bel et bien vrai en tout point d'observation traversé, mais algorithme ne donne pas le résultat escompté dans la mesure où 0 peut difficilement être considéré comme le quotient de -5 par 2. En fait, ce n'est pas tant que l'algorithme soit incorrect, mais plutôt qu'il y a des conditions d'utilisation implicites qui ne sont pas respectées. Cet exemple montre donc bien l'utilité des préconditions pour s'assurer d'une utilisation correcte d'une séquence d'énoncés.

L'algorithme de division entière annoté donne :

$\{\mathtt{x} \geq 0 \wedge \mathtt{y} > 0\}$
```
r = x ;
q = 0 ;
while (r >= y) {
```
$\{\mathtt{r} \geq \mathtt{y} \wedge \mathtt{x} = \mathtt{y} \times \mathtt{q} + \mathtt{r}\}$
```
   r = r - y ;
   q = q + 1 ;
```
$\{0 \leq \mathtt{r} \wedge \mathtt{x} = \mathtt{y} \times \mathtt{q} + \mathtt{r}\}$
```
}
```
$\{0 \leq \mathtt{r} < \mathtt{y} \wedge \mathtt{x} = \mathtt{y} \times \mathtt{q} + \mathtt{r}\}$

Plus grand commun diviseur

Le deuxième exemple concerne le calcul du plus grand commun diviseur (PGCD) de deux entiers contenus dans les variables `a` et `b`. Nous allons pour cela utiliser l'algorithme d'Euclide. L'idée générale de cet algorithme consiste à définir deux variables `x` et `y` contenant initialement les valeurs de `a` et `b`. Ensuite l'algorithme soustrait répétitivement la plus petite valeur de l'autre variable jusqu'à ce que les deux variables aient la même valeur ; cette valeur sera le PGCD des valeurs de `a` et `b`.

Pourquoi cet algorithme fonctionne-t-il ? Soit p le PGCD de `a` et `b` et $\mathtt{x} > \mathtt{y}$. Initialement, p est le PGCD de `x` et `y` puisqu'elles contiennent les valeurs de `a` et `b`. Soient $\mathtt{x} = k_1 p$ et $\mathtt{y} = k_2 p$ alors $\mathtt{x} - \mathtt{y} = (k_1 - k_2)p$. Donc le PGCD de `a` et `b` est aussi le PGCD de $\mathtt{x} - \mathtt{y}$ et de `y`. Comme au moins l'une des deux valeurs diminue à chaque étape et que la soustraction de la plus petite valeur de la plus grande garantit que $\mathtt{x} > 0$ et $\mathtt{y} > 0$, on doit finir par sortir de la répétition. Comme par ailleurs le PGCD d'un nombre et de lui-même est justement lui-même, on doit finir par trouver $\mathtt{x} = \mathtt{y}$. L'algorithme d'Euclide s'écrit de la manière suivante :

```
x = a ;
y = b ;
while (x != y) {
    if (x > y) {
        x = x - y ;
    } else {
        y = y - x ;
    }
}
```

La précondition qui assure que l'algorithme fonctionne consiste simplement à exiger que `a` et `b` soient strictement positifs : $\{\mathtt{a} > 0 \wedge \mathtt{b} > 0\}$. L'invariant est plus difficile à trouver mais ressort de la discussion précédente. En fait, ce qui est vrai en tout point d'observation est que le PGCD de `x` et `y` est toujours égal au PGCD de `a` et `b` : $\{pgcd(\mathtt{x}, \mathtt{y}) = pgcd(\mathtt{a}, \mathtt{b})\}$. L'argumentation précédente sur le fonctionnement de l'algorithme nous convainc qu'il s'agit bien d'un invariant. La même argumentation nous permet de dire que la répétition va nécessairement se terminer, puisque la valeur $max(\mathtt{x}, \mathtt{y})$, bornée inférieurement par 0, diminue strictement à chaque itération. L'assertion finale est : $\{\mathtt{x} = \mathtt{y} = pgcd(\mathtt{a}, \mathtt{b})\}$.

L'annotation de l'algorithme d'Euclide illustre un cas qui se produit assez régulièrement, à savoir que l'invariant est trop fort pour être utilisé directement dans un programme. En effet,

le but de l'annotation est de produire des expressions logiques exécutables, et donc vérifiables à l'exécution. La difficulté d'utiliser l'invariant de l'algorithme d'Euclide est qu'il faudrait connaître le PGCD de `a` `b` alors qu'il s'agit justement de le calculer ! Dans ce genre de cas, on peut utiliser une anotation avec un invariant affaibli mais suffisant pour vérifier l'exactitude de l'algorithme. Dans le cas d'Euclide, on utilise la précondition appliqué aux valeurs de `x` et `y` comme invariant affaibli :

$\{\texttt{a} > 0 \wedge \texttt{b} > 0\}$
```
x = a;
y = b;
while (x != y) {
```
$\{\texttt{x} \neq \texttt{y} \wedge \texttt{x} > 0 \wedge \texttt{y} > 0\}$
```
   if (x > y) {
      x = x - y;
   } else {
      y = y - x;
   }
}
```
$\{\texttt{x} = \texttt{y}\}$

Ces assertions sont suffisantes pour garantir le bon comportement de l'algorithme.

4.7 Exercices

4.7.1. Écrivez un énoncé d'alternative qui, selon la valeur d'un nombre `n`, imprime `"la valeur de n est positive"`, `"la valeur de n est égale à zéro"`, ou `"la valeur de n est négative"`.

4.7.2. Écrivez un énoncé d'alternative qui, étant donné trois variables `a`, `b` et `c`, range dans une variable `somme` la valeur de `a`, `b` ou `c` si cette valeur est égale à la somme des valeurs des deux autres variables. On range également `true` dans la variable `trouve` si on a effectivement trouvé que l'une des valeurs était égale à la somme des deux autres et `false` sinon.

4.7.3. Un guichet automatique de la SNCF doit calculer la réduction à laquelle ont droit les clients. Trois réductions peuvent s'appliquer : (long)

1. une réduction dite famille nombreuse de 30% si le client a 3 enfants ou plus,
2. une réduction dite couple de 15% si le(s) client(s) voyage(nt) en couple,
3. une réduction dite personne âgée de 25% si le client plus de 65 ans.

De plus, des règles de cumul sont édictées. Un client de plus de 65 ans peut éventuellement cumuler sa réduction de personne âgée avec la réduction couple ; la réduction couple peut aussi être cumulée avec la réduction famille nombreuse. Par contre, les réductions famille nombreuse et personne âgée sont mutuellement exclusives ; c'est alors la plus avantageuse qui est appliquée. En supposant que les trois variables booléennes suivantes sont définies et initialisées en fonction de la situation du client :
– `a3enfantsOuPlus` : vraie si le client a 3 enfants ou plus.
– `estEnCouple` : vraie si le client voyage en couple.
– `estAge` : vraie si le client a plus de 65 ans.
Écrivez la séquence d'énoncé qui calcule dans une variable `reduction` le pourcentage de réduction auquel a droit le client.

4.7.4. Écrivez une séquence d'énoncés qui, étant donné un entier `n`, range dans la variable diviseur le plus grand diviseur de `n`.

4.7.5. Écrivez une séquence d'énoncés qui, étant donné un certain entier positif `nombre`, range dans la variable `s` la somme de ses chiffres. Par exemple, la somme des chiffres de 123 est 6.

(long) **4.7.6. Intégrale selon la méthode des trapèzes.** Écrire une classe racine `Integrale` comportant une méthode `double f(double)` représentant une fonction à intégrer et qui dans sa méthode `main` réalise cette intégration en utilisant la méthode des trapèzes.

Méthode des trapèzes en m sous-intervalles pour calculer l'intégrale de la fonction f sur l'intervalle $[a, b]$, avec $h = \frac{(b-a)}{m}$ et $x_j = a + jh$:

$$\int_a^b f(x)\,dx = \frac{h}{2}[f(a) + f(b) + 2\sum_{j=1}^{m-1} f(x_j)] - \frac{(b-a)h^2}{12} f''(\mu)$$

pour un certain $\mu \in [a, b]$.

Note : le terme en $f''(\mu)$ exprime l'erreur ; on peut le négliger ici.

(long) **4.7.7. Intégrale selon la méthode de Simpson.** Reprendre l'exercice précédent mais cette fois-ci avec la méthode de Simpson.

Méthode de Simpson en $2m$ sous-intervalles pour calculer l'intégrale de la fonction f sur l'intervalle $[a, b]$, avec $h = \frac{(b-a)}{2m}$ et $a = x_0 < x_1 < \ldots < x_{2m} = b$, où $x_j = a + jh$ pour $j = 0, 1, \ldots, 2m$:

$$\int_a^b f(x)\,dx = \sum_{j=1}^{m} \{\frac{h}{3}[f(x_{2j-2}) + 4f(x_{2j-1}) + f(x_{2j})] - \frac{h^5}{90} f^{(4)}(\mu)\}$$

pour un certain $\mu \in [x_{2j-2}, x_{2j}]$.

Note : le terme en $f^{(4)}(\mu)$ exprime l'erreur ; on peut le négliger ici.

4.7.8. La séquence suivante calcule le plus grand entier inférieur ou égal à une certaine valeur n donnée (contenue dans une variable `n`), qui soit une puissance de 2 :

```
i = 1;
while (2 * i <= n) {
    i = 2 * i ;
}
```

Proposez une formule logique qui représente le résultat attendu de ce calcul et tirez-en un invariant pour la répétition. Y a-t-il une condition initiale pour que la séquence fonctionne ?

4.7.9. Soit une fonction continue f sur l'intervalle $[\lfloor a \rfloor, \lfloor b \rfloor + 1]$ tel que[8] $f(\lfloor a \rfloor)$ et $f(\lfloor b \rfloor + 1)$ ont des signes opposés. Le problème consiste à trouver une valeur k entière telle que $\lfloor a \rfloor \leq k \leq (\lfloor b \rfloor + 1)$ et $f(k)$ et $f(k+1)$ ont des signes opposés. Les valeurs k et $k+1$ sont alors les deux valeurs entières qui encadre au plus près un zéro de la fonction f se situant dans l'intervalle $[\lfloor a \rfloor, \lfloor b \rfloor + 1]$. Un algorithme efficace pour trouver une telle valeur i utilise le principe de la bisection : à chaque étape de calcul, on débute avec un intervalle entier $[i, j]$ encadrant un zéro de f et le but de l'étape courante est de diminuer la taille de l'intervalle par deux en prenant la valeur médiane m de l'intervalle $[i, j]$ et en déterminant dans quel sous-intervalle $[i, m]$ ou $[m, j]$ se situe le zéro. cet algorithme est efficace car on peut montrer qu'au plus $\log_2((\lfloor b \rfloor + 1) - \lfloor a \rfloor)$ étapes sont nécessaires pour obtenir la valeur k cherchée.
Écrivez et annotez un algorithme de recherche de k par bisection.

[8] $\lfloor a \rfloor$ dénote la partie entière de a.

Chapitre 5

Les tableaux

L'ordinateur tire sa puissance en bonne partie de sa capacité à traiter de grandes quantités de données. Jusqu'à maintenant, nous n'avons vu que des données individuelles, ou en terme mathématique *scalaires*. Chacune de ces données scalaires est rangée dans un mot mémoire désigné par une variable. Lorsque de grandes quantités de données doivent être traitées, il est impossible de désigner individuellement chacune de ces données par une variable. Il devient nécessaire de les désigner à l'aide de collections vues comme des entités propres. Ces collections sont aux données individuelles ce que l'ensemble, le vecteur ou la matrice est à l'élément en mathématique. Cela dit, là où l'ensemble mathématique ne propose aucune structure et une cardinalité variable, les collections de données informatiques sont souvent ordonnées (on peut parler du premier élément, du dernier, du suivant, etc.) et elles ont une capacité finie et souvent fixe. C'est le cas des tableaux, les collections de données les plus simples proposées dans les langages impératifs.

5.1 Agrégation de données : la notion de tableau

De nombreux problèmes nécessitent l'utilisation de collections de données pour être résolus par informatique. Considérons un exemple simple de statistiques météorologiques illustrant cette nécessité. Supposons que l'on demande d'écrire une portion de programme qui va lire au terminal des données de température quotidienne, jusqu'à ce qu'une valeur sentinelle inférieure à -500.0 soit entrée, et alors en imprimera la moyenne. En utilisant les concepts que nous avons déjà vu jusqu'ici, il est relativement simple d'écrire ce bout de programme qui accumule dans une variable `total` la somme des températures et compte dans `nblu` le nombre de valeurs lues[1] :

```
int    nblu ;
double t, total ;

total = 0.0 ;
nblu = 0 ;
t = "prochaine température lue"  ;
while (t >= -500.0) {
    total = total + t ;
    nblu = nblu + 1 ;
    t = "prochaine température lue"  ;
```

[1] N'ayant pas encore vu la lecture de valeurs au terminal, ce qui sera fait au chapitre 7, cette ligne reste sous une forme descriptive.

```
}
System.out.println(total / nblu) ;
```

Ce problème peut donc être traité avec deux variables parce qu'il n'est pas nécessaire de mémoriser les données pour le résoudre. Une modification a priori simple du problème va bousculer cet état de fait. Supposons maintenant qu'il faut lire au terminal des données de température quotidienne , jusqu'à ce qu'une valeur sentinelle inférieure à -500.0 soit entrée, en calculer la moyenne et afficher pour chaque jour la température et l'écart avec la moyenne. On constate rapidement que l'on doit connaître toutes les températures pour calculer la moyenne. Or, on ne peut calculer et imprimer l'écart de chaque température avec la moyenne sans avoir calculé au préalable la moyenne. Il faut donc mémoriser les températures au fur et à mesure du calcul de la moyenne pour revenir ensuite calculer les écarts avec cette moyenne. Mais comment faire ?

On ne connaît pas à l'avance le nombre de valeurs lues, mais si on peut le borner, une solution simple à notre disposition consiste à créer sept variables, par exemple `t1` à `t`n pour une borne n, à y lire les données, puis à les utiliser pour calculer les écarts avec la moyenne. Cette solution, possible dans le cas où le nombre de valeur est très limité, ne supporte pas un passage à une échelle plus grande. Qu'arriverait-il en effet s'il fallait traiter les températures de l'année ? ou des 10 dernières années ? Il faudrait créer 365 variables, voire plus de 3650 dans le second cas ! Cette approche n'est ni pratique, ni pratiquable. La solution est de disposer d'un moyen de mémoriser toutes les données dans une collection dont la taille peut s'adapter au problème.

Tableaux

Le tableau est la collection de données la plus courante dans les langages impératifs.[2] Bien que moins central en programmation par objets, la plupart des langages à objets offrent la notion de tableau. La notion de tableau n'est pas sans rappeler les vecteurs et les matrices de l'algèbre linéaire dont elle emprunte les principales intuitions.

> **tableau :** ensemble fini et ordonné de valeurs du même type, chacune de ces valeurs constituant un élément du tableau directement accessible par un indice.

Le tableau est une collection finie dans le sens où il ne contiendra jamais qu'un nombre fini d'éléments, nombre qui est déterminé lors de sa création. Le tableau est aussi une collection *homogène* de valeurs, car toutes les valeurs contenues dans un tableau sont du même type. On parle de tableau d'entiers, de réels, voire d'objets (rectangles, tortues, etc.).

Chaque élément d'un tableau est directement accessible par son *indice*. L'indice i est un numéro séquentiel désignant le i^{e}élément du tableau. Dans ce sens, le tableau est une collection ordonnée, puisque l'on peut parler du premier élément, du i^{e}, ou encore du dernier. Selon les langages de programmation, les valeurs admissibles comme indice peuvent varier. En Java, les indices sont des entiers entre 0 et $n-1$, où n est le nombre d'éléments du tableau. D'autres langages utilisent les entiers de 1 à n, voire peuvent admettre l'utilisation de valeurs de types discrets muni d'un ordre total (les jours de la semaine, les mois de l'année, les lettres de l'alphabet, etc.).

La figure 5.1 donne un exemple de tableau à partir du problème des températures qui nous a servi de motivation. La mise en œuvre des tableaux nécessitent d'allouer l'espace nécessaire

[2] On peut dire que le tableau est aux langages impératifs ce que la liste est aux langages fonctionnelles : la collection de données la plus facilement et la plus couramment utilisée.

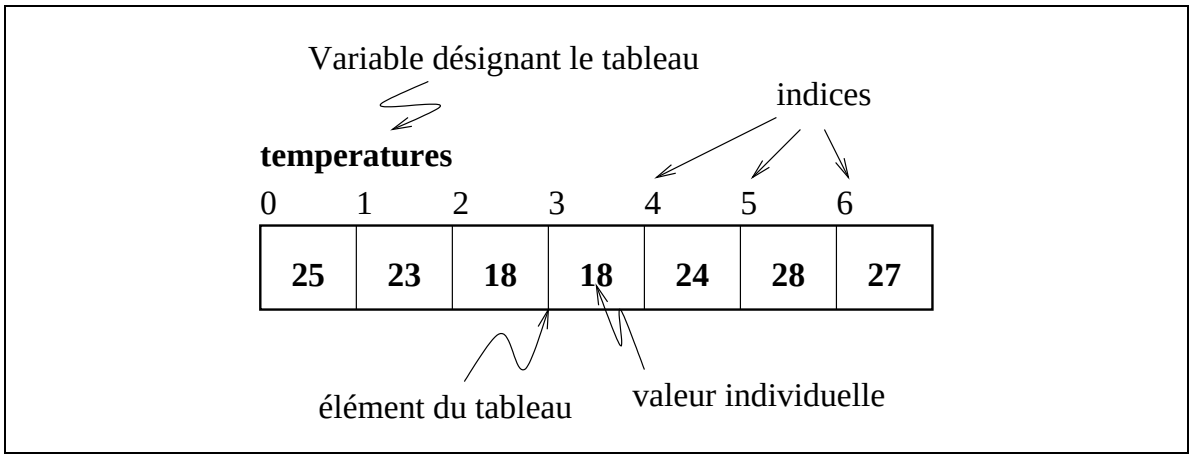


FIG. 5.1 – Tableau de températures

pour mémoriser toutes les valeurs du tableau. Dans le cas de l'exemple des températures, le tableau doit pouvoir contenir sept valeurs de type réel. Une valeur de type réel exige un mot mémoire ; le tableau des températures demandent donc l'allocation de sept mots mémoire. La déclaration d'un tableau en Java se fait en utilisant comme type le type des éléments du tableau suivi de crochets pour indiquer un type tableau :

```
double[] temperatures ;    // temperatures peut contenir un
                           // tableau de réels
```

La création d'un tableau se fait par une expression `new` où le type de tableau à créer indique le nombre d'élements que le nouveau tableau va contenir :

```
temperatures = new double[7] ;  // temperatures contient un
                                // tableau de 7 réels
```

Cette expression crée le tableau, mais pas les éléments du tableau. Les éléments du tableau sont initialisés aux valeurs par défaut spécifiées par Java (`null` pour les tableaux d'objets, et la valeur par défaut pour les tableaux de types primitifs). Les indices 0 à 6 référent à chacun de ces mots individuellement. L'ensemble du tableau est désigné par la variable `temperatures`. L'opération d'indiçage permet de référer à chacune des sept valeurs :

- La notation `temperatures[i]` utilisée dans une expression représente la valeur du $(\texttt{i}+1)^{\text{e}}$ élément du tableau `temperatures`.[3]
- L'énoncé d'affectation '`temperatures[i] = <expression>;`' indique qu'il faut ranger dans le $(\texttt{i}+1)^{\text{e}}$ élément du tableau `temperatures` la valeur calculée à partir de `<expression>`.

La notation de l'opération d'indiçage s'utilise donc comme une variable. Dans une expression, elle désigne la valeur contenue dans l'emplacement mémoire référencé, alors que dans une affectation, elle désigne l'adresse de cet emplacement mémoire pour qu'on puisse y ranger une valeur. Indicer un tableau en dehors de ses bornes (0 à $n-1$) est toujours une erreur de logique de programme. Certains langages de programmation, comme Java, déclenchent explicitement une erreur et arrête le programme s'il fait une tentative d'indiçage hors bornes. D'autres langages ne vérifient pas cette erreur et s'en remettent au programmeur pour que cela ne se produise pas.

[3]En prenant pour base l'indice 0 comme le premier élément.

Si on revient au problème des températures, les notions introduites sur les tableaux nous permettent d'ores et déjà de donner une solution satisfaisante :

```
double[] temperature = new double[NOMBRE_MAX] ;
double total ;
int nblu, i ;

total = 0.0 ;
nblu = 0 ;
temperatures[0] = "prochaine température lue"  ;
while (temperatures[nblu] >= -500.0 && nblu < NOMBRE_MAX - 1) {
    total = total + temperatures[nblu] ;
    nblu = nblu + 1 ;
    temperatures[nblu] = "prochaine température lue"  ;
}
moyenne = total / nblu :
for (i = 0 ; i < nblu ; i++) {
    System.out.print(temperatures[i]) ;
    System.out.println(temperatures[i] - moyenne) ;
}
```

Tableaux multidimensionnels

Tout comme l'algèbre linéaire nous a appris à manipuler des vecteurs et des matrices, les tableaux dans les langages de programmation peuvent avoir plusieurs dimensions. Un tableau à deux dimensions s'assimile à une matrice. Par exemple, une matrice de $m \times n$ d'entiers peut se représenter par un tableau d'entiers à deux dimensions dont les bornes maximales sont m dans la première dimension et n dans la seconde. L'opération d'indiçage s'étend facilement au cas multidimensionnel. Pour un tableau à deux dimensions `t`, la notation `t[i][j]` désigne le `j`$^{\text{e}}$ élément de la `i`$^{\text{e}}$ ligne de `t`. Cette notation se généralise immédiatement à n dimensions sous la forme :

`tableau[`i_1`][`i_2`] ... [`i_n`]`

Déclaration/création des tableaux

Comme toute autre forme de données d'un langage, un tableau est défini par un type. Une variable qui contiendra une valeur de type tableau doit d'abord être déclarée. Ensuite, une valeur de type tableau peut être créée et rangée dans cette variable. La déclaration et la création d'un tableau doivent déterminer un certain nombre d'informations définissant complètement le tableau :

- le type des éléments,
- le nombre de dimensions, et
- le nombre d'éléments (par dimension).

Les éléments absolument nécessaires à la déclaration du tableau sont le type de ses éléments et le nombre de ses dimensions. Le nombre d'éléments n'est utile que lors de la création du tableau et lors de son utilisation subséquente. Dans certains langages de programmation, la création du tableau est indissociable de sa déclaration. Dans ces langages, l'espace nécessaire pour le tableau est alloué dès la déclaration d'une variable de type tableau. Dans d'autres langages, la déclaration est dissociée de la création ; c'est le cas de Java comme nous le verrons dans la section 5.5.

Initialisation statique

L'initialisation d'un tableau se fait généralement par l'affectation de valeurs aux différents éléments lors de l'exécution du programme. Certains langages permettent cependant l'initialisation complète d'un tableau par des valeurs constantes. Pour un tableau dont le type des valeurs et le nombre de dimension est déterminé, les constantes données comme initialisation servent à déterminer la taille du tableau. Dans le cas d'un tableau à une dimension, il suffit d'énumérer les valeurs dans l'ordre croissant des indices. La syntaxe (Java) pour faire cette énumération est la suivante :

```
{1, 2, 3, 4, 5, 6, 7}
```

Cette énumération de valeurs est utilisé dans le contexte de la création d'un tableau unidimensionnel d'entiers. Cela peut se faire soit lors de la déclaration d'une variable de type tableau ou encore par l'utilisation explicite de l'opérateur `new` (voir §5.5). de sept valeurs et l'initialisation des éléments de ce tableau à l'aide des valeurs 1 à 7. Le cas multidimensionnel s'obtient par simple généralisation. L'énumération des éléments de la dernière dimension suit la notation précédente. Pour énumérer les valeurs des dimensions précédentes, il suffit d'énumérer entre accolades les valeurs des dimensions suivantes (de gauche à droite dans la déclaration). Pour deux dimensions, cela donne l'énumération suivante :

```
{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}}
```

Cette notation se généralise directement au cas de n dimensions, avec $n > 2$.

5.2 Quelques algorithmes classiques sur les tableaux

Les tableaux formant la collection de données par excellence des langages impératifs, il va de soi que de nombreux problèmes ont, par le passé, été traités en les utilisant. De ces problèmes ressortent quelques traitements maintenant devenus classiques car les solutions aux problèmes pratiques sont souvent des variantes assez simples de ces algorithmes de base, ou du moins en empruntent les intuitions de base.

5.2.1 Traversée d'un tableau

Le problème le plus classique consiste à traverser le tableau pour traiter chacun de ses éléments. Dans le cas unidimensionnel, il s'agit de parcourir les éléments du tableau un à un et de leur appliquer un traitement donné. Le patron se généralise à partir du problème des températures quotidiennes vu au début du chapitre. Il suffit d'utiliser une variable i servant d'indice, et donc variant entre 0 et $n-1$ (n étant la taille du tableau), pour accéder aux éléments individuels du tableau.

```
for (i = 0 ; i <= (n − 1) ; i++) {
    {inv : ∀k ∈ [0, i [ (t[k] est traité) }
    traiter t[i] ;
}
```

L'invariant assure que les éléments d'indice de 0 à `i - 1` ont déjà été traités. À la fin de la répétition, on a que `i` est égal à n, donc tous les éléments ont été traités.

Traverser un tableau à plusieurs dimensions

La traversée d'un tableau se généralise au cas multidimensionnel en balayant toutes les dimensions et dans chaque dimension en balayant chaque éléments. Sur le cas à deux dimensions, on peut balayer le tableau ligne par ligne (première dimension) et sur chaque ligne balayer toutes les colonnes (seconde dimension). Pour cela, on utilise deux variables, par exemple `i` et `j`, jouant les rôles d'indices selon chacune des dimensions. Deux répétitions imbriquées suffisent à balayer toutes les valeurs du tableau :

```
for (i = 0 ; i <= (m − 1) ; i++) {
   for (j = 0 ; j <= (n − 1) j++) {
      traiter l'élément t[i][j] ;
   }
}
```

Cette approche se généralise à n dimensions en ajoutant autant de niveaux d'imbrication des répétitions qu'il y a de dimensions à balayer. Choisir de faire varier le premier indice d'abord est une question de choix en fonction du traitement à faire.

5.2.2 Recherche séquentielle

Un tableau peut servir de moyen pour mémoriser une collection de données. L'exemple des températures illustre bien ce besoin. Dans ce cas, il arrive très souvent qu'il faille rechercher si une valeur apparaît bien dans la collection. Dans certains cas, il suffit de retourner simplement vrai ou faux selon que la valeur a été trouvée ou non ; dans d'autres cas, il faut retourner l'indice auquel la valeur a été trouvée pour y référer par la suite. Considérons le cas simple où on se contente de retourner vrai ou faux.

L'énoncé de problème est le suivant. Étant donné un tableau `t` de n valeurs, retourner vrai ou faux selon qu'une valeur contenue dans la variable `x` y apparaît ou non. L'idée générale de l'algorithme consiste à parcourir le tableau à partir de l'indice 0 et arrêter dès que l'on rencontre une valeur égale à `x`. Si on s'arrête avant la fin du tableau, on a trouvé la valeur. Si on s'arrête après la fin du tableau (`i` $= n$), alors la valeur n'est pas dans le tableau. Cette idée générale mène au code suivant, où `n` contient la taille du tableau `t` :

```
i = 0 ;
while (i < n && t[i] != x) {
   i = i + 1 ;
}
resultat = i < n && t[i] == x ;
```

On initialise `i` à 0, puis on entre dans la répétition qui va continuer tant que la fin du tableau n'est pas atteinte (`i < n`) et que l'on n'a pas rencontré la valeur de `x` (`t[i] != x`). Il faut noter que si `i` est plus grand ou égal à `n`, alors on est en dehors des bornes d'indiçage correct et que `t[i]` n'existe pas, donc ne peut être égal à `x`.

L'argumentation selon laquelle cet algorithme est correct est basé sur l'annotation donnée à la figure 5.2. L'invariant dit que la valeur de l'indice `i` peut varier entre 0 et `n` et que toutes les valeurs d'indice strictement inférieur à `i` sont différentes de la valeur cherchée. Dès l'entrée dans la répétition, sachant donc que l'expression de contrôle est vraie, on en déduit que l'assertion $a1$ disant que `i` est strictement inférieure à `n`, que `t[i]` $\neq$ `x` et par hypothèse que l'invariant est vrai (c'est le cas à la première itération, il faut encore prouver que cela est vrai à chaque itération). La valeur `t[i]` étant différente de `x`, on en conclut ($a2$) maintenant que toutes les valeurs d'indice 0 à `i` inclusivement sont différentes de `x`. L'assertion $a3$ vise

```
i = 0 ;
{inv : 0 ≤ i ≤ n ∧ (∀k ∈ [0, i[ t[k] ≠ x)}
{i = 0 ∧ inv}
while (i < n && t[i] != x) {
    {a1 : i < n ∧ t[i] ≠ x ∧ inv}
    {a2 : 0 ≤ i < n ∧ (∀k ∈ [0, i] t[k] ≠ x)}
    {a3 : 0 < i + 1 ≤ n ∧ (∀k ∈ [0, i + 1[ t[k] ≠ x)}
    i = i + 1 ;
    {a4 : 0 < i ≤ n ∧ (∀k ∈ [0, i[ t[k] ≠ x)}
}
{a5 : (i ≥ n ∨ t[i] = x) ∧ (0 ≤ i ≤ n ∧ (∀k ∈ [0, i[ t[k] ≠ x))}
{a6 : (i = n ∧ (∀k ∈ [0, i[ t[k] ≠ x))∨
          (0 ≤ i ≤ n ∧ t[i] = x ∧ (∀k ∈ [0, i[ t[k] ≠ x))}
{a7 : (i = n ∧ (∀k ∈ [0, i[ t[k] ≠ x) ∨ (0 ≤ i ≤ n ∧ t[i] = x))}
resultat = i < n && t[i] == x ;
```

FIG. 5.2 – Algorithme de recherche séquentielle annoté

simplement à préparer le terrain pour appliquer le raisonnement sur l'affectation '`i = i + 1`'. On fait donc apparaître '`i + 1`' à la place de `i` en modifiant inégalités et bornes d'intervalle en conséquence. La substitution de '`i + 1`' par `i` dans $a3$ permet d'obtenir l'assertion de fin de corps de l'itération $a4$, qui elle-même entraîne la validité de l'invariant en début de répétition.

L'invariant ayant été démontré, on passe à la suite de la répétition en déduisant l'assertion $a5$ de la négation de l'expression de contrôle de la répétition et de l'invariant. L'assertion $a6$ est obtenue en appliquant la distributivité de $\wedge$ sur $\vee$ et en simplifiant les inégalités. Notons que dans le second terme du ou logique, `i` ne peut être égal à `n` et en même temps avoir `t[i]` $=$ `x`, car `i` serait alors en dehors des bornes d'indiçage. L'assertion $a7$ simplifie $a6$ dans le second terme puisqu'il nous importe peu que la valeur de `x` ne se trouve pas aux indices inférieurs à `i` si effectivement on la trouve à l'indice `i`. De $a7$, on conclut que soit `i` est inférieur à `n` et alors qu'on a trouvé la valeur de `x` dans le tableau, soit `i` n'est pas inférieur à `n` et alors on n'a pas trouvé la valeur de `x`.

5.2.3 Plus grand élément d'un tableau

Le troisième algorithme classique que nous allons étudier est la recherche du plus grand élément d'un tableau. Il s'agit du prototype du problème plus général qui consiste à trouver parmi les éléments du tableau un élément qui ait une certaine propriété. Nous allons nous concentrer sur le cas unidimensionnel ; le cas multidimensionnel s'obtient facilement par adaptation de l'algorithme de traversée. L'idée de l'algorithme est de balayer le tableau dans l'ordre des indices et conserver dans une variable `plusGrand` la plus grande valeur vue depuis le début du tableau ; à la fin, `plusGrand` doit contenir la valeur maximale cherchée. Cette idée se concrétise dans l'algorithme suivant :

```
{n > 0}
plusGrand = t[0] ;
i = 1 ;
{inv : 0 < i ≤ n ∧ plusGrand = M ∧ (∀k ∈ [0, i[ M ≥ t[k])}
while (i <= n − 1 ) {
   {i ≤ n − 1 ∧ 0 < i ≤ n ∧ plusGrand = M ∧ (∀k ∈ [0, i[ M ≥ t[k])}
   if (t[i] > plusGrand) {
      {t[i] > M ∧ 0 < i < n ∧ plusGrand = M ∧ (∀k ∈ [0, i[ M ≥ t[k])}
      plusGrand = t[i] ;
      {a1 : t[i] > M ∧ 0 < i < n ∧ plusGrand = t[i] ∧ (∀k ∈ [0, i[ M ≥ t[k])}
   } else {
      {t[i] ≤ M ∧ 0 < i < n ∧ plusGrand = M ∧ (∀k ∈ [0, i[ M ≥ t[k])}
      continuer ;
      {a2 : t[i] ≤ M ∧ 0 < i < n ∧ plusGrand = M ∧ (∀k ∈ [0, i[ M ≥ t[k])}
   }
   {a3 : 0 < i < n ∧ plusGrand = max(M, t[i]) ∧ (∀k ∈ [0, i[ M ≥ t[k])}
   {0 < i < n ∧ plusGrand = M' ∧ (∀k ∈ [0, i] M' ≥ t[k])}
   {0 < i + 1 ≤ n ∧ plusGrand = M ∧ (∀k ∈ [0, i + 1[ M ≥ t[k])}
   i = i + 1 ;
   {0 < i ≤ n ∧ plusGrand = M ∧ (∀k ∈ [0, i[ M ≥ t[k])}
}
{i > n − 1 ∧ 0 < i ≤ n ∧ plusGrand = M ∧ (∀k ∈ [0, i[ M ≥ t[k])}
{i = n ∧ plusGrand = M ∧ (∀k ∈ [0, n[ M ≥ t[k])}
resultat = plusGrand ;
```

FIG. 5.3 – Algorithme du plus grand élément d'un tableau annoté

```
plusGrand = t[0] ;
for (i = 1 ; i <= n − 1 ; i++) {
   if (t[i] > plusGrand) {
      plusGrand = t[i] ;
   }
}
resultat = plusGrand ;
```

L'initialisation de `plusGrand` à `t[0]` suppose qu'il y a au moins un élément dans le tableau et part du principe que sur la portion de tableau allant de l'indice 0 à 0 inclusivement, la plus grande valeur est `t[0]`. L'algorithme procède par approximation succssive où à chaque itération, la plage des indices couverts pour trouver la valeur la plus grande s'élargit peu à peu. L'invariant, apparaissant à la figure 5.3, indique bien que les valeurs dans la plage d'indice allant de 0 à `i - 1` sont toutes inférieures ou égales à `plusGrand`.

Outre les techniques de raisonnement déjà utilisées dans les algorithmes précédents, le point le plus délicat ici est le passage des assertions $a1$ et $a2$ obtenues dans les deux parties de l'aternative à l'assertion $a3$. L'assertion $a3$ est le ou logique des deux autres. C'est de (`t[i]` $> M \wedge$ `plusGrand` $=$ `t[i]`) $\vee$ (`t[i]` $\leq M \wedge$ `plusGrand` $= M$) que l'on obtient que `plusGrand` $= max(M,$ `t[i]`$)$. Il suffit ensuite d'observer qu'en posant $M' = max(M,$ `t[i]`$)$ pour obtenir que `plusGrand` contient maintenant la valeur la plus grande sur la plage d'indice 0 à `i` inclusivement. Le reste du raisonnement suit par simple manipulation d'inégalités et de bornes d'intervalles.

5.2.4 Recherche dichotomique

L'algorithme de recherche séquentielle que nous avons vu permet de trouver une valeur dans une collection qui n'a pas de structure particulière. Il est cependant peu efficace. En effet, on peut se demander combien d'éléments du tableau, en moyenne, l'algorithme compare à `x` avant de sortir de la répétition. Supposons d'abord que la valeur de `x` est bel et bien dans le tableau. Si on suppose que la valeur est tirée au hasard de façon équiprobable entre les différents indices, tantôt la valeur sera trouvée dans un élément d'indice faible (0, 1, 2, ...), tantôt dans un élément d'indice élevé (n, $n-1$, $n-2$, ...). Le nombre de valeurs comparées à `x` étant égal à l'indice où cette valeur se trouve, en moyenne on devrait comparer $n/2$ valeurs.

Maintenant, si la valeur de `x` n'est pas dans le tableau, l'algorithme doit la comparer à toutes les valeurs du tableau avant de décider qu'elle n'y apparait pas. Ainsi, à chaque fois que la valeur de `x` n'est pas dans le tableau, on fait n comparaisons. Si une fois sur deux la valeur n'est pas ans le tableau, on a n comparaisons pour la moitié des recherches et $n/2$ en moyenne l'autre moitié. On a donc alors $\frac{3n}{4}$ comparaisons en moyenne. Si la collection contient plusieurs centaines de milliers de valeurs, la recherche risque d'être assez longue.

Comment faire pour être plus efficace dans la recherche ? Comme souvent face à ce genre d'interrogation, l'informaticien doit faire quelques hypothèses supplémentaires sur la structure du problème, hypothèses qui lui serviront ensuite à concevoir des algorithmes plus rapides. Dans notre cas, posons comme hypothèse sur les valeurs du tableau qu'elles sont ordonnées de manière croissante en fonction ds indices, c'est-à-dire :

$$\forall i \in [0, n-1[\ \ (\texttt{t[}i\texttt{]} \leq \texttt{t[}i+1\texttt{]})$$

De la même façon que la recherche dans un dictionnaire tire partie du fait que les mots sont dans l'ordre croissant, on peut tirer partie de cette hypothèse pour faire une recherche plus efficace, une recherche dite dichotomique. L'idée de la recherche dichotomique est la suivante. Il s'agit d'encadrer la valeur `x` dans un intervalle `[i,j]` où `i` et `j` sont des variables qui vont servir d'indice dans le tableau. C'est-à-dire que l'on aura `t[i]` $\leq$ `x` $\leq$ `t[j]`. À chaque itération, l'algorithme de recherche dichotomique va couper l'intervalle en deux et déterminer le sous-intervalle qui encadre la valeur de `x`. On répète le processus sur ce sous-intervalle, jusqu'à ce que le sous-intervalle ne contienne plus qu'une seule valeur (c'est-à-dire `i` est égal à `j`), `x`, ou que l'on sache que la valeur de `x` n'est pas dans le tableau.

D'un point de vue de la performance, on voit que cet algorithme va exécuter un nombre d'itérations qui est égal au nombre de fois que l'on peut diviser en deux l'intervalle de départ (l'ensemble du tableau) avant d'obtenir un intervalle contenant une seule valeur. En fait ce nombre est égal à $\log n$ où n est le nombre de valeurs dans le tableau. C'est-à-dire qu'un tableau de 1.000 valeurs va nécessiter environ dix itérations et qu'un tableau de 1.000.000 de valeurs n'en prendra que 20 ! L'algorithme est donc beaucoup plus efficace que l'algorithme de recherche séquentielle sur les collections de grande taille.

L'algorithme suivant met en œuvre l'idée de la recherche dichotomique :

```
i = 0;
j = n - 1;
while (i < j && (t[i] <= x && t[j] >= x)) {
   m = (i + j) / 2;
   if (x <= t[m]) {
      j = m;
   } else {
      i = m + 1;
   }
}
resultat = (t[i] == x);
```

Observons l'exécution de cet algorithme sur deux cas : un cas positif, où l'élément cherché apparaît dans le tableau, et un cas négatif où l'élément cherché n'y apparaît pas. Considérons le tableau suivant :

1	4	5	7	10	15	16	21	25

et cherchons-y la valeur `x` = 7. Le tableau suivant nous donne l'état des variables `i`, `j` et `m` de même que le résultat de l'expression booléenne de contrôle de la répétition. Les points d'observations sont le début du corps de la répétition (PODC), la fin du corps de la répétition (POFC) et le point d'observation final (POF).

	`i`	`j`	`m`	test
PODC itération 1	0	8	4	vrai
POFC itération 1	0	4	4	
PODC itération 2	0	4	2	vrai
POFC itération 2	3	4	2	
PODC itération 3	3	4	3	vrai
POFC itération 3	3	3	3	
POF	3	3		faux

À la première itération, `i` et `j` valent 0 et 8. L'expression de contrôle est vraie ($0 < 8 \wedge 1 \leq 7 \leq 25$. La valeur de `m` calculé est 4 ; comme 7 est inférieur à `t[4]` = 10, on affecte la valeur de `m` à `j`. À la fin de l'itération 1, le nouvel intervalle est [0, 4] et, au début de l'itération 2, l'expression de contrôle est toujours vraie. La nouvelle valeur de `m` est 2 ; comme 7 est supérieur à `t[2]` = 5, on affecte la valeur `m` + 1 à `i`, ce qui nous donne l'intervalle [3, 4]. Au début de l'itération 3, l'expression de contrôle est toujours vraie et la nouvelle valeur de `m` est 3. Comme 7 est inférieur ou égal à `t[3]` = 7, on affecte la valeur 3 à `j`, ce qui donne l'intervalle [3, 3]. L'expression de contrôle devient alors fausse puisque `i` = 3 n'est plus inférieur à `j` = 3. Or, on a bien `t[i]` = 7, la valeur cherchée.

Répétons cette expérience, cette fois-ci en cherchant la valeur 6. On obtient les observations suivantes :

	`i`	`j`	`m`	test
PODC itération 1	0	8	4	vrai
POFC itération 1	0	4	4	
PODC itération 2	0	4	2	vrai
POFC itération 2	3	4	2	
POF	3	3		faux

```
{n > 0}
i = 0 ;
j = n - 1 ;
{inv : 0 ≤ i ≤ j ≤ n − 1}
while (i < j && (t[i] <= x && t[j] >= x)) {
   {a1 : 0 ≤ i < j ≤ n − 1 ∧ t[i] ≤ x ≤ t[j]}
   m = (i + j) / 2 ;
   {a2 : 0 ≤ i ≤ m < j ≤ n − 1 ∧ t[i] ≤ x ≤ t[j]}
   if (x <= t[m]) {
      {a3 : 0 ≤ i ≤ m < j ≤ n − 1 ∧ x ≤ t[m] ∧ t[i] ≤ x ≤ t[j]}
      j = m ;
      {a4 : 0 ≤ i ≤ j ≤ n − 1 ∧ t[i] ≤ x ≤ t[j]}
   } else {
      {a5 : 0 ≤ i ≤ m < j ≤ n − 1 ∧ x > t[m] ∧ t[i] ≤ x ≤ t[j]}
      {a6 : 0 ≤ i < m + 1 ≤ j ≤ n − 1 ∧ x > t[m + 1 - 1] ∧ t[i] ≤ x ≤ t[j]}
      i = m + 1 ;
      {a7 : 0 ≤ i ≤ j ≤ n − 1 ∧ t[i - 1] < x ≤ t[j]}
   }
   {a8 : 0 ≤ i ≤ j ≤ n − 1 ∧ (t[i] ≤ x ≤ t[j] ∨ t[i - 1] < x ≤ t[j])}
   {a9 : 0 ≤ i ≤ j ≤ n − 1 ∧ t[i - 1] < x ≤ t[j]}
}
{a10 : ((i ≥ j ∧ t[i] ≤ x ≤ t[j]) ∨ (x < t[i] ∨ x > t[j])) ∧ 0 ≤ i ≤ j ≤ n − 1}
{a11 : (i = j ∧ t[i] = x = t[j]) ∨ (i ≤ j ∧ (x < t[i] ∨ x > t[j]))}
resultat = (t[i] == x) ;
```

FIG. 5.4 – Algorithme de recherche dichotomique annoté

Après la première itération, la valeur 6 est encadrée entre `t[0]` = 1 et `t[4]` = 10. À la deuxième itération, on calcule `m` = 2 et comme 6 est supérieur à `t[2]` = 5, la valeur de `i` passe à 3. L'évaluation de l'expression de contrôle de la répétition donne alors faux, car la valeur 6 n'est plus encadrée par `t[3]` = 7 et `t[4]` = 10. À la sortie de la répétition, on a donc `i` inférieur à `j`, mais la valeur de `x` en dehors de l'intervalle obtenu, et donc pas dans le tableau.

Malgré sa simplicité apparente, la mise au point de cet algorithme est loin d'être aussi triviale qu'il n'y paraît.[4] Sans un raisonnement très précis, il est difficile de comprendre les subtilités de l'algorithme, et encore plus difficile d'en déterminer précisément les conditions et les actions. En effet, pourquoi fait-on la répétition tant que `i` est strictement inférieur à `j` ? Pourquoi pas inférieur ou égal ? Pourquoi affecter la valeur `m + 1` à `i` dans la prtie sinon de l'alternative ? Pourquoi retourner le résultat du test `t[i] == x` ? Pour comprendre, il faut se pencher sur l'annotation de l'algorithme à la figure 5.4.

Quel est l'objectif de la répétition ? Si la recherche est fructueuse, l'état final devrait être tel que l'intervalle a été réduit à un seul élément du tableau et cet élément doit être égal à `x`. Si la recherche n'est pas fructueuse, l'état après la répétition devrait être tel que la valeur de `x` n'est plus encadrée entre `t[i]` et `t[j]`. En notation assertionnelle, cela se traduit par :

$$\{(\texttt{i} = \texttt{j} \wedge \texttt{t[i]} = \texttt{x}) \vee (\texttt{x} < \texttt{t[i]} \vee \texttt{x} > \texttt{t[j]})\}$$

[4] Meyer rapporte que dans un étude où on a demandé à un groupe de programmeurs expérimentés d'écrire cet algorithme, peu ont réussi à trouver les bonnes conditions pour que leur programme fonctionne du premier coup.

Si on cherche à paramétriser cette assertion par les états successifs des variables dont la valeur change au cours des itérations, `i` et `j`, on constate que d'une part `i` demeure inférieur ou égal à `j`, mais l'autre partie sur laquelle on pourrait jouer, c'est-à-dire la valeur de `x` comparée à celles des bornes de l'intervale ne nous donne rien de plus. En effet, la valeur de `x` reste dans l'intervalle si elle se trouve dans le tableau, mais si elle n'y est pas elle va sortir de l'intervalle. Ce qui paraît invariant ici, c'est que la valeur de `x` est ou n'est pas dans l'intervalle, ce qui n'est pas d'un grand secours !

L'invariant est donc relativement simple : `i` est inférieur ou égal à `j`, et les deux valeurs sont bornées par 0 et $n-1$, où n est la taille du tableau. En notation assertionnelle, on obtient donc $\{inv : 0 \leq \texttt{i} \leq \texttt{j} \leq n-1\}$. Tout le raisonnement sur l'algorithme va donc essentiellement reposer sur les informations tirées des conditiions sur la répétition et l'alternative. À l'entrée de la répétition, l'assertion $a1$ est le reflet de l'expression de contrôle de la répétition, avec en plus les bornes sur les indices. L'assertion $a2$ ajoute la connaissance sur `m` que l'on peut déduire de son calcul. Pour $\texttt{i} < \texttt{j}$, il n'y a que deux cas de figures possibles : soit `j - i`≥ 2 et alors on aura $\texttt{i} < \texttt{m} < \texttt{j}$, soit `j - i`$= 2$ et alors on aura $\texttt{m} = \texttt{i}$. Donc, on en conclut $\texttt{i} \leq \texttt{m} < \texttt{j}$.

L'assertion $a3$ n'ajoute que ce que l'on sait par le fait que l'expression de contrôle de l'alternative est vraie. L'affectation de la valeur de `m` à `j` nous permet de conclure deux choses : $\texttt{i} \leq \texttt{j}$ puisque $\texttt{i} \leq \texttt{m}$, et aussi que $\texttt{x} \leq \texttt{t[j]}$ puisque $\texttt{x} \leq \texttt{t[m]}$, ce que l'on retrouve dans $a4$. Dans la partie sinon de l'alternative, l'assertion $a5$ s'obtient de $a2$ en ajoutant l'information que l'expression de contrôle de l'alternative est fausse. Pour préparer le traitement de l'affectation '`i = m + 1`', on introduit le terme `m + 1` dans $a5$ pour obtenir $a6$, en modifiant les inégalités en conséquence. Après l'affectation, la simple substitution de `m + 1` par `i` nous permet d'obtenir $a7$, si on observe que la condition $\texttt{x} > \texttt{t[i - 1]}$ est la meilleure borne qu'on obtienne sur la valeur de `x` par rapport à la valeur de `t[i - 1]`.

À la sortie de l'alternative, $a8$ est le résultat du ou logique de $a4$ et $a7$, nous donne bien que l'invariant est vrai (et donc vrai en tout PO de la répétition), mais on voit que la valeur de `x` ne peut plus être assurément bornée inférieurement par `t[i]`, mais plutôt par `t[i - 1]` ($a9$). En fait, c'est lorsqu'elle se trouvera entre `t[i - 1]` et `t[i]` qu'on saura qu'elle n'appartient pas au tableau.

À la sortie de la répétition, on a l'invariant vrai et l'information obtenue du fait que l'expression de contrôle de la répétition est fausse, ce qui nous donne l'assertion $a10$. En utilisant la distributivité du et sur le ou et en simplifiant les inégalités, on en tire $a11$. Cette assertion nous permet alors de réfléchir sur la bonne condition à vérifier pour retourner vrai si la valeur de `x` et dans le tableau et faux sinon. Un test sur les valeurs comparées de `i` et `j` n'est pas concluant dans la mesure où on peut avoir $\texttt{i} = \texttt{j}$ que la valeur soit ou non dans le tableau. On utilise donc plutôt le critère `t[i] == x` qui lui sera vrai si la valeur est dans le tableau et faux sinon.

Revenons maintenant brièvement sur le choix de poser '`i = m + 1`' dans la partie sinon de l'alternative plutôt que simplement '`i = m`' ? Pourquoi ce choix ? En fait, il vient de la nécessité d'assurer que la répétition se termine bien à tous les coups. De l'assertion $a2$, on voit que `m` peut être égal à `i`. Si on se trouve dans ce cas, et qu'en plus `x` est supérieur à `t[m]`, affecter simplement la valeur de `m` à `i` laisserait `i` inchangé. On se retrouverait alors dans un cas de répétition qui ne s'arrête pas.

Considérez l'exemple de la recherche de la valeur 6 ci-haut. Lorsque `i` vaut 0 et `j` vaut 4, on calcule $\texttt{m} = 2$, et on a `x` plus grand que $\texttt{t[2]} = 5$. À l'itération suivante on aurait alors $\texttt{i} = 2$ et $\texttt{j} = 4$. On calcule un nouveau $\texttt{m} = 3$, et alors on a `x` inférieur ou égal à $\texttt{m} = 7$, et donc modification de `j` pour valoir 3. À l'itération suivante, on a maintenant $\texttt{i} = 2$, $\texttt{j} = 3$ et `m`

calculé à 2. La valeur de `x` est alors supérieure à `t[2]` = 5, on affecte alors 2 à `i` et on repart à l'itération suivante avec à nouveau `i` = 2, `j` = 3 ! Il est donc crucial de faire l'affectation `i = m + 1` pour assurer la terminaison de la répétition.

5.3 Tableaux comme collections de données à taille variable

Les tableaux servent souvent de moyen de mémorisation de collections de données dont la taille est connue et fixée à l'avance, ou à tout le moins au moment du début de l'exécution des programmes. C'est le cas de l'exemple des températures utilisés en début de chapitre où on peut supposer qu'au plus au début de l'exécution du programme, l'utilisateur sait combien de valeurs de températures il doit fournir au programme pour calculer la moyenne.

Il existe cependant un grande classe de problèmes où la taille de la collection de données n'est pas connue à l'avance. Dans la plupart de ces problèmes, il est même clair que cette taille va varier en cours d'exécution pour augmenter et diminuer au gré de la progression du calcul à faire. Dans ce cas, l'utilisation des tableaux est moins évidente, dans la mesure où la contrainte qui se pose est que la taille d'un tableau est décidée au plus tard lors de la création, et qu'elle ne peut pas être modifiée par la suite.

Une idée devenue maintenant un patron générique à l'ensemble de ces problèmes consiste alors à allouer un tableau d'une taille suffisante pour mémoriser le nombre maximal de données atteint au cours de l'exécution, et de tenir à jour une variable `nombreElements` contenant précisément le nombre de valeurs rangées dans le tableau à tout instant. On parle alors de taille allouée par rapport à la taille effective.

taille allouée : nombre d'éléments du tableau tels qu'il a été créé.

taille effective : nombre de valeurs significatives contenues dans le tableau.

On pose aussi comme hypothèse que si m valeurs sont rangées dans un tableau de taille n, $m \leq n$, ces m valeurs sont toujours rangées dans les éléments d'indice 0 à $m - 1$. La taille allouée devient donc la taille effective maximale admise par le tableau. Par rapport aux algorithmes vus auparavant, tous peuvent être adaptés très facilement au contexte des collections à taille variable en tenant simplement compte de la taille effective plutôt que de la taille allouée.

5.3.1 Collections non-ordonnées

Par définition, une collection à taille variable peut perdre des valeurs et en gagner. Il y aura donc des opérations d'ajout et de retrait de valeurs. Les opérations d'ajout doivent tenir compte de la taille allouée pour ne pas ajouter de valeurs si le tableau est plein ($m = n$). Lors du retrait, il faut aussi s'assurer qu'il y a au moins une valeur dans le tableau ($m > 0$). Considérons quelques algorithmes d'ajout et de retrait en supposant que les éléments ne sont pas ordonnés dans la collection.

Ajout d'un nouvel élément

Rappelons les hypothèses de travail. La collection est représentée par un tableau que l'on nomme `elements`. La taille allouée par ce tableau est définie à la création et accessible soit par une constante `MAX_ELEMENTS`, soit par l'attribut `length` du tableau en Java. Une variable

```
/**
 * @invariant this.taille() >= 0 && this.taille() <= MAX_ELEMENTS
 */
public class          CollectionTableau
{
    /** Le nombre maximal d'éléments dans la collection       */
    public static final       int     MAX_ELEMENTS = 100 ;

    /** Nombre d'élements actuellement dans la collection     */
    protected int             nombreElements ;
    /** Tableau contenant les éléments de la collection       */
    protected Object[]        elements ;

    /**
     * @pre   true
     * @post  this.estVide()
     */
    public            CollectionTableau()
    {
        elements = new Object[MAX_ELEMENTS] ;
        nombreElements = 0 ;
    }
}
```

FIG. 5.5 – Déclaration (partielle) de la classe `CollectionTableau`.

`nombreElements` indique le nombre d'éléments effectivement contenu dans le tableau, éléments rangés aux indices 0 à (`nombreElements` - 1). La figure 5.5 présente la déclaration (partielle) d'une classe `CollectionTableau` inspirée de ces hypothèses.

Pour ajouter un nouvel élément, il suffit de le placer au premier indice libre, lequel se trouve à être précisément l'indice `nombreElements` puisque les indices précédents correspondent tous à des positions prises. Il suffit donc de placer le nouvel élément à l'indice `nombreElements` et à incrémenter `nombreElements` de 1 :

```
{nombreElements < MAX_ELEMENTS}
elements[nombreElements] = nouveau ;
nombreElements = nombreElements + 1 ;
```

Ajout à l'indice k

Il est rarement nécessaire d'ajouter un nouvel élément à une position donnée dans la collection. En effet, dans la plupart des cas, la position n'importe guère. Pourtant, si on utilise un algorithme de recherche séquentielle, on sait qu'on trouve plus rapidement les éléments en début de tableau qu'en fin de tableau. L'ajout à une position donnée peut permettre de positionner un élément aux indices faibles du tableau si on sait par avance qu'on va souvent les accéder par recherche séquentielle.

Pour respecter le fait que les éléments effectifs sont concentrés en début de tableau, l'indice `k` demandé doit être compris entre 0 et (`nombreElements` - 1). Les positions 0 à (`nombreElements` - 1) étant prises, il faut faire de la place à l'indice `k` avant d'y placer la nouvelle valeur. Pour faire de la place, il suffit de déplacer l'élément à la première position libre, c'est-à-dire `nombreElements`. La place à l'indice `k` étant alors libre, il suffit d'y affecter la nouvelle valeur :

```
{nombreElements < MAX_ELEMENTS ∧ k < nombreElements}
elements[nombreElements] = elements[k] ;
elements[k] = nouveau ;
nombreElements = nombreElements + 1 ;
```

Retrait d'un élément à l'indice k

Le retrait d'un élément à l'indice `k` est envisageable dans le cas où l'indice d'un élément à supprimer est connu, ce qui permet de le faire sans avoir à rechercher sa position. L'élément étant retiré, la place à l'indice `k` devient libre, et il convient alors de la combler. Pour ce faire, il faut prendre l'élément à l'indice (`nombreElements` - 1) et le placer à l'indice `k` :

```
{nombreElements > 0 ∧ k < nombreElements}
elements[k] = elements[nombreElements - 1] ;
nombreElements = nombreElements - 1 ;
```

Retrait d'un élément donné

Pour retirer un élément donné, il faut d'abord en rechercher l'indice. Une fois cet indice obtenu, on peut appliquer l'approche du retrait à l'indice `k` précédent :

```
i = 0 ;
while (aRetirer != elements[i] && i < nombreElements) {
    {inv : ∀k ∈ [0, i − 1], elements[k] != aRetirer}
    i = i + 1 ;
}
{aRetirer = elements[i]∨
    (i = nombreElements ∧ (∀k ∈ [0, nombreElements − 1], elements[k] != aRetirer))}
if (i < nombreElements) {
    elements[i] = elements[nombreElements - 1] ;
    nombreElements = nombreElements - 1 ;
}
```

5.3.2 Collections ordonnées

Les collections non-ordonnées ont cet avantage de permettre une gestion simple et efficace des ajouts et retraits. Sauf pour le cas du retrait d'un élément donné, dans lequel il faut d'abord chercher l'élément, les autres opérations se font en deux ou trois affectations, ce qui est peu coûteux. Par contre, un rangement désordonnés des éléments dans la collection forcent à faire la recherche d'éléments par l'approche séquentielle. Nous avons vu que lorsque les collections sont de grande taille, il vaut mieux utiliser un algorithme de recherche plus efficace, et en particulier la recherche dichotomique.

La difficulté est d'obtenir un rangement des éléments ordonnés de manière croissante. Un coût supplémentaire s'ajoute pour maintenir la collection ordonnée lors de l'ajout et du retrait des éléments. En effet, lors de l'insertion, il faut insérer en ordre croissant, et lors du retrait,

il faut préserver l'ordonnancement.

On constate donc que les deux approches gagnent là où l'autre perd. L'approche désordonnée est rapide sur les insertions et les ajouts mais mauvaise sur la recherche. L'approche ordonnée est rapide sur la recherche mais mauvaise lors des ajouts et des retraits. Confronté à un tel compromis, l'informaticien doit recueillir des statistiques sur l'utilisation du programme et décider de favoriser l'une ou l'autre des opérations en fonction de leur fréquence respective. Dans le cas présent, si les ajouts et les retraits sont beaucoup plus nombreux, une collection non-ordonnée est mieux adaptée. Mais si au contraire, le programme fait beaucoup plus de recherches de valeurs que d'ajouts ou de retraits, la collection ordonnée sera plus appropriée.

Insertion en ordre

L'insertion d'un nouvel élément doit respecter l'ordre établi. On doit donc commencer par trouver la position à laquelle le nouvel élément doit s'insérer, puis faire de la place pour insérer l'élément. Trouver la position où faire l'insertion demande de rechercher un indice i tel que le nouvel élément est plus grand ou égal à `elements[`i`]` mais inférieur à `elements[`$i+1$`]`. Ensuite, pour faire de la place, il faut décaler les éléments d'indice $i+1$ à (`nombreElements` - 1) d'une case vers la droite (indices plus élevés), puis affecter la nouvelle valeur à l'indice $i+1$.

```
i = 0 ;
while (nouveau >= elements[i] && i < nombreElements) {
   i = i + 1 ;
}
```
{`nouveau` $<$ `elements[i]` $\vee$ `i` $=$ `nombreElements`}
{inv : `elements[j + 1]` *est libre*}
```
for (j = nombreElements - 1 ; j >= i ; j = j-1) {
   elements[j + 1] = elements[j] ;
}
```
{`elements[j + 1]` *est libre* $\wedge$ `j` $=$ `i` -1}
```
elements[i] = nouveau ;
nombreElements = nombreElements + 1 ;
```

Retrait d'un élément donné

Pour retirer un élément donné de la collection, il faut commencer par trouver l'indice i auquel cet élément se trouve. Une fois cet indice trouvé, il faut remplir le trou laissé par l'élément retiré de telle façon que l'ordre des éléments soit préservé. Il faut donc décaler un à un les éléments d'indices supérieurs à i d'une case vers la gauche (indice moins élevés) :

```
{nombreElements < MAX_ELEMENTS}
i = 0 ;
while (aRetirer != elements[i] && i < nombreElements) {
   i = i + 1 ;
}
{i ≥ nombreElements ∨ aRetirer = elements[i]}
if (i < nombreElements) {
   {aRetirer = elements[i]}
   for (j = i ; j <= nombreElements - 2 ; j++) {
      {elements[j] est libre}
      elements[j] = elements[j + 1] ;
   }
   nombreElements = nombreElements - 1 ;
}
```

Tri par sélection

Comment faire maintenant si une collection a d'abord été crée de manière non-ordonnée et qu'après un certain temps, le programme décide qu'il vaudrait mieux passer à une collection ordonnée ? Il faut ordonner les éléments, opération que les informaticiens appellent traditionnellement le tri des valeurs du tableau. Le tri est une des opérations les plus importantes en traitement de données. Il a donc été étudié intensivement et de nombreux algorithmes de tri ont été proposés. Considérons ici l'un des algorithmes les plus simples : le tri par sélection.[5]

L'algorithme de tri par sélection part d'un constat très simple. Considérons la position 0 dans le tableau trié. Quelle valeur doit se retrouver à cette position ? La valeur la plus petite du tableau, bien sûr. Il suffit donc de trouver la valeur la plus petite du tableau et l'amener en position 0 par échange avec la valeur actuellement en position 0. Maintenant, on peut répéter l'opération avec l'indice 1. La valeur placée en position 0 étant la plus petite du tableau, la valeur qui va aller à la position 1 est la deuxième plus petite. Mais dit autrement, c'est la plus petite valeur du tableau parmi les indices 1 à (`nombreElements` - 1). Si on répète l'opération pour les positions 2, 3 jusqu'à (`nombreElements` - 2), on obtiendra un tableau trié à la fin :

```
for (i = 0 ; i <= nombreElements - 2 ; i++) {
   {∀k < i (∀l ∈ [k, nombreElements] (elements[k] ≤ elements[l]}
   for (j = i + 1 ; j <= nombreElements - 1 ; j++) {
      {∀k < i (∀l ∈ [k, nombreElements] (elements[k] ≤ elements[l]))
         ∧(∀m ∈ [i, j − 1] (elements[i] ≤ elements[m]))}
      if (elements[i] > elements[j]) {
         t = elements[i] ;
         elements[i] = elements[j] ;
         elements[j] = t ;
      }
   }
}
{∀k ∈ [0, nombreElements − 2[ (elements[k] ≤ elements[k + 1])}
```

On peut montrer que le travail que nécessite le tri d'un tableau par cet algorithme est dominé par le nombre de comparaisons des valeurs. Pour chaque indice i du tableau, il faut comparer la valeur à cet indice avec les valeurs d'indice $i + 1$ à $n - 1$, où n est le nombre

[5] D'autres tris seront vus au prochain semestre dans le module «Structures de données et algorithmes».

d'éléments à trier. Il y aura donc $n-1$ comparaisons pour trouver la valeur à l'indice 0, puis $n-2$ pour celle à l'indice 1, et ainsi de suite. Le nombre de comparaisons est donc :

$$\text{nombre de comparaisons} = \sum_{i=1}^{n} i = \frac{n(n-1)}{2} \approx n^2$$

Le tri par sélection n'est pas très performant. Il existe des algorithmes de tri qui nécessitent un travail proportionnel à $n \log n$. Ces algorithmes dépassent cependant le cadre du présent module.

5.3.3 Parcours standardisé des éléments d'une collection

Les collections peuvent être implantées de bien des manières en fonction de la performance des opérations d'accès, d'ajout et de retrait des éléments de la collection. On appelle généralement *structure de données* l'organisation particulière d'une collection d'éléments. Il est souvent utile de pouvoir parcourir tous les éléments d'une collection sans en connaître l'implantation précise. Pour cela, on utilise un intermédiaire appelé objet visiteur, ou plus simplement *visiteur*.

En Java, un objet visiteur est un objet sachant répondre à deux messages : `hasMoreElements` et `nextElement`. Un visiteir sur une collection répondra vrai au message `hasMoreElements` tant qu'il reste encore des éléments à visiter. Au message `nextElement`, il répondra en retournant le prochain élément à visiter. Ce comportement standard est défini par l'interface `java.util.Enumeration` dont la déclaration est :

```
public abstract interface     Enumeration
{
    public abstract boolean hasMoreElements() ;
    public abstract Object  nextElement() ;
}
```

Pour parcourir les éléments d'un tableau, un objet visiteur doit simplement avoir la référence sur l'objet tableau, le nombre d'éléments contenu dans le tableau et l'indice du prochain élément à visiter. On peut ainsi définir une classe `VisiteurTableau` dont le visiteur sera instance :

```
import java.util.Enumeration ;  // interface prédéfinie de la bibliothèque Java

public class            VisiteurTableau
                        implements java.util.Enumeration
{
    /** indice du prochain élément à visiter                    */
    protected int       indice ;
    /** nombre d'éléments à visiter dans le tableau             */
    protected int       tailleMax ;
    /** tableau contenant les éléments à visiter                */
    protected Object[]  leTableau ;

    /**
     * @pre    t != null && max >= 0 && max <= t.length
     */
    public              VisiteurTableau(Object[] t, int max)
    {
        indice = 0 ;
        tailleMax = max ;
        leTableau = t ;
    }
```

```
    public boolean      hasMoreElements()
    {
        return indice < tailleMax  ;
    }

    /**
     * @pre    this.hasMoreElements()
     */
    public Object       nextElement()
    {
        return leTableau[indice++] ;
    }
}
```

Pour parcourir un tableau d'objets `t` contenant 10 éléments (pas un tableau d'éléments des types de base de Java), il suffit alors de créer le visiteur et d'appliquer les méthodes de visite :

```
VisiteurTableau v = new VisiteurTableau(t, 10) ;
while (v.hasMoreElements()) {
    traiter v.nextElement() ;
}
```

Les classes de collection, comme la classe CollectionTableau, peuvent masquer leur représentation en définissant une méthode `enumerate` qui retourne un visiteur de type `Enumeration` :

```
    public Enumeration  enumerate()
    {
        return new VisiteurTableau(elements, nombreElements) ;
    }
```

L'utilisateur peut alors parcourir la collection en utilisant ce visiteur, comme dans l'exemple suivant où on imprime tous les objets de la collection :

```
    CollectionTableau   maCollection ;
    Enumeration         visiteur ;

    ...
    visiteur = maCollection.enumerate() ;
    while (visiteur.hasMoreElements()) {
        System.out.println(visiteur.nextElement().toString()) ;
    }
```

L'utilisation du type `Enumeration` masque même le nom de la classe d'instantiation du visiteur, ce qui découple complètement l'utilisateur des choix d'implantation de la collection.

5.4 Tableaux et notion de réference

Nombre des opérations que nous avons vues dans les sections précédentes peuvent être réalisées par des méthodes. La question qui se pose alors, compte tenu de ce que nous avons vu au chapitre précédent, est de savoir quel mode de passage de paramètres sera adopté pour les tableaux. Rappelons que dans le mode de passage de paramètres par valeur, la valeur du paramètre réel est copiée dans les emplacements mémoire alloués pour le paramètre formel. En passage par référence, le paramètre formel devient un alias du paramètre réel pour la durée de l'exécution de la méthode.

La difficulté à laquelle sont confrontés les concepteurs de langages de programmation est que le passage par valeur sur les tableaux impliquent la copie de tous les éléments du tableau,

opération qui peut s'avérer très coûteuse dès que les tableaux sont de taille un peu grande. Or, si on y réfléchit bien, peu importe la méthode que l'on écrit et qui prend un tableau en paramètre, de deux choses l'une :

- soit la méthode en modifie pas le tableau, et alors passage par valeur ou par référence sont parfaitement équivalents,
- soit la méthode modifie le tableau et alors on souhaite généralement que les modifications soient faites directement sur le tableau passé en paramètre.

Dans le second cas, pensons par exemple aux opérations de retrait et d'ajout de valeurs, ou encore au tri. Dans tous ces cas, on veut souvent que le tableau soit modifié de façon définitive par l'opération appliquée. Les cas où on ne souhaite pas modifier le tableau passé en paramètre sont rares ; le coût de la copie étant élevé, la plupart des langages de programmation réalisent donc le passage de tableau par référence pour les tableaux. C'est le cas de Java, mais pour une raison moins directe qui sera discutée plus loin (§5.5). Une variable de type tableau peut donc être considérée comme détenant une référence sur le tableau.

Par ailleurs, nous avons vu que les éléments d'un tableau correspondent à des emplacements mémoire. Pour cette raison, la notation d'indiçage est utilisable dans le contexte gauche d'une affectation, de la même manière qu'une variable. La question que l'on peut se poser est de savoir si un élément individuel d'un tableau peut être utilisé en passage de paramètre par référence. Dans les langages de programmation qui offrent le passage par référence, c'est effectivement le cas. Par exemple, si une procédure attend un paramètre par référence de type entier, il sera possible de passer un élément individuel d'un tableau d'entier en utilisant son indice.

Dans les langages, comme Java, où il n'existe pas de passage par référence sauf pour les tableaux (et les objets), on peut simuler un passage par référence à un élément d'un tableau en passant indépendamment le tableau et l'indice de l'élément à modifier. Ainsi, la méthode suivante :

```
public static void    modifieElement(
    int[] tableau,
    int indice,
    int valeur)
{
   tableau[indice] = valeur ;
}
```

car la conjonction du passage par référence du tableau et de l'utilisation de l'indice à l'intérieur de la méthode donne l'équivalent d'un passage par référence de l'élément de tableau. Cette simulation d'un passage par référence nous permet d'écrire une méthode d'échange des valeurs de deux éléments d'un tableau en passant le tableau indépendamment des deux indices :

```
public static void    echange(int[] tableau, int i, int j)
{
   int temp ;

   temp = tableau[i] ; tableau[i] = tableau[j] ; tableau[j] = temp ;
}
```

On peut donc faire par cette simulation ce que nous avions réussi à faire aussi en passant par un objet.

5.5 Élément de savoir-faire : tableaux en Java

5.5.1 Tableaux versus objets

En Java, un tableau est considéré comme un objet. Une classe définit les propriétés les plus importantes des tableaux : `Array`. Les tableaux sont donc manipulés par référence, et plus précisément à travers leur identificateur d'objet. Une autre conséquence de l'analogie tabelaux/objets vient du fait que Java applique la même politique de déclaration et de création aux tableaux qu'aux objets. En particulier, une variable de type tableau sera déclarée comme n'inporte quelle autre variable et pourra contenir un objet tableau. De plus, un tableau est créé dynamiquement (à l'exécution) à l'aide de l'opérateur `new`. La gestion de la mémoire occupée par un tableau est donc la même que celle des objets, c'est-à-dire faite de manière complètement automatique.

Cependant, Java prévoit des éléments de syntaxe facilitant l'utilisation des tableaux, et en particulier pour la déclaration, la création et l'accès aux tableaux. Ces éléments de syntaxe font des tableaux des objets tout de même un peu particulier dans la langage.

La déclaration d'une variable de type tableau en Java détermine le nom de la variable, le type des éléments du tableau et le nombre de dimensions. Elle ne fait pas intervenir le nombre d'éléments dans le tableau qui ne sera déterminé qu'au moment de la création du tableau. Voici quelques exemples de déclarations de tableaux :

```
int[] tableauEntiers ;
int tableauEntiers[] ;
byte[][] matriceOctets ;
Rectangle tableauRectangles[] ;
```

L'ajout de crochets au nom du type ou au nom de la variable est indifférent à Java. Généralement, on préfère la première option car elle fait apparaître plus clairement le type tableau. Le nombre de paires de crochets ouvrant/fermant définit le nombre de dimensions du tableau. Un tableau peut aussi contenir des objets, et alors on utilise le nom de la classe des objets comme nom de type, comme il se doit.

La création d'un tableau fait intervenir le type des éléments ainsi que le nombre de dimensions du tableau et détermine le nombre d'éléments. En utilisant l'opérateur `new`, des exemples de création donne :

```
tableauEntiers = new int[10] ;
tableauRectangles = new Rectangles[100] ;
```

Le nombre d'éléments contenu dans le tableau à créer est donné entre crochet suivant le nom du type. Remarquez que dans le cas des tableaux d'objets, il s'agit de créer le tableau et non les objets eux-mêmes ; le nom de la classe (ici `Rectangle`) ne fait donc pas référence aux méthodes d'initialisation de cette classe. On peut aussi créer un tableau à partir de valeurs initiales à l'aide d'une forme dite d'initialisation statique. Deux cas sont alors possibles : une forme dite initialiseur qui n'est utilisable qu'en conjonction avec la déclaration de la variable tableau :

```
int[] tableauEntiers = { 1, 2, 4, 8, 16, 32, 64 } ;
```

et une forme dite en création dynamique avec initialisation :

```
tableauEntiers = new int[] { 1, 2, 4, 8, 16, 32, 64 } ;
```

Par ailleurs, tout tableau connait sa taille (nombre d'éléments) par son attribut `length`, comme par exemple dans `tableauEntiers.length`.

L'accès aux éléments se fait par l'opération d'indiçage similaire à celle vue en cours :

`tableauEntiers[i]`, `matriceOctets[i][j]`.

L'indiçage s'utilise à la fois à gauche et à droite d'une affectation, dans le premier cas pour désigner l'emplacement dans le tableau où une valeur doit être rangée et dans le second cas pour désigner le contenu de l'élément de tableau. Lors d'un indiçage, Java vérifie toujours la conformité de l'indice utilisé avec les bornes d'indiçage définies par la création du tableau. Pour un tableau `t`, l'indice doit toujours se situer entre les bornes 0 et '`t.length - 1`'. Si ce n'est pas le cas, une erreur de débordement de tableau est déclarée.

5.5.2 Exemple d'utilisation des tableaux dans les objets

Les tableaux servent, comme nous avons vu, deux grandes utilisations : la mémorisation d'une série de données que l'on veut traitées comme un tout (voir l'exemple des températures), ou encore pour mémoriser des collections de données dont la taille peut varier en cours d'exécution. Nous proposons un exemple du premier cas : une classe `Polynome` utilisant un tableau pour mémoriser comme un tout l'ensemble des coefficients du polynôme représenté.

La classe `Polynome`

Nous avons vu au chapitre 3 un exemple portant sur la définition d'une classe `Poly2` pour représenter des polynômes de degré 2 par des objets. La classe `Polynome` vise à étendre cette classe `Poly2` pour représenter des polynômes de degré supérieur à 2. Pour représenter un polynôme de degré supérieur à 2, il faut pouvoir conserver les coefficients des termes des différents degrés. Pour cela, plusieurs choix sont bien sûrs possibles. Même si l'on décide ici d'utiliser un tableau de réels pour conserver ces coefficients, plusieurs solutions restent possibles. Nous choisissons la plus simple : un tableau de taille maximale fixée contient à chaque indice i le coefficient du terme x^i de degré i dans le polynôme. À l'indice 0 de ce tableau de coefficients, on trouve le coefficient constant, à l'indice 1, le coefficient du terme x, et ainsi de suite.

Ce choix nous amène à adopter la représentation indiquée à la figure 5.6 : le tableau de coefficients `coefficients` d'éléments de type `double` sera crée à une taille qui peut soit être la taille par défaut déclarée par la constante `N_COEF_DEFAUT` ou encore à une taille choisie lors de la création de l'objet et accessible par l'attribut `length` du tableau `coefficients`. Que l'un ou l'autre des solutions soit choisie, elle limite le nombre de termes et donc de coefficients dans le polynômes à une valeur finie. La valeur `coefficients.length - 1` donne l'indice correspondant au terme du plus haut degré dans le polynôme.

Par exemple, le polynôme $3x^4 + x^2 - 2x$ peut être créé avec une taille bornée au coefficient de degré 4, ce qui nous donnera une représentation où `coefficients.length` est égal à 4, et où les valeurs des coefficients aux indices 0 à 4 sont respectivement 0, 2.0, 1.0, 0, 3.0. S'il était créé sans indication de degré maximal, le tableau de coefficients serait crée de taille `N_COEF_DEFAUT` et tous les coefficients de degré supérieur à 4 serait initialisés à 0.

```
/**
 * <STRONG> Description </STRONG>
 *
 *      Représentation de polynômes a(n) * x^n + a(n-1) * x^(n-1) + ... +
 *      a(1) * x + a(0).  On peut additionner, soustraire, multiplier, dériver
 *      les polynômes, ainsi que les mulitplier par un scalaire.
 *
 * <STRONG> Principes et fonctionnalités </STRONG>
 *
 *      Un objet de type Polynome est défini par les coefficients et les degrés
 *      de ses termes.  On peut additionner, soustraire et dériver des
 *      polynômes.
 *
 * <STRONG> Date </STRONG>
 *
 *      10 octobre 2001
 *
 * <STRONG> Liste des mises à jour </STRONG>
 *
 *<table BORDER=2 COLS=2 WIDTH="100%" NOSAVE ><tr>
 *<td> néant           </td><td>                                          </td>
 *</tr></table>
 *
 * <STRONG> Invariant structurel </STRONG>
 *
 * @invariant   coefficients != null
 *
 * @author      Jacques Malenfant
 */
public class    Polynome
{
    /** Nombre de coefficients par défaut                               */
    static private final int            N_COEF_DEFAUT = 100 ;
    /** Polynôme zéro                                                   */
    public static final Polynome        ZERO = new Polynome() ;

    /** tableau des coefficients du polynôme                            */
    protected double[]          coefficients ;

    /** initialisation de Polynome 0 à N_COEF_DEFAUT coefs              */
    public              Polynome() ;
    /** initialisation de Polynome 0 à nombre de coefs                  */
    public              Polynome(int degre) ;
    /** initialisation de Polynome avec coefs initiaux                  */
    public              Polynome(double[] coefs) ;
    /** initialisation de Polynome par copie                            */
    public              Polynome(Polynome autrePoly) ;

    /** égalité des polynômes, i.e. égalité de leurs coefficients       */
    public boolean      equals(Polynome autrePoly) ;
    /** addition de polynômes                                           */
    public Polynome     additionne(Polynome autrePoly) ;
    /** soustraction de polynômes                                       */
    public Polynome     soustrait(Polynome autrePoly) ;
    /** retourne la dérivée du polynôme receveur                        */
    public Polynome     derive() ;
    /** retourne l'inverse additif du receveur                          */
    public Polynome     inverseAdditif() ;
    /** retourne une chaîne présentant le polynôme                      */
    public String       toString() ;
}
// ------------------------------------------------------------ Classe Polynome
```

FIG. 5.6 – Classe `Polynome`

La figure 5.6 nous indique également quels sont les méthodes de cette classe. Quatre méthodes d'initialisation sont proposées. La première méthode d'initialisation est sans argument et elle crée un objet représentant le polynôme 0 représenté à l'aide d'un tableau de coefficients de taille `N_COEF_DEFAUT` dont tous les éléments sont initialisés à 0 :

```
/**
 * initialisation du polynôme 0 à N_COEF_DEFAUT coefs
 *
 *    Initialise le nouveau polynôme à 0 avec N_COEF_DEFAUT coefficients.
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     true
 * @post    this.equals(this.inverseAdditif())
 */
public              Polynome()
{
    /*
     * Crée le tableau de coefficients et l'initialise à 0 partout
     * pour former le polynôme 0.
     */
    int     i ;

    coefficients = new double[N_COEF_DEFAUT] ;
    for ( i = 0 ; i < N_COEF_DEFAUT ; i++ ) {
        coefficients[i] = 0.0 ;
    }                       // for (i = 0 ; i < N_COEF_DEFAUT ; i++)
}
```

La seconde méthode d'initialisation crée elle aussi un objet représentant le polynôme 0 mais dont le degré maximal possible après coup sera limité à une valeur donnée en argument :

```
/**
 * initialisation du polynôme 0 à un "degré" donné
 *
 *    Initialise le nouveau polynôme à 0 avec nombre de coefficients
 *    donné par le degré passé en paramètre plus 1.
 *
 * @param   "degré" du nouveau polynôme
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     degre >= 0
 * @post    this.equals(this.inverseAdditif())
 */
public              Polynome(
    int degre
    )
{
    /*
     * Crée le tableau de coefficients et l'initialise à 0 partout
     * pour former le polynôme 0.
     */
    int     i ;

    coefficients = new double[degre + 1] ;
    for ( i = 0 ; i < coefficients.length ; i++ ) {
        coefficients[i] = 0.0 ;
    }                                       // for (i = 0 ; ...)
}
```

La troisième méthode d'initialisation crée un objet représentant un polynôme dont les coefficients sont donnés par un tableau passé en argument et dont le degré maximal possible

est limité à la taille du tableau donné en argument. Les valeurs de coefficients sont rangés dans le tableau argument selon le même principe de rangement que dans le tableau de coefficients servant à la représentation du polynôme, c'est-à-dire le coefficient de degré i à l'indice i. Cela nous donne donc :

```
/**
 * initialisation d'un polynôme avec coefficients initiaux
 *
 *    Initialise le polynôme avec les coefficients initiaux donnés
 *    par l'argument.  L'argument est un tableau qui doit respecter
 *    la règle que le coefficient de degré i se trouve à l'indice i.
 *
 * @param   tableau des coefficients initiaux
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     coefs != null
 * @post    true
 */
public              Polynome(
    double[] coefs
    )
{
    /*
     * Pour éviter de faire référence à un tableau connu aussi de
     * l'extérieur, crée le tableau des coefficients et initialise
     * par copie de l'argument.
     */
    int     i ;

    coefficients = new double[coefs.length] ;
    for ( i = 0 ; i < coefs.length ; i++ ) {
        coefficients[i] = coefs[i] ;
    }                                       // for (i = 0 ; ... )
}
```

Enfin, la quatrième méthode d'initialisation crée un polynôme par copie d'un polynôme existant :

```
/**
 * initialisation de polynôme par copie d'un polynôme existant
 *
 * @param   autre objet polynôme à dupliquer
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     autrePoly != null
 * @post    this.equals(autrePoly)
 */
public              Polynome(
    Polynome autrePoly
    )
{
    /*
     * Copie les coefficients d'autreObj dans le nouveau
     * polynôme.
     */
    int     i ;

    coefficients = new double[autrePoly.coefficients.length] ;
    for ( i = 0 ; i < coefficients.length ; i++ ) {
        coefficients[i] = autrePoly.coefficients[i] ;
    }                                       // for (i = 0 ; ... )
}
```

Les méthodes `equals`, `additionne` et `soustrait` prennent un polynôme en paramètre et respectivement le compare, l'additionne ou le soustrait du polynôme recevant le message (et donc sur lequel s'exécute la méthode). Ces trois opérations sont similaires, puisqu'il s'agit dans les trois cas de parcourir les tableaux de coefficients des deux polynômes pour faire les comparaisons, additions ou soustractions terme à terme entre les deux polynômes. Dans les deux cas, une condition posée en précondition pour que l'opération se fasse est que la taille des deux polynômes soient la même. Cette condition pourrait être levée en changeant l'implantation des méthodes (voir les exercices de fin de chapitre). On obtient donc :

```
/**
 * égalité des polynômes, c'est-à-dire égalité de leurs coefficients
 *
 *    si le degré des deux polynômes n'est pas le même, ils peuvent être
 *    égaux si tous les coefficients supplémentaires de polynôme de degré
 *    le plus élevé sont égaux à 0.
 *
 * @param   autre polynôme à comparer au receveur
 * @return  vrai si les deux polynômes sont égaux
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     autrePoly != null
 * @pre     coefficients.length == autrePoly.coefficients.length
 * @post    true
 */
public boolean      equals(
    Polynome autrePoly
    )
{
    boolean         ret ;
    int             i ;

    ret = true ;
    for ( i = 0 ; i < coefficients.length ; i++ ) {
        ret = ret && (coefficients[i] == autrePoly.coefficients[i]) ;
    }                                           // for (i = 0 ; ... )
    return ret ;
}

/**
 * addition de polynômes
 *
 *    Retourne le polynôme obtenu en additionnant terme à terme
 *    les coefficients du polynôme receveur à ceux du polynôme
 *    argument.
 *
 * @param   second argument de l'addition
 * @return  nouveau polynôme résultant de l'addition
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     autrePoly != null
 * @pre     coefficients.length == autrePoly.coefficients.length
 * @post    return.soustrait(this).equals(autrePoly)
 */
public Polynome     additionne(
    Polynome autrePoly
    )
{
    Polynome        ret ;
    int             i ;

    ret = new Polynome(coefficients.length - 1) ;

    for ( i = 0 ; i < coefficients.length ; i++ ) {
        ret.coefficients[i] = coefficients[i]
```

```
                                + autrePoly.coefficients[i] ;
    }                                           // for (i = 0 ; ... )

    return ret ;
}

/**
 * soustraction de polynômes
 *
 *    Retourne le polynôme obtenu en additionnant terme à terme
 *    les coefficients du polynôme receveur à ceux du polynôme
 *    argument.
 *
 * @param   second argument de l'addition
 * @return  nouveau polynôme résultant
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     autrePoly != null
 * @pre     coefficients.length == autrePoly.coefficients.length
 * @post    return.additionne(autrePoly).equals(this)
 */
public Polynome     soustrait(
    Polynome autrePoly
    )
{
    Polynome        ret ;
    int             i ;

    ret = new Polynome(coefficients.length - 1) ;

    for ( i = 0 ; i < coefficients.length ; i++ ) {
        ret.coefficients[i] = coefficients[i]
                            - autrePoly.coefficients[i] ;
    }                                       // for (i = 0 ; ... )

    return ret ;
}
```

La méthode dérivation retourne en résultat le polynôme obtenu en dérivant le polynôme receveur du message. La règle de dérivation des monômes est bien connue :

$$\frac{d\,ax^n}{dx} = nax^{n-1}$$

La dérivation d'un polynôme consiste à dériver chacun de ses monômes. La méthode `derive` traite donc les termes du polynôme en commençant à l'indice 1 (sauf pour un polynôme constant, c'est-à-dire de degré 0, où le résultat est nécessairement le polynôme 0) :

```
/**
 * retourne la dérivée du polynôme receveur
 *
 * @return  nouveau polynôme, la dérivée du receveur
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     coefficients.length >= 1
 * @post    coefficients.length > 1 implies
 *                  return.coefficients.length == coefficients.length - 1
 * @post    coefficients.length == 1 implies return.equals(ZERO)
 */
public Polynome     derive()
{
    /*
     *   Chaque terme est dérivé successivement selon la règle
```

```
     *   de dérivation des termes :
     *              dérivée(a*x**n) == n*a*x**(n-1)
     *   sauf pour le terme de degré 0 dont la dérivée est nulle.
     *   On commence donc la boucle à l'indice 1.
     */
    Polynome        ret ;
    int             i ;

    if ( coefficients.length == 1 ) {
        ret = new Polynome(0) ;     // dérivée d'une constante
    } else {
        ret = new Polynome(coefficients.length - 1) ;
        for ( i = 1 ; i < coefficients.length ; i++ ) {
            ret.coefficients[i - 1] = coefficients[i] * i ;
        }                           // for (i = 1 ; ... )
    }                               //  if ( coefficients.length == 1 )
    return ret ;
}
```

La méthode `toString` se borne à retourner une chaîne de caractères représentant le polynôme uniquement par ses coefficients dans l'ordre décroissant des degrés, l'ordre donnant le degré correspondant au coefficient. Par exemple, le polynôme $3x^4 + x^2 - 2x$ sera représenté par la chaîne `"[3.0, 0.0, 1.0, 2.0, 0.0]"`.

```
/**
 * retourne une chaîne présentant le polynôme
 *
 * @return String   : chaîne présentant le polynôme
 */
public String       toString()
{
    String  ret ;
    int     i ;

    for ( i = coefficients.length - 1 ; i >= 0 && coefficients[i] == 0.0 ;
          i-- ) {
         ;
    }                       // for (i = coefficients.length - 1 ; ... )
    ret = "[" ;
    for (  ; i > 0 ; i-- ) {
        ret = ret + coefficients[i] + ", " ;
    }                       // for ( ; i >= 0 ; i--)
    ret = ret + coefficients[0] + "]" ;

    return ret ;
}
```

5.6 Exercices

5.6.1. Un palindrôme est une séquence de lettres qui donnent le même mot ou la même phrase, qu'il soit lu de gauche à droite ou de droite à gauche. Écrire une méthode `verifiePalindrome` qui reçoit un tableau de lettres en paramètre et qui retourne vrai s'il forme un palindrôme et faux sinon.

5.6.2. La fréquence d'une valeur dans une collection de données est le nombre de fois que cette valeur apparaît dans la collection. Écrire une méthode `calculeFrequence` qui reçoit en paramètre une lettre ainsi qu'un tableau de lettres, et qui retourne comme résultat la fréquence de la lettre dans le tableau.

5.6.3. La *médiane* d'une collection de nombres est la valeur m telle que la moitié des valeurs de la collection sont inférieures à m et l'autre moitié sont supérieures, si le nombre de valeurs dans la collection est impair. Si le nombre de valeurs est pair, la médiane est la moyenne arithmétique des deux éléments m_1 et m_2 tels que la moitié des éléments restants sont supérieurs ou égaux à m_1 et m_2. Écrire une méthode `calculeMediane` qui prend un tableau de nombres en paramètre et qui retourne la médiane des valeurs contenues dans le tableau.

5.6.4. Le *mode* d'une collection de nombres est la valeur m qui se répète le plus souvent dans la collection. Écrire une méthode `calculeMode` qui prend un tableau de nombres en paramètre et retourne son mode.

5.6.5. Écrire une méthode `multiplieMatriceVecteur` qui prend un tableau à deux dimensions ainsi qu'un tableau à une dimension et qui retourne le résultat de la multiplication de la matrice représentée par le premier par le vecteur représenté par le second.

5.6.6. Écrire une méthode `multiplieMatrices` qui prend deux tableaux à deux dimensions et qui retourne le résultat de la multiplication matricielle entre les deux.

5.6.7. Étendre la classe `Polynome` en définissant une méthode `multiplicationScalaire` qui prend un réel en argument et multiplie le polynôme receveur par ce réel.

5.6.8. Reprendre les méthodes `equals`, `addition` et `soustraction` de la classe `Polynome` et les modifier pour permettre la comparaison, l'addition et la soustraction de polynômes de degrés différents.

5.6.9. Étendre à nouveau la classe `Polynome` en définissant une méthode `multiplication` qui prend en argument un autre polynôme et retourne le poynôme résultant de la multiplication du receveur avec l'argument de la méthode.

Chapitre 6

Développement systématique de classes

Dans les chapitres précédents, nous avons successivement introduit les notions d'objet et de classe, puis les notions de programmation impérative sous la forme d'un ensemble d'énoncés permettant d'écrire de programmes. Dans le présent chapitre, nous intégrons ces différentes connaissances dans un processus systématique qui prend une classe telle que définie après une analyse du problème vue chapitre 3 et qui aura comme résultat la classe complète et exécutable au sein d'un programme. Nous insistons sur le besoin d'une méthodologie de programmation qui, à la fois, assure la qualité du logiciel produit et facilite le travail en équipe.

6.1 Méthodologie de programmation

Nous avons vu au chapitre 3 comment définir les classes d'une application à partir d'une description narrative d'un problème. Les classes sont identifiées à partir des entités du problème le plus souvent désignées par des substantifs dans le texte. Les opérations que ces entités doivent réaliser sont le plus souvent désignées par les verbes utilisés pour décrire les actions de ces entités. Cette approche peut être systématisée, et l'une des méthodes pour le faire est celle des fiches proposée par Budd [Bud98], ce dont nous n'allons cependant pas traiter ici.

À l'issue de cette analyse du problème, les classes doivent être développées pour obtenir l'application. Nous avons vu également au chapitre 3 quelques règles pour guider le programmeur dans ce processus. Deux difficultés majeures apparaissent alors : comment arriver à une définition de chacune des classes qui soit cohérente, correcte et complète ? et comment diviser le travail entre plusieurs programmeurs de telle façon que les classes développées indépendamment puissent être assemblés sans surprise pour donner une application finale qui fonctionne ?

La programmation reste aujourd'hui encore une activité en grande partie de nature artisanale, or les objectifs précédents ne peuvent être atteints que si les programmeurs luttent contre cette tendance et pour cela s'appuient sur une méthodologie la plus précise et la plus rigoureuse possible. Une méthodologie de programmation se définit comme une succession de tâches permettant de prendre une spécification relativement informelle du programme à développer et de produire comme résultat un programme complet qui fonctionne et qui résout le problème initialement soumis. Des telles méthodologies systématiques sont de plus en plus proposées pour aider le programmeur à arriver à ses fins. Les deux principaux avantages à

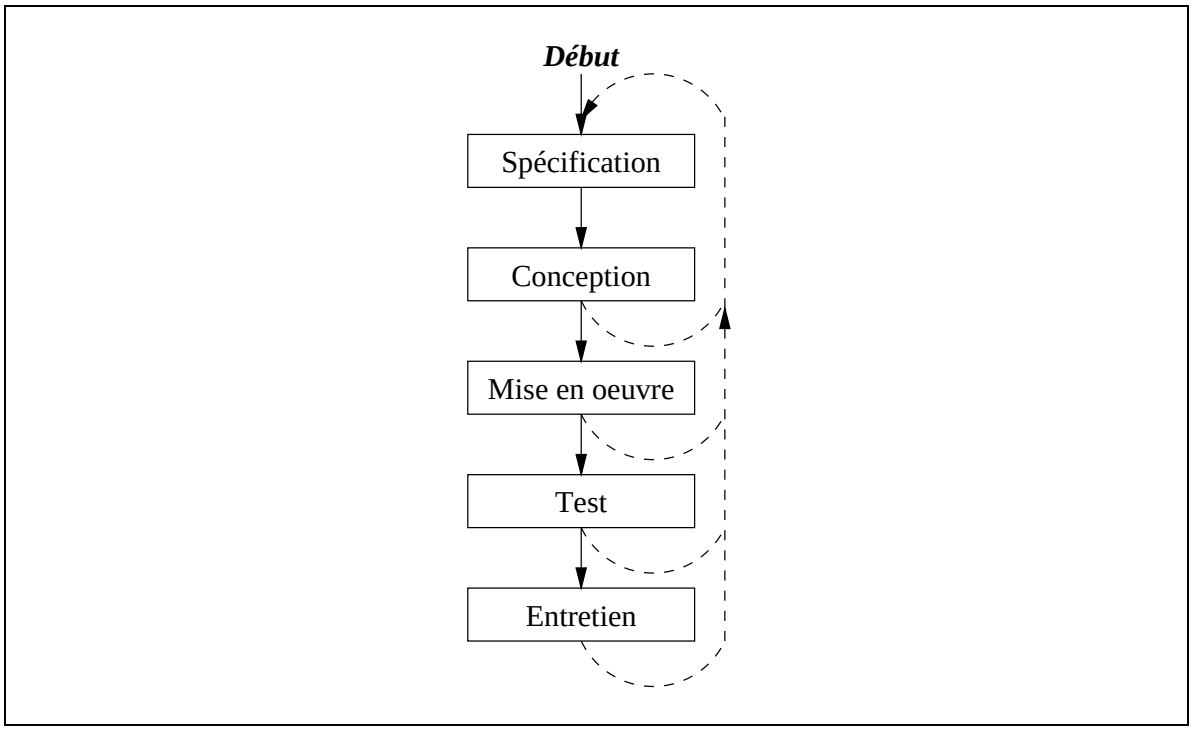


FIG. 6.1 – Cycle traditionnel de développement du logiciel.

travailler selon une méthodologie précise sont la qualité du logiciel produit et l'ouverture au travail en équipe.

Travailler en suivant une méthodologie assure une certaine confiance dans le fait que tous les aspects importants du programme ont bel et bien été considérés, que rien n'a été oublié et que les problèmes soulevés ont été résolus. Tout cela contribue à augmenter la confiance dans la qualité du programme obtenu. Quant au deuxième avantage, lié au travail en équipe, l'adoption d'une méthodologie commune facilite les échanges entre les programmeurs, de même qu'elle facilite la reprise du travail d'un programmeur par un autre lorsqu'il faut corriger ou modifier le programme. Elle facilite la planification du travail et l'imbrication des développements faits par plusieurs personnes, ce qui relève de la *gestion de projets*. Une méthodologie bien suivie et supportée par une documentation de projet adéquate permet également de retracer plus facilement l'origine des erreurs commises et donc de retracer toutes leurs implications dans le programme. L'importance de la qualité des logiciels et les coûts du développement puis de la maintenance de ces derniers sont tels que ces aspects sont des enjeux techniques et économiques cruciaux aujourd'hui.

C'est la discipline appelée *génie logiciel* qui s'intéresse à l'ensemble des méthodes systématiques, structurées et éventuellement formelles (fondées mathématiquement) permettant le développement économiquement viable de logiciels corrects et sûrs. Dans le présent chapitre, nous vous proposons une première méthodologie systématique. Certes cette méthodologie n'est pas de niveau industriel, et elle est loin d'être complète du point de vue génie logiciel, qui est de plus en plus vu comme un métier à part entière. La méthodologie que nous vous proposons est cependant relativement facile à aborder pour des programmeurs novices et elle s'appuie sur les notions introduites dans les chapitres précédents.

De manière générale, le développement d'un logiciel suit ce que le génie logiciel appelle un *cycle de développement*. Le cycle de développement identifie conceptuellement les grandes

étapes qui sont nécessaires pour passer d'un problème à un produit. Le cycle traditionnel de développement, illustré à la figure 6.1, comporte cinq grandes étapes :

Spécification : À cette étape, l'objectif est d'expliciter de manière complète ce que le programme doit faire. Les questions que l'on se pose typiquement à cette étape sont : quelles sont les entrées du programme ? comment seront-elles saisies ? quelles seront les sorties ? sous quelle forme doivent-elles être rendus ? comment le programme sera-t-il utilisé ? Le résultat de la phase de spécification est un document décrivant de manière informelle ou formelle[1] ce que le programme devra faire.

Conception : Autant la spécification vise à décrire ce que le programme doit faire, autant la conception vise à indiquer de manière schématique comment le programme va le faire. Il s'agit donc d'identifier les principaux composants du logiciel et de décrire leurs interrelations. Les questions typiques qui se posent alors sont : quelles sont les composants principaux ? quels composants communiquent avec quels autres ? quelle information est nécessaire à chaque composant ? comment organiser les composants ? Le résultat de cette phase de conception est un document décrivant de manière précise la structure du programme, les données sur lesquelles il doit travailler, les principaux algorithmes qui doivent être utilisés, etc. Le tout est exprimé de manière au moins structurée et parfois formelle. Dans le monde industriel, la conception utilise de plus en plus des notations structurées du type UML, dont nous avons utilisé les diagrammes de classes et d'objets dans les chapitres précédents.

Mise en œuvre : C'est l'étape de programmation proprement dite. Il s'agit de l'expression de la solution dans un langage de programmation donné, et d'une façon qui reflète la compréhension du problème (explicitation des intentions). Le résultat de cette étape est donc constitué de l'ensemble des fichiers de code source constituant le programme.

Tests : La phase de test consiste à exécuter le programme sur un échantillon comprenant une grande variété d'entrées possibles afin soit de pouvoir dire avec suffisamment de confiance que le logiciel est correct, ou de détecter les erreurs laissées par les programmeurs à l'étape de mise en œuvre et les corriger. Le résultat de cette étape est la qualification du logiciel pour sa livraison au(x) client(s). On distingue généralement deux types de tests : les tests unitaires faits sur les composants (classe, objets) individuellement, et les tests d'intégration qui ont pour but de vérifier les bonnes «connexions» entre les composants.

Entretien : Cette phase se développe après la livraison au(x) premier(s) client(s) et elle vise à corriger les erreurs détectées sur le terrain et à effectuer des modifications à la demande du client (dont les intentions sont rarement parfaitement claires au départ). Des études empiriques ont montré que cette phase est la plus coûteuse dans les projets de développement de logiciels, accaparant généralement environ 60% des ressources totales allouées au projet.[2] En effet, corriger des erreurs demande du temps car il faut souvent recomprendre le programme plusieurs mois après sa livraison. Les modifications faites après la conception, parce qu'elles n'ont pu être prises en compte dans l'architecture du programme, demandent généralement beaucoup d'efforts de redéveloppement.

Contrairement à ce que l'on peut croire à première vue, ce cycle de développement se réalise rarement sans ratés et à coup. La vision idyllique de la cascade où entre un problème et d'où sort une solution en suivant les étapes les unes après les autres n'est malheureusment pas de ce monde. Au contraire, il arrive très souvent que la conception amène des questions sur les

[1] Une spécification formelle est exprimée dans un langage mathématiquement rigoureux, alors qu'une spécification informelle peut prendre la forme de description en langue naturelle (français, anglais, etc.), de diagrammes ou de descriptions le plus souvent rigoureusement structurées (par eexemple en UML.

[2] En augmentation constante, puisqu'on a observé jusqu'à 80% sur certains projets récents.

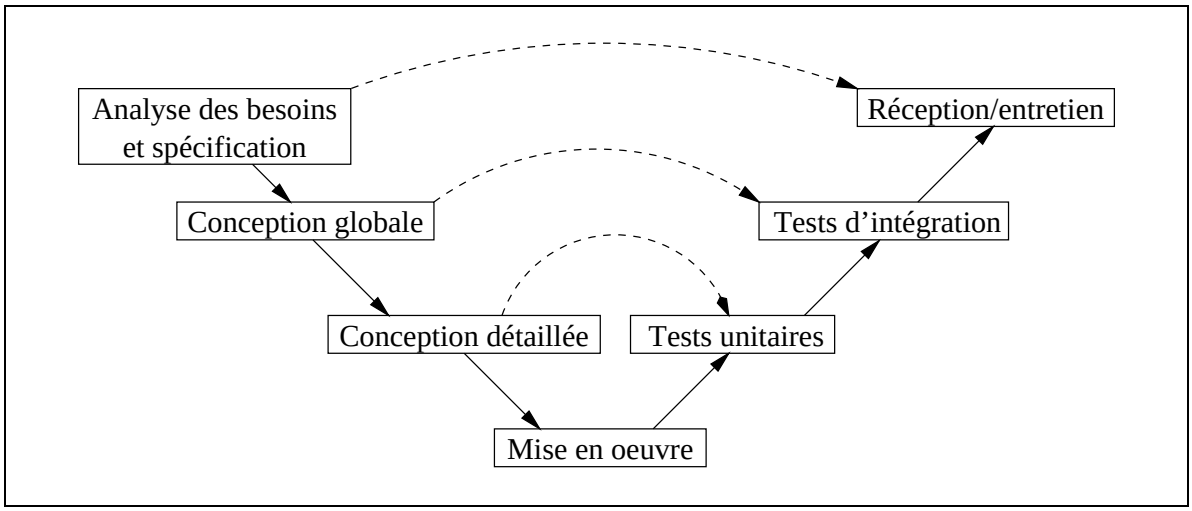


FIG. 6.2 – Cycle traditionnel de développement du logiciel.

spécifications qui induisent la reprise au moins partielle de la phase de spécification. De même, il arrive souvent que la phase de mise en œuvre fasse apparaître des erreurs dans la conception dont la correction exige de reprendre au moins en partie la phase de conception. Enfin, les tests et l'entretien, par leur nature même, implique un retour sur la mise en œuvre, voire même avant lorsque les erreurs détectées font remonter jusqu'à des problèmes de spécifications. On a donc bien affaire à un cyle, indiqué par les flèches en retour dans la figure 6.1.

Considérant l'ensemble du développement, la mise en oeuvre apparaît comme un point charnière. D'une part, les efforts de spécifications et de conception y mène inéluctablement et les phases de tests et d'entretien en découlent nécessairement. Mais en plus, on peut voir apparaître un parallélisme possible entre les étapes du cycle. En effet, les spécifications faites, il devient immédiatement possible de préparer les tests d'utilisation qui précèdent la livraison d'un produit au client. En effet, les spécifications cherchant à exprimer de manière correcte et complète les fonctions du logiciel, elles suffisent à définir les tests d'utilisation. De même, dès la conception globale réalisée, les composants du logiciel sont identifiés et leurs fonctions sont spécifiées à leur tour. Ceci fait, on peut déjà définir les tests d'intégration entre ces composants. Enfin, dès que la conception détaillée de chacun de ces composants est faite, il est possible de produire les tests qui permettront de les vérifier individuellement.

Le lien et le parallélisme entre ces différentes phases sont rendus visuellement par le cycle de développement dit en V qui est présenté à la figure 6.2. C'est à partir de ce cycle en V que nous vous proposons de travailler. La méthode d'analyse introduite au chapitre 3 permet d'identifier les principales classes d'un programme, c'est-à-dire qu'elle identifie ces classes et leurs principales méthodes. Le résultat de cette phase d'analyse peut donner un diagramme de classe incomplet que l'on peut considérer comme le résultat intermédiaire de la phase de conception globale du cycle en V. La méthodologie que nous proposons maintenant vise à prendre chaque classe ainsi identifiée, à la développer complètement et à la valider. Après avoir ainsi traité chacune des classes, le retour au cycle en V prescrit d'effectuer les tests d'intégration puis la livraison au client.

L'approche objet introduit une interprétation particulière du cycle de développement. Dans cette approche, la décomposition du programme consiste d'abord à produire des objets indépendants réalisant conjointement le calcul en échangeant des messages. À l'intérieur de chaque objet, les méthodes sont définies par décomposition descendante, que nous allons

1.	Spécification fonctionnelle	Documentation embarquée
2.	Spécification détaillée de l'interface	Programmation contractuelle
3.	Définition des tests	Classes de tests
4.	Mise en œuvre 4.1 Définition de la structure et invariants 4.2 Mise en œuvre des méthodes	 Programmation axiomatique Décomposition descendante et programmation axiomatique
5.	Vérification	Exécution des classes de tests et perturbation de code

FIG. 6.3 – Grandes étapes de la méthodologies et approches proposées.

présenter plus loin. Les composants sont des groupes d'objets définis par des groupes de classes cohérents assurant conjointement une fonctionnalité bien définie dans l'application. Les tests unitaires consistent à tester chaque groupe de classes isolément, alors que les tests d'intégration demandent de connecter des groupes d'objets de fonctionnalités différentes pour en vérifier le fonctionnement général.

La méthodologie de développement systématique de classes que nous proposons va de la spécification fonctionnelle à la validation des contrats en passant par les tests. Nous ne distinguons pas tests unitaires et tests d'intégration pour plus de simplicité, compte tenu de la taille réduite des applications auxquelles nous nous attaquons ici. Cette méthodologie se décompose en cinq grandes étapes, chacune s'appuyant sur une technique ou une approche plus ou moins formalisée. L'ordre proposé est un ordre idéal puisque la méthodologie est plutôt cyclique, incluant des retours en arrière au fur et à mesure de la détection des erreurs ou des oublis. Les cinq étapes et les approches correspondantes sont résumées à la figure 6.3 et elles sont décrites dans le reste du chapitre. Nous nous appuyons sur un exemple concret pour en illustrer l'utilisation.

6.2 Spécification fonctionnelle de la classe

Dans le contexte de notre méthodologie de développement systématique des classes, la spécification fonctionnelle consiste à produire une première description de cette classe en Java à partir d'une description plus ou moins structurée obtenue après une analyse intiale du problème, description qui peut éventuellement inclure un diagramme de classe UML. Cette première description de la classe doit détailler sa fonction dans l'application ainsi que déclarer et décrire les méthodes identifiées lors de l'analyse. Le résultat sera un premier squelette de classe complètement documenté du point de vue fonctionnel et qui répondra aux deux questions : que représente cette classe dans le programme ? et que permet-elle de faire ?

La documentation est un élément essentiel à tout programme. Malgré tous les efforts pour développer des interfaces graphiques simples, claires et auto-desriptives, l'utilisation d'un programme complexe demande généralement des explications données dans la documentation d'utilisation du programme (manuel d'utilisation). De même, au niveau des composants du programme, ils doivent pouvoir être utilisés par les autres programmeurs du projet (voire d'autres projets), ce qui nécessite la mise à leur disposition d'une documentation d'utilisation des classes complète et à jour décrivant la fonction de la classe et celle de chacune des méthodes (manuel de référence. C'est la première fonction de la documentation.

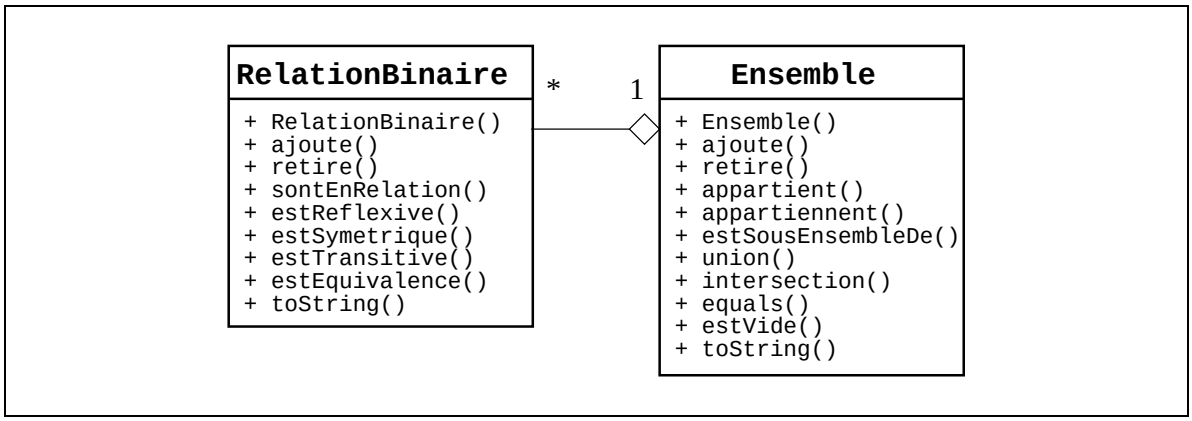


FIG. 6.4 – Diagramme (partiel) de classe `RelationBinaire`.

Bien que l'histoire du développement des langages de programmation soit essentiellement celle de la quête de moyens toujours plus clairs pour exprimer l'intention du programmeur, nous avons vu qu'il est souvent difficile d'inférer l'objectif d'une séquence de lignes de programme par leur seule inspection. L'objectif premier de la documentation de programme est donc de faciliter la compréhension du programme par un autre programmeur lorsque celui-ci l'utilise et surtout lorsqu'il intervient en maintenance sur ce programme. C'est la deuxième fonction de la documentation.

Enfin, la production de logiciel est une activité mobilisant aujourd'hui de grandes équipes, impliquant donc un suivi de projet et une qualité de produit nécessitant la traçabilité complète des composants et de leur évolution. Cette traçabilité est obtenue par documentation du source pour indiquer les éléments essentiels du contexte de développement de la classe : auteur, projet, date de validation dans le référentiel commun, dates et descriptions des modifications faites depuis la première validation, numéro de version courante, nom des audits.[3] C'est la troisième fonction de la documentation.

L'expérience a montré que le code doit être documenté dans le fichier même où il est écrit. C'est la manière la plus sûre de faire suivre le code de sa documentation au cours de sa vie utile, et aussi la manière la plus convaincante de faire en sorte que la documentation soit maintenue à jour lors des modifications du code. Pour introduire la documentation dans un fichier source, tous les langages de programmation offrent une syntaxe de commentaires. Toutes les organisations de production de logiciel imposent un norme de documentation édictant les règles précises devant être suivies par tous les programmeurs pour introduire la documentation de suivi de projet et de traçabilité. Bien que les normes changent d'une organisation à l'autre, l'idée est toujours la même : faciliter le suivi de projet et l'échange de fichiers de source entre programmeurs de l'organisation. Le premier squelette d'une classe est l'occasion de mettre en place cette documentation selon la norme choisie.

En Java, on dispose en plus d'une norme de documentation embarquée appelée Javadoc (voir chapitre 3). Cette norme de documentation a pour principal avantage d'introduire la documentation d'utilisation dans le code source tout en permettant la génération automatique de manuels d'utilisation ou de pages de documentation des classes (manuel de référence) par extraction des commentaires du fichier de code source. Cette norme de documentation doit être utilisée pour toute la documentation fonctionnelle de la classe, c'est-à-dire la description

[3]Une pratique courante dans l'industrie pour assurer la qualité du logiciel est la relecture croisée du code d'un programmeur par un autre programmeur appelé audit pour l'occasion.

```
//      RelationBinaire.java -- relations binaires sur ensembles d'objets

package calculEnsembliste ;

/**
 * <STRONG> Description </STRONG>
 *
 *      La classe RelationBinaire permet de représenter une relation binaire
 *      sur un certain ensemble, lequel est représenté par un objet instance
 *      de la classe Ensemble du même package.
 *
 * <STRONG> Principes et fonctionnalités </STRONG>
 *
 *      Une relation binaire R sur un ensemble E est un sous-ensemble S du
 *      produit cartésien E x E de l'ensemble E avec lui-même tel que si le
 *      couple (a, b) appartient à S, alors on dit que a est en relation R
 *      avec b, noté a R b.  La classe RelationBinaire permet de créer des
 *      objets représentant de telles relations qui sont construites sur un
 *      objet e représentant l'ensemble E sur lequelle elle est définie, puis
 *      par ajout et retrait de couples (a, b) à la relation.  L'objet
 *      représentant une relation binaire R sait répondre à des messages pour
 *      savoir si deux éléments de l'ensemble E sont en relation par R.  Il
 *      sait également répondre à des messages demandant si la relation R
 *      est réflexive, symétrique, transitive, ou d'équivalence.
 *
 * <STRONG> Date </STRONG>
 *
 *      7 septembre 2000
 *
 * <STRONG> Liste des mises à jour </STRONG>
 *
 *<table BORDER=2 COLS=2 WIDTH="100%" NOSAVE ><tr>
 *<td> néant           </td><td>                                          </td>
 *</tr></table>
 *
 * <STRONG> Invariant structurel </STRONG>
 *
 * @invariant   à définir                       // FONCTIONNEL
 * @invariant   à définir                       // REPRÉSENTATIONNEL
 *
 * @author      Jacques Malenfant
 */
public class RelationBinaire
{
    //  --- Méthodes publiques ----------------------------------------
    //

    //  Les méthodes d'initialisation
    //

    // ------------------------------------------------------
    /**
     * initialise la relation à vide
     */
    public              RelationBinaire( ? )
```

FIG. 6.5 – Premier squelette de la classe `RelationBinaire` (1).

de la fonction de la classe et des méthodes. Le premier squelette de la classe doit également mettre en place cette documentation fonctionnelle qui sera élaborée au fur et à mesure du développement subséquent de la classe.

Exemple 54 L'exemple que nous allons utiliser tout au long de ce chapitre est celui d'une classe `RelationBinaire` permettant de créer des objets représentant chacun une relation bi-

```
    //  Les modificateurs
    //

    /**
     * ajoute un couple à la relation
     *
     *  Cette méthode ajoute un couple à la relation, si celui-ci n'y
     *  appartient pas déjà.  Après l'exécution, l'exécution de la
     *  méthode sontEnRelation sur les éléments du couple doit retourner
     *  vrai.  Le receveur est retourné en résultat pour permettre les
     *  cascades.
     */
    public type?        ajoute( ? )

    /**
     * retire un couple à la relation
     *
     *  Cette méthode retire un couple de la relation à condition que
     *  ce couple en fasse effectivement déjà partie.  Après l'exécution
     *  de cette méthode sur un couple donnée, l'appel de la méthode
     *  sontEnRelation sur ce couple doit retourner faux.  Le receveur
     *  est retourné en résultat pour permettre les cascades.
     */
    public type?        retire( ? )
```

FIG. 6.6 – Premier squelette de la classe `RelationBinaire` (2).

naire construite sur un certain ensemble d'objets, lui-même représenté par un objet instance de la classe `Ensemble`. La figure 6.4 donne le diagramme de classe correspondant. Sur la classe `Ensemble`, outre un constructeur, les méthodes permettent d'ajouter (`ajoute`) ou de retirer (`retire`) un élément de l'ensemble. On peut aussi vérifier l'appartenance de un (`appartient`) ou plusieurs (`appartiennent`) éléments à l'ensemble. Les méthodes `estSousEnsembleDe` et `egal` permettent de vérifier si un ensemble est sous-ensemble d'un autre, ou si deux ensembles sont égaux. Enfin, les opérations ensemblistes classiques sont réalisées par les méthodes `union`, `intersection`, et `moins`.

La classe `RelationBinaire`, outre le constructeur de relation sur un certain ensemble d'objets, possède les méthodes suivantes :

- `ajoute` pour ajouter un nouveau couple à la relation,
- `retire` pour retirer un couple de la relation,
- `sontEnRelation` pour vérifier si deux objets de l'ensemble de définition sont en relation dans cette relation,
- `estReflexive` pour vérifier si la relation est réflexive,
- `estSymetrique` pour vérifier si la relation est symétrique,
- `estTransitive` pour vérifier si la relation est transitive,
- `estEquivalence` pour vérifier si la relation est une relation d'équivalence, et
- `toString`, une implantation particulière pour `RelationBinaire` du protocole standard pour produire une représentation sous forme de chaîne de caractères d'un objet relation binaire.

□

À partir de ces informations, un premier squelette de la classe `RelationBinaire` est proposé dans les figures 6.5, 6.6 et 6.7. La classe `Ensemble` quant à elle est présentée complètement à l'annexe A. Le bloc de commentaires Javadoc à la figure 6.5 regroupe documentation d'utilisation et de traçabilité.

L'extraction de la documentation d'utilisation servira à générer un manuel d'utilisation de la classe en format HTML (d'où l'utilisation de balises HTML). Cette documentation présente une description succinte de la classe, les principes et fonctionnalités de cette mise en œuvre. La documentation de traçabilité supporte le suivi et l'intégration de la classe dans le projet de développement logiciel. On y retrouve les informations sur l'auteur et la version de la classe, et parfois aussi sur le projet auquel elle appartient, son auditeur et les mises à jour.

Les figures 6.6 et 6.7 présentent les méthodes publiques de la classe. Des commentaires Java divisent les grandes sections de la classe : méthodes publiques, parmi lesquelles on retrouve les méthodes d'initialisation, les modificateurs et les accesseurs. Chaque déclaration de méthode est précédée d'un commentaire Javadoc qui pour l'instant ne contient qu'une description de la fonction de chacune des méthodes incluant son nom et sa fonction générale et une petite documentation d'utilisation. La déclaration de la méthode peut être incomplète si certains éléments, comme le type du résultat ou le nombre, les types et les noms des paramètres, ne sont pas déterminables de manière immédiate. La prochaine étape de développement, la spécification de l'interface, va compléter cette information.

6.3 Spécification de l'interface et programmation contractuelle

La deuxième étape de notre méthodologie de développement consiste à détailler l'interface de la classe. À cette étape, les principales questions qui se posent sont : quels sont les paramètres de chaque méthode ? quels en sont les résultats ? quelles sont les contraintes posées sur les valeurs de paramètres pour que la méthode s'exécute correctement ? quelles sont les contraintes sur le résultat pour qu'il soit valide ?

L'identification des paramètres et des résultats va permettre de compléter la signature de chaque méthode, c'est-à-dire donner les types et les noms des paramètres de même que les types des résultats. L'identification des contraintes sur les paramètres et sur les résultats va permettre de définir les pré- et postconditions sur chaque méthode.

6.3.1 Programmation contractuelle

Au chapitre 2, les assertions sur l'état des variables ont été utilisées pour raisonner sur l'effet d'un appel à une procédure. Dans la section 3.3 du chapitre 3, nous avons vu comment ces assertions peuvent être utilisées pour définir des invariants structurels (fonctionnels et représentationnels) sur les classes et être associées aux méthodes sous la forme de pré- et postconditions. Rappelons également que l'interprétation de ces pré- et postconditions sur les méthodes est de considérer ces deux assertions comme faisant partie d'un contrat entre l'appelant et l'appelé. Si la méthode est appelée dans un état satisfaisant sa précondition et l'invariant structurel de l'objet, alors elle promet de s'exécuter normalement (sans erreur) et de se terminer dans un état respectant sa postcondition et l'invariant structurel de l'objet.

Cette notion de contractualisation est à la base de ce que Meyer a défini comme la *programmation contractuelle* [Mey97]. Plusieurs analogies avec des situations réelles peuvent être données pour justifier cette approche contractuelle. Quand une automobile est confiée au garagiste pour une réparation, il existe un contrat entre le client (l'appelant) et le garagiste (l'appelé). Il est acquis par exemple que le garagiste est bien mécanicien ou qu'il dispose de mécaniciens compétents (invariant structurel) et que vous, le client, lui apportez bien une automobile et pas une machine à laver le linge (précondition). Lorsque le garagiste donne un rendez-vous, il a vérifié qu'il était libre ; il doit donc être libre lorsque le client se présente

```
    // Les accesseurs
    //

    /**
     * vérifie si les deux paramètres sont en relation
     */
    public boolean      sontEnRelation( ? )

    /**
     * vérifie si la relation est réflexive
     *
     *    Une relation binaire R est réflexive si pour tout élément e, e
     *    est en relation avec lui-même.
     */
    public boolean      estReflexive( ? )

    /**
     * vérifie si la relation est symétrique
     *
     *    Une relation binaire R est symétrique si pour tous éléments a
     *    et b, si a est en relation avec b, alors b est en relation avec a.
     */
    public boolean      estSymetrique( ? )

    /**
     * vérifie si la relation est transitive
     *
     *    Une relation est transitive si et seulement si pour tous éléments a,
     *    b et c, si a est en relation avec b et b est en relation avec c,
     *    alors a est en relation avec c.
     */
    public boolean      estTransitive( ? )

    /**
     * vérifie si la relation est une relation d'équivalence
     *
     *    Une relation est une relation d'équivalence si elle est réflexive,
     *    symétrique et transitive.
     */
    public boolean      estEquivalence( ? )

    /**
     * retourne une représentation chaîne de caractères
     *
     *    Cette méthode retourne une représentation de la relation sous la
     *    forme d'une chaîne de caractères obéissant au format suivant :
     *    <OL>
     *    <LI> relation vide : {}
     *    <LI> relation contenant les couples (1,2), et (2,3) : {(1,2), (2,3)}
     *    <OL>
     *    L'ordre dans lequel les couples apparaissent n'a pas d'importance,
     *    i.e. dans le second exemple, on pourrait tout aussi bien obtenir
     *    {(2,3), (1,2)}.
     */
    public String       toString()

} // ------------------------------------------------- Classe RelationBinaire
```

FIG. 6.7 – Premier squelette de la classe `RelationBinaire` (3).

(précondition supplémentaire sur l'état de l'objet). Après la réparation, le garagiste doit rendre une automobile en bon état de marche (postcondition sur le résultat). Il faut aussi qu'il soit toujours garagiste (invariant structurel) et de surcroît libre à nouveau pour accueillir un autre client (postcondition supplémentaire sur l'état de l'objet).

Pour concrétiser informatiquement la notion de programmation contractuelle, reprenons l'exemple du rectangle représenté par ses deux points supérieur gauche et inférieur droit. La méthode `translate` permet de bouger le rectangle dans le plan cartésien en bougeant ses deux points. Lorsqu'on exécute la méthode `translate` (il n'y a pas de conditions particulières sur les paramètres de l'appel), le rectangle est dans un état initial respectant son invariant structurel. À l'issue de la méthode, le rectangle doit être à nouveau dans un état final respectant son invariant structurel. Mais on peut aussi dire des choses en plus sur la relation entre l'état initial et l'état final : l'état final doit être celui obtenu par application de la translation spécifiée par les paramètres d'appel sur l'état initial. La postcondition permet alors de vérifier que la méthode a bien fait son travail.

La programmation contractuelle cherche à intégrer une notion de contrôle de qualité dans la programmation. Dans toute industrie, on vérifie que les matériaux utilisés dans la production sont corrects (précondition) et que les produits ont la qualité requise avant livraison (postcondition).

Aux chapitres 2 et 4, nous avons utilisé les assertions pour annoter les énoncés de telle manière à raisonner sur l'exactitude d'une séquence d'énoncés par rapport à ce qu'elle doit calculer. L'approche utilisant systématiquement les assertions est appelée l'approche axiomatique. Quelle est la différence entre les approches axiomatiques et contractuelles ? En fait, l'approche contractuelle exploite les notions d'assertions pour mettre en œuvre la métaphore du contrat entre objets clients et serveurs. Elle est cependant d'une puissance plus limitée que l'approche axiomatique dans le sens suivant.

D'un point de vue pragmatique, l'approche contractuelle insiste sur le fait que les invariants, pré- et postconditions sont vérifiées à l'exécution du programme au moins durant la phase de développement. L'approche axiomatique, elle, utilise les assertions pour raisonner et éventuellement prouver l'exactitude d'un programme avant même son exécution. Parce que les expressions de la logique du premier ordre généralement utilisée dans les assertions ne sont pas toujours exécutables, l'approche contractuelle peut aussi être vue comme une version affaiblie mais concrètement opérationnelle de l'approche axiomatique.

Typiquement, une assertion quantifiée universellement ($\forall$), ce qui impliquerait en approche axiomatique une preuve sur l'ensemble des éléments visé par le quantificateur. En pratique, il est impossible de vérifier exhaustivement tous les cas. En programmation contractuelle par une construction qui va, on se limitera à vérifier que l'assertion est vraie pour une nombre fini et relativement petit de valeurs. De manière similaire, une assertion quantifiée existentiellement ($\exists$), qui impliquerait en approche axiomatique de trouver un élement parmi l'ensemble des éléments visé par le quantificateur pour lequel l'assertion est vraie, va se traduire en approche contractuelle par l'identification d'une valeur qui vérifie l'assertion parmi un nombre fini de valeurs à ce point du programme.

Par exemple, si on dit dans une assertion «pour tout x dans l'ensemble des entiers, p(x) est vrai», cela devra se traduire en programmation contractuelle par une expression qui va balayer par exemple un tableau d'entiers et vérifier que p(x) est vrai pour tous les entiers dans ce tableau. Cela revient donc à vérifier exhaustivement la propriété mais uniquement pour les éléments du tableau, or on sait bien qu'en mathématique on ne prouve pas un propriété quantifiée universellement par vérification exhaustive, puisque la plupart des ensembles sont de cardinalité finie. L'approche axiomatique apparaît donc supérieure, mais se heurte par contre à la difficulté bien connue de construire des preuves en mathématique. De plus, l'automatisation complète de la démonstration mathématique de la validité d'un programme reste encore du domaine de la recherche. Vu la taille des programmes, la quantité de preuves à faire est hors de la portée humaine. Tout n'est donc pas rose non plus du côté axiomatique !

La passage de l'approche axiomatique à la programmation contractuelle est donc un compromis entre la preuve formelle difficile à obtenir et le test exhaustif sur des ensembles de cardinalité bornée qui ne prouve jamais le cas général, mais vérifie simplement le cas précis de l'exécution en cours. En théorie, la preuve formelle est hautement souhaitable. En pratique, la vérification a montré son utilité en produisant des programmes de meilleure qualité.

6.3.2 Spécification de l'interface de la classe

Cette étape consiste donc à passer en revue chacune des méthodes pour identifier les paramètres et les résultats attendus, de même que pour définir les contrats au moins partiels de chacune. Les contrats pourront être incomplet dans la mesure où nous ne nous sommes pas encore interrogés sur le représentation des objets et donc sur les variables d'instance. Les contrats où nous devons arriver à cette étape sont donc des contrats purement sur la fonctionnalité de chacune des méthodes. Si des contraintes apparaissent du fait de la réprésentation choisie, ces éléments de contrats seront ajoutés à l'étape suivante.

Pour la méthode d'initialisation `RelationBinaire`, il est clair que toute relation binaire est construite sur un ensemble de définition existant. Il faut donc nécessairement que la méthode d'initialisation reçoive une instance de la classe Ensemble en paramètre. Cet ensemble doit être défini, on exclut donc la possiblité que la constante `null` puisse être passée en paramètre lors d'un appel. À la fin de l'exécution de la méthode, il faudra que l'objet intialisé ait bien mémorisé que l'ensemble passé en paramètre est son ensemble de définition. Pour vérifier cela, il faut être en mesure d'accéder à l'ensemble de définition de l'objet relation binaire, ou encore être en mesure de vérifier si un ensemble donné est l'ensemble de définition de l'objet. Si on choisit la première alternative, il faut ajouter à la classe une méthode `lEnsembleDeDefinition` qui retourne l'ensemble de définition de l'objet. Cette méthode peut être définie comme privée, protégée, de paquetage ou publique ; nous choisissons la dernière solution puisque cela ne pose pas de problème particulier d'accéder à l'ensemble de définition. Cela nous conduit donc à la mise à jour suivante dans le fichier de la classe :

```
/**
 * initialise la relation à vide
 *
 *    Une fois initialisée, l'ensemble de définition de la relation doit
 *    être défini.
 *
 * @param   ensemble sur lequel la relation est définie
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     e != null                       // ensemble bien défini
 * @post    lEnsembleDeDefinition() == e    // ensemble mémorisé
 */
public              RelationBinaire(
    Ensemble e
    )
```

Pour la méthode `ajoute`, il faut donner le couple à ajouter à la relation, c'est-à-dire un élément du domaine de la relation et son correspondant du codomaine. Leur type est `Object`, puisque nos ensembles sont des ensembles d'objets. La méthode `ajoute` est un modificateur ; il n'y a pas a priori de raison de lui faire retourner un résultat. Cependant, il peut être utile d'ajouter successivement plusieurs couples à la relation. Pour cela, plutôt que d'écrire par exemple :

```
rb.ajoute(new Integer(1), new Integer(2)) ;
```

```
rb.ajoute(new Integer(3), new Integer(2)) ;
```

on pourra aussi écrire un série d'envoi de messages :

```
rb.ajoute(new Integer(1), new Integer(2)).ajoute(new Integer(3), new Integer(2)) ;
```

ce qui peut s'avérer plus clair dans certains cas. Pour cela, il suffit de retourner le receveur du message en résultat, qui pourra alors servir de destinataire du message suivant. Le type du résultat de `ajoute` doit donc être `RelationBinaire`. Pour ce qui est du contrat, il faut que les deux éléments passés en paramètres soient de l'ensemble de définition, et après l'exécution de la méthode, les deux éléments sont en relation et la résultat est le receveur du message. Cela donne donc :

```
/**
 * ajoute un couple à la relation
 *
 *    Cette méthode ajoute un couple à la relation, si celui-ci n'y
 *    appartient pas déjà.  Après l'exécution, l'exécution de la
 *    méthode sontEnRelation sur les éléments du couple doit retourner
 *    vrai.  Le receveur est retourné en résultat pour permettre les
 *    cascades.
 *
 * @param   élément du domaine
 * @param   élément du codomaine
 * @return  retourne le receveur
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     lensembleDeDefinition().appartient(dom)
 *                                      // dom dans l'ensemble
 * @pre     lensembleDeDefinition().appartient(codom)
 *                                      // codom dans l'ensemble
 * @post    sontEnRelation(dom, codom)      // relation établie
 * @post    this == return                  // retourne soi-même
 */
public RelationBinaire      ajoute(
    Object dom,
    Object codom
    )
```

La méthode `retire` est très similaire à la méthode `ajoute`. Pour retirer un couple, il faut en connaître l'élément du domaine et celui du codomaine, et donc les recevoir en paramètre. Le résultat est de type `RelationBinaire` pour retourner le receveur comme résultat de façon à permettre l'écriture de cascades de messages pour ajouter ou retirer des couples. Pour ce qui concerne le contrat, les paramètres doivent être éléments de l'ensemble de définition, et ils doivent être en relation avant l'appel pour que l'opération ait un sens. Après l'exécution de la méthode, les éléments ne sont plus en relation et le résultat doit être égal au receveur du message :

```
/**
 * retire un couple à la relation
 *
 *    Cette méthode retire un couple de la relation à condition que
 *    ce couple en fasse effectivement déjà partie.  Après l'exécution
 *    de cette méthode sur un couple donnée, l'appel de la méthode
 *    sontEnRelation sur ce couple doit retourner faux.  Le receveur
 *    est retourné en résultat pour permettre les cascades.
 *
 * @param   élément du domaine
```

```
 * @param   élément du codomaine
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     lensembleDeDefinition().appartient(dom)
 *                                          // dom dans l'ensemble
 * @pre     lensembleDeDefinition().appartient(codom)
 *                                          // codom dans l'ensemble
 * @pre     sontEnRelation(dom, codom)      // couple existe
 * @post    !sontEnRelation(dom, codom)     // relation brisée
 * @post    this == return                  // retourne soi-même
 */
public RelationBinaire      retire(
    Object dom,
    Object codom
    )
```

Les méthodes suivantes sont des accesseurs qui, par définition, ne modifient pas l'état de l'objet receveur. L'identification des paramètres et des préconditions est cependant similaire au cas des modificateurs. Pour la méthode `sontEnRelation`, il faut connaître les éléments supposés du domaine et du codomaine de la relation, et ceux-ci doivent nécessairement faire partie de l'ensemble de définition de la relation.

Du point de vue postconditions, il n'y a rien à vérifier sur l'état de l'objet. Parce qu'ils sont appelés pour leur résultat, celui-ci est cependant plus complexe à calculer généralement. Les postconditions sur le résultat sont là pour vérifier que la méthode fonctionne bien, mais répéter le code de la méthode ne sert à rien puisque si erreur il y avait, celle-ci serait simplement répétée et donc pas captée. Nous avons vu à la section 4.5 du chapitre 4 par l'exemple de la racine carrée, qu'une postcondition sur le résultat n'est utile que si le calcul qui y est fait pour vérifier est différent de celui de la méthode. Dans le cas de la méthode `sontEnRelation`, il n'y a pas de moyen simple et efficace de vérifier le résultat ; nous renonçons donc à la postcondition correspondante. On obtient la spécification suivante :

```
/**
 * vérifie si les deux paramètres sont en relation
 *
 * @param   élément du domaine
 * @param   élément du codomaine
 * @return  vrai si dom est en relation avec codom
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     lensembleDeDefinition().appartient(dom)
 *                                          // dom dans l'ensemble
 * @pre     lensembleDeDefinition().appartient(codom)
 *                                          // codom dans l'ensemble
 */
public boolean      sontEnRelation(
    Object dom,
    Object codom
    )
```

Pour les quatre méthodes `estReflexive`, `estSymétrique`, `estTransitive`, et `estEquivalence`, il n'y a pas de paramètres puisqu'à chaque fois l'opération consiste à vérifier une propriété sur la relation binaire représentée par le receveur. À chaque fois le résultat est booléen. Les postconditions font largement appel aux expressions `forall` du langage de contrats dont la forme syntaxque est la suivante :

```
forall <Classe> <var> in <Enumeration> | <exp_var>
```

`<Enumeration>` est une expression retournant une valeur de type `Enumeration` de Java, un visiteur qui permettra de parcourir toutes les valeurs d'une collection sur laquelle la propriété doit être vérifiée. `<var>` est une variable de type `<Classe>` qui va successivement contenir chaque valeur visitée. `<exp_var>` est une expression à résultat booléen faisant référence à `<var>` et qui exprime la propriété souhaitée.

Pour la méthode `estReflexive`, une relation est réflexive si et seulement tout élément de l'ensemble de définition est en relation avec lui-même. Il faut donc parcourir tous les éléments de l'ensemble de définition pour vérifier s'ils sont en relation avec eux-mêmes. En supposant que la classe `Ensemble` définit une méthode `enumerate` retournant un visiteur sur l'ensemble, on obtient :

```
/**
 * vérifie si la relation est réflexive
 *
 *    Une relation binaire R est réflexive si pour tout élément e, e
 *    est en relation avec lui-même.
 *
 * @return  vrai si la relation est réflexive et faux sinon
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     true
 * @post    return == (forall Object e in
 *                                    lEnsembleDeDefinition().enumerate() |
 *                          this.sontEnRelation(e, e))
 */
public boolean      estReflexive()
```

De manière similaire, une relation est symétrique si pour tout a et b tels que $a\ R\ b$ on a $b\ R\ a$. Cette propriété est exprimée par un double parcours de l'ensemble de définition et l'implication que pour tout couple de valeurs en relation, le couple symétrique est aussi en relation :

```
/**
 * vérifie si la relation est symétrique
 *
 *    Une relation binaire R est symétrique si pour tous éléments a
 *    et b, si a est en relation avec b, alors b est en relation avec a.
 *
 * @return  vrai si la relation est symétrique et faux sinon
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     true
 * @post    return == (forall Object e1 in
 *                                    lEnsembleDeDefinition().enumerate() |
 *                          (forall Object e2 in
 *                                    lEnsembleDeDefinition().enumerate() |
 *                              this.sontEnRelation(e1, e2)
 *                                  implies this.sontEnRelation(e2, e1)))
 */
public boolean      estSymetrique()
```

Une relation est transitive si pour tout a, b et c, $a\ R\ b$ et $b\ R\ c$ implique $a\ R\ c$:

```
/**
 * vérifie si la relation est transitive
 *
 *    Une relation est transitive si et seulement si pour tous éléments a,
```

```
 *    b et c, si a est en relation avec b et b est en relation avec c,
 *    alors a est en relation avec c.
 *
 * @return  vrai si la relation est transitive et faux sinon
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     true
 * @post    return == (forall Object e1 in
 *                                    lEnsembleDeDefinition().enumerate() |
 *                         (forall Object e2 in
 *                                    lEnsembleDeDefinition().enumerate() |
 *                             (forall Object e3 in
 *                                    lEnsembleDeDefinition().enumerate() |
 *                                 (this.sontEnRelation(e1, e2)
 *                                           && this.sontEnRelation(e2, e3)
 *                                    implies
 *                                        this.sontEnRelation(e1, e3)))))
 */
public boolean      estTransitive()
```

Enfin, une relation est une relation d'équivalence si et seulement si elle est réflexive, symétrique et transitive, ce qui s'exprime évidemment en utilisant les trois méthodes précédentes :

```
/**
 * vérifie si la relation est une relation d'équivalence
 *
 *    Une relation est une relation d'équivamence si elle est réflexive,
 *    symétrique et transitive.
 *
 * @return  vrai si la relation est une équivalence et faux sinon
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     true
 * @post    return == (this.estReflexive() && this.estSymetrique()
 *                     && this.estTransitive())
 */
public boolean      estEquivalence()
```

Pour la méthode `toString`, il n'y a pas de paramètre non plus, et le résultat est encore une fois trop complexe pour qu'on tente de le contrevérifier de manière efficace par une postcondition. Par contre, une condition affaiblie peut nous permettre de capter des erreurs éventuelles. Ici, on peut déjà vérifier que la chaîne retournée est bien définie (c'est-à-dire pas la constante `null`). Il arrive assez souvent en programmation contractuelle que la postcondition choisie soit un affaiblissement de la vraie postcondition au sens axiomatique, affaiblissement qui en soit ne prouve pas que la méthode est correcte, mais qui peut utilement détecter certaines erreurs. On pourrait renforcer cette postcondition précédente en comptant le nombre de couples représentés dans la chaîne et vérifier s'il correspond au nombre de couples de la relation. La spécification retenue pour la méthode est la plus simple et nous donne :

```
/**
 * toString :       retourne une représentation chaîne de caractères
 *
 *    Cette méthode retourne une représentation de la relation sous la
 *    forme d'une chaîne de caractères obéissant au format suivant :
 *    <OL>
 *    <LI> relation vide : {}
 *    <LI> relation contenant les couples (1,2), et (2,3) : {(1,2), (2,3)}
 *    <OL>
```

```
     *    L'ordre dans lequel les couples apparaissent n'a pas d'importance,
     *    i.e. dans le second exemple, on pourrait tout aussi bien obtenir
     *    {(2,3), (1,2)}.
     *
     * @return  chaîne représentant le contenu de la relation
     *
     * <STRONG> Contrat </STRONG>
     *
     * @post    return != null                  // chaîne existe
     */
    public String       toString()
```

6.4 Définition des tests et classes de test

À partir de la définition complète des interfaces, il est directement possible de préparer les tests à faire pour vérifier la classe, et ce avant même de se lancer dans sa mise en œuvre. Il y a deux bonnes raisons pour procéder à cette étape de définition des tests avant la mise en œuvre :

1. L'expérience montre que la pression pour livrer les applications aux clients monte de manière continue pendant la durée du projet pour arriver à son plus haut niveau à la fin du processus de développement. Dans l'ordre plus classique des choses, c'est-à-dire préparer les tests après la fin de la mise en œuvre, l'étape qui subit en général une forte compression des moyens alloués est celle de la préparation des tests. Or, un logiciel incomplètement ou trop peu testé est un logiciel de mauvaise qualité qui va induire un grand nombre d'erreurs détectées à l'utilisation, donc d'opérations de maintenance ce qui, on l'a déjà dit, est une source d'augmentation des coûts globaux du projet. Dans un contexte concurrentiel, trop d'erreurs détectées à l'utilisation est source d'insatisfaction et de perte de clientèle.
2. En préparant les séquences de test avant la mise en œuvre, on gagne aussi du temps en se servant d'un sous-ensemble[4] des séquences de tests de vérification pour la mise au point du logiciel. On évite ainsi une prolifération de petits tests développés par les programmeurs individuellement pour les besoins de mise au point de leur code.

6.4.1 Le test du logiciel

Le test est un sujet de la plus haute importance en génie logiciel car, à défaut de moyens de prouver formellement l'exactitude des programmes, c'est par le test que l'on juge le plus souvent de la qualité des programmes. On a vu depuis le début de ce module qu'un programme se caractérise par des chemins d'exacution qui dépendent des valeurs soumises en entrée du programme. En effet, pour certaines valeurs d'entrées, les alternatives exécuteront l'une ou l'autre de leurs branches ; pour certaines valeurs d'entrées, certaines répétition exécuteront leur corps ou non. Tous ces chemins d'exécution dans le programme sont susceptibles de contenir des erreurs. Le degré d'aptitude d'une séquence de tests à faire exécuter tous les

[4]On précise sous-ensemble des séquences car le danger d'utiliser toutes les séquences de test au moment de la mise en œuvre est d'inciter les programmeurs à produire du code qui ne se conforme qu'aux tests prévus, et non aux spécifications plus générales. Un phénomène similaire existe en apprentissage pour les réseaux neuromimétiques ou encore par des processus statistiques. On utilise un sous-ensembles des données pour faire l'apprentissage, et le reste pour vérifier la qualité de cet apprentissage. L'idée ici est similaire.

```
//      RelationBinaireT.java -- Test de la classe RelationBinaire

package calculEnsembliste ;

public  class   RelationBinaireT
{

    /**
     * afficheMessageTest       : affiche un message positif ou négatif
     *
     * @param cond      : condition à vérifier
     */
    public static void          afficheMessageTest(
        boolean cond    // condition à vérifier
        ) {

        if ( cond ) {
            System.out.println( "Test OK" ) ;
        } else {
            System.out.println( " *** Test infructueux *** " ) ;
        }                                       //  if ( cond )
    } // ------------------------------------------------ afficheMessageTest()

    /**
     * main :   lanceur du test
     */
    public static void          main (String[] args)
    {

        //  --- Variables locales  -----------------------
        //
        RelationBinaire unObj ;
        RelationBinaire unAutreObj ;
        Ensemble        e ;
        Object[]        initiales = {new Integer(1), new Integer(2),
                                     new Integer(2), new Integer(3),
                                     new Integer(4), new Integer(5)} ;
        String          res ;
        boolean         resBool1, resBool2, resBool3 ;

        System.out.println ("\nTest de la classe 'RelationBinaire'") ;

// DebTest ---------------------------------------------
//
        System.out.println("\n    Séquence n.1 : construction et affichage\n") ;
        e = new Ensemble(initiales) ;
        unObj = new RelationBinaire(e) ;
        res = unObj.toString() ;
        System.out.println(res) ;
        afficheMessageTest(res.compareTo("{}") == 0) ;

        System.out.println("\n    Séquence n.2 : ajoute\n") ;
        unObj.ajoute(new Integer(1), new Integer(5)) ;
        res = unObj.toString() ;
        System.out.println(res) ;
        afficheMessageTest(res.compareTo("{(1,5)}") == 0) ;
```

FIG. 6.8 – Classe de test `RelationBinaireT` (1).

chemins d'exécutions possibles d'un programme est ce que l'on appelle la notion de *couverture* des tests.

Une séquence de test est formée d'un ensemble de valeurs des paramètres d'entrée à un programme ou à une portion de programme (méthode par exemple). Une bonne séquence de test est celle qui couvre tous les chemins d'exécution possibles dans la portion de programme

```
        System.out.println("\n    Séquence n.3 : retire\n") ;
        unObj = new RelationBinaire(e) ;
        unObj.ajoute(new Integer(1), new Integer(5)).ajoute(new Integer(2), new Integer(2))
             .ajoute(new Integer(4), new Integer(3)) ;
        unObj.retire(new Integer(2), new Integer(2)) ;
        res = unObj.toString() ;
        System.out.println(res) ;
        afficheMessageTest(res.compareTo("{(1,5), (4,3)}") == 0) ;

        System.out.println("\n    Séquence n.4 : sontEnRelation\n") ;
        System.out.print("1 est en relation avec 5 dans "
                         + unObj.toString() + " ?  ") ;
        resBool1 = unObj.sontEnRelation(new Integer(1), new Integer(5)) ;
        System.out.println( resBool1 + "") ;
        System.out.print("3 est en relation avec 4 dans "
                         + unObj.toString() + " ?  ") ;
        resBool2 = unObj.sontEnRelation(new Integer(3), new Integer(4)) ;
        System.out.println( resBool2 + "") ;
        unObj = new RelationBinaire(e) ;
        System.out.print("1 est en relation avec 5 dans "
                         + unObj.toString() + " ?  ") ;
        resBool3 = unObj.sontEnRelation(new Integer(1), new Integer(5)) ;
        System.out.println( resBool3 + "") ;
        afficheMessageTest(resBool1 && !resBool2 && !resBool3) ;

        System.out.println("\n    Séquence n.5 : estReflexive\n") ;
        unObj = new RelationBinaire(e) ;
        unObj.ajoute(new Integer(1), new Integer(5)).ajoute(new Integer(2), new Integer(2))
             .ajoute(new Integer(4), new Integer(3)) ;
        System.out.print("La relation " + unObj.toString()
                           + " est-elle réflexive ?  ") ;
        resBool1 = unObj.estReflexive() ;
        System.out.println( resBool1 + "") ;
        unObj = new RelationBinaire(e) ;
        unObj.ajoute(new Integer(1), new Integer(5)).ajoute(new Integer(1), new Integer(1))
             .ajoute(new Integer(2), new Integer(2)).ajoute(new Integer(4), new Integer(3))
             .ajoute(new Integer(3), new Integer(3)).ajoute(new Integer(4), new Integer(4))
             .ajoute(new Integer(5), new Integer(5)) ;
        System.out.print("La relation " + unObj.toString()
                           + " est-elle réflexive ?  ") ;
        resBool2 = unObj.estReflexive() ;
        System.out.println( resBool2 + "") ;
        afficheMessageTest(!resBool1 && resBool2) ;
```

FIG. 6.9 – Classe de test `RelationBinaireT` (2).

à tester. Malheureusement, le nombre de chemins d'exécution augmente très rapidement (exponentiellement) avec la taille des programmes. Il est donc très difficile de construire des séquences de tests ayant une couverture totale tout en ayant une taille raisonnable.

Ces difficultés peuvent être atténuées en pratiquant le test indépendant des unités de logiciel, ce qui permet de réduire l'explosion combinatoire du nombre de cas de test à produire. Ensuite, on peut utiliser les assertions pour identifier les frontières entre les ensembles de paramètres qui vont susciter le passage entre une branche de programme ou une autre.

Par exemple, une alternative découpe l'état en deux sous-ensembles ; un cas de test par sous-ensemble est généralement suffisant. Pour les répétitions, il y a les états où le corps n'est pas exécutés, ceux où il est exécuté une fois et les autres. Il suffit alors généralement de produire un cas de test par sous-ensemble pour avoir une bonne couverture de l'unité testée. Même en utilisant systématiquement cette approche, le nombre de cas de test nécessaires croît exponentiellement avec le nombre de points de choix dans le programme. Heureusement, un cas de test assure généralement le passage dans plusieurs chemins d'exécution. Des recherches sont

```
        System.out.println("\n    Séquence n.6 : estSymetrique\n") ;
        unObj = new RelationBinaire(e) ;
        unObj.ajoute(new Integer(1), new Integer(5)).ajoute(new Integer(2), new Integer(2))
             .ajoute(new Integer(4), new Integer(3)) ;
        System.out.print("La relation " + unObj.toString()
                         + " est-elle symétrique ?  ") ;
        resBool1 = unObj.estSymetrique() ;
        System.out.println( resBool1 + "") ;
        unObj = new RelationBinaire(e) ;
        unObj.ajoute(new Integer(1), new Integer(5)).ajoute(new Integer(5), new Integer(1))
             .ajoute(new Integer(4), new Integer(3)).ajoute(new Integer(3), new Integer(4)) ;
        System.out.print("La relation " + unObj.toString()
                         + " est-elle symétrique ?  ") ;
        resBool2 = unObj.estSymetrique() ;
        System.out.println( resBool2 + "") ;
        afficheMessageTest(!resBool1 && resBool2) ;

        System.out.println("\n    Séquence n.7 : estTransitive\n") ;
        unObj = new RelationBinaire(e) ;
        unObj.ajoute(new Integer(1), new Integer(5)).ajoute(new Integer(2), new Integer(2))
             .ajoute(new Integer(5), new Integer(3)) ;
        System.out.print("La relation " + unObj.toString()
                         + " est-elle transitive ? ") ;
        resBool1 = unObj.estTransitive() ;
        System.out.println( resBool1 + "") ;
        unObj = new RelationBinaire(e) ;
        unObj.ajoute(new Integer(1), new Integer(5)).ajoute(new Integer(2), new Integer(2))
             .ajoute(new Integer(5), new Integer(3)).ajoute(new Integer(1), new Integer(3)) ;
        System.out.print("La relation " + unObj.toString()
                         + " est-elle transitive ?  ") ;
        resBool2 = unObj.estTransitive() ;
        System.out.println( resBool2 + "") ;
        afficheMessageTest(!resBool1 && resBool2) ;

        System.out.println("\n    Séquence n.8 : estEquivalence\n") ;
        unObj = new RelationBinaire(e) ;
        unObj.ajoute(new Integer(1), new Integer(5)).ajoute(new Integer(2), new Integer(2))
             .ajoute(new Integer(5), new Integer(3)).ajoute(new Integer(1), new Integer(3)) ;
        System.out.print("La relation " + unObj.toString()
                         + " est-elle une relation d'équivalence ?  ") ;
        resBool1 = unObj.estEquivalence() ;
        System.out.println( resBool1 + "") ;
        unObj.ajoute(new Integer(5), new Integer(1)).ajoute(new Integer(3), new Integer(5))
             .ajoute(new Integer(3), new Integer(1)).ajoute(new Integer(1), new Integer(1))
             .ajoute(new Integer(3), new Integer(3)).ajoute(new Integer(4), new Integer(4))
             .ajoute(new Integer(5), new Integer(5)) ;
        System.out.print("La relation " + unObj.toString()
                         + " est-elle une relation d'équivalence ?  ") ;
        resBool2 = unObj.estEquivalence() ;
        System.out.println( resBool2 + "") ;
        afficheMessageTest(!resBool1 && resBool2) ;

        System.out.println("\n\n\nFin du test de 'RelationBinaire'") ;

// FinTest ------------------------------------------------
//

    } // ------------------------------------------------- main()

} // ---------------------------------------- Test de RelationBinaire
```

FIG. 6.10 – Classe de test `RelationBinaireT` (3).

menées dans le domaine de la génération automatique des cas de test qui ont pour objectif de générer le plus petit ensemble de cas de tests qui couvre tous les chemins d'exécution possibles.

6.4.2 Définition des tests sur la classe

Notre méthodologie propose essentiellement des heuristiques de travail pour développer les classes. Concernant les tests, les heuristiques que nous proposons consistent à préparer une classe de test pour chacune des classes développées, appelée du même nom que la classe testée suffixé d'un `"T"`. Cette classe est une classe racine qui doit définir une méthode `main` contenant une séquence de test par méthode de la classe testée. Pour chacune de ces séquences, à cette étape de notre méthodologie de développement, l'objectif est de tester la fonctionnalité de la méthode.

Une heuristique simple est par exemple que pour toute méthode à résultat booléen, il faut au minimum tester un cas positif et un cas négatif. Par exemple, dans la classe `RelationBinaire`, sur la méthode `sontEnRelation`, il faut tester au minimum un cas où le résultat doit être vrai et un cas où le résultat doit être faux. Une seconde heuristique consiste à toujours inclure des tests sur les cas limites, car c'est souvent là que les erreurs se produisent. Pour la méthode `sontEnRelation`, un cas limite peut se produire lorsque la relation contient aucun couple, ce qui mérite donc d'être testé. Même chose pour la méthode `toString` pour laquelle une relation vide est probablement un cas qui peut causer des problèmes.

L'exécution de la classe de test doit produire un rapport (sorties à l'écran dans le cas le plus simple) des séquences de tests. L'objectif de ce rapport est d'attirer l'attention sur les cas problème. S'agissant d'un programme, la séquence de test peut utiliser tous les moyens de la programmation pour vérifier les résultats et signaler les cas problème. En fait, chaque test est doté d'un *oracle*, c'est-à-dire une procédure de décision qui dit si le résultat est correct ou non. Dans la classe `RelationBinaireT`, chaque séquence de tests se termine par quelques énoncés qui vérifient si les résultats sont conformes à ce qui était attendu selon le programmeur du test (jouant donc le rôle d'oracle en l'occurrence) et signalent par un message `"*** Test infructueux ***"` ou `"Test OK"` le cas approprié.

Les figures 6.8, 6.9 et 6.10 présentent une classe de tests `RelationBinaireT` préparée pour la vérification de la classe `RelationBinaire`. L'ordre dans lequel les tests sont faits tient compte des dépendances entre les méthodes et aussi des besoins du test. Il est clair par exemple que la méthode `toString` est particulièrement utile à la production du rapport de test. Il va de soi qu'on la teste le plus tôt possible dans la séquence.

L'adoption d'un format relativement strict dans le rapport de test est utile pour rechercher ensuite les résultats parmi la masse d'informations produites. La chose peut être rendue graduellement plus sophistiquée. Le format et la construction du test de la classe `RelationBinaire` vous donne un exemple qui reste volontairement assez simple.

6.5 Mise en œuvre

La quatrième étape est celle de la mise en œuvre de la classe, considérée à tort comme la plus importante parce qu'elle a pour résultat la logiciel lui-même. Nous disons à tort parce qu'un logiciel qui n'est pas vérifié systématiquement doit être considéré comme incorrect.[5] S'il est mal conçu et non-documenté, c'est au mieux un logiciel jetable après usage, c'est-à-dire qu'on ne peut le considéré comme un investissement. Néanmoins, il n'est pas de logiciel s'il

[5]Un professeur de statistiques nous disait lorsque j'étais étudiant qu'un calcul qui n'a pas été fait deux fois et de deux façons différentes doit être considéré comme faux, peu importe son résultat. Je reprends la même idée ici : un logiciel non vérifié est considéré comme incorrect, peu importe les résultats qu'il donne à l'exécution.

n'est pas mis en œuvre. La quatrième étape est donc celle de la programmation comme telle. Elle passe d'abord par le choix d'une représentation pour les objets de la classe, puis par la programmation de chacune des méthodes.

6.5.1 Définition de la structure et invariants

La définition de la structure des instances de la classe est d'abord la conséquence immédiate des informations qu'il faut conserver dans l'objet. Bien souvent, l'analyse du problème dicte des informations que chaque objet va devoir conserver parce qu'il s'agit de son rôle dans l'application. Un objet client dans une application commerciale va nécessairement contenir toute l'information nécessaire pour l'identification et le contact avec ce client : nom, adresse, numéro de téléphone, numéro de compte, etc. De même, un objet représentant une facture doit contenir une date, un client, un montant à payer, un mode de paiment, etc. C'est lors de l'analyse du problème que la plupart de ces informations sont identifiées, et elles peuvent être incluses dans le diagramme de classe sous la forme de variables d'instances dans la classe.

Le diagramme de classe indique également les relations existant entre les objets de différentes classes. Si relation il y a, comme dans notre exemple entre la classe `RelationBinaire` et la classe `Ensemble`, il est normal de de matérialiser cette relation en faisant apparaître dans un objet une variable permettant de mémoriser l'identificateur d'objet de l'autre. Pour les objets instances de `RelationBinaire`, une variable `ensembleDeDefinition` permet de conserver dans l'objet représentant la relation l'identificateur de l'objet représentant l'ensemble de définition de la relation.

Outre des informations d'identification simples, les objets peuvent également servir à collecter un ensemble de données. Dans notre exemple de la classe `RelationBinaire`, une relation binaire doit contenir des couples d'éléments du domaine et du codomaine de la relation. Définir la structure de l'instance devient alors l'affaire d'un choix entre plusieurs approches pour mémoriser ces informations.

Nous choisissons ici de mémoriser les couples dans deux tableaux d'éléments de l'ensemble de définition, des objets en l'occurrence : un tableau d'éléments du domaine et un tableau d'élements du codomaine avec la contrainte qu'à un indice donné, on trouve dans le tableau des élements du domaine l'élément qui est en relation avec l'élément du codomaine placé à ce même indice dans le second tableau. Nous utilisons donc ces deux tableaux pour mémoriser deux collections de données, à ceci près qu'une contrainte existe entre les deux tableaux permettant de voir les éléments à un même indice comme formant un couple. Nous avons vu à la section 5.3 du chapitre 5 ce genre d'utilisation des tableaux. Parce que le nombre de couples de la relation peut varier dans le temps, au gré des ajouts et retraits de couples, mais que par ailleurs un tableau est créé à une taille fixe, on utilise l'idée développée précédemment qui consiste à avoir une variable, ici `nCouples` contenant le nombre de couples actuellement mémorisés dans les deux tableaux, et une constante de classe `NMAX` qui contient la taille à allouer lors de la création des tableaux. La figure 6.11 illustre de manière visuelle la représentation choisie.

Ce choix de représentation fait, il se traduit par des déclarations de constantes et de variables dans la classe, accompagné d'un commentaire d'implantation, placé au début de son corps :

```
/*
 *  Implantation
 *
 *  La relation binaire est représentée par deux tableaux d'entiers,
 *  domaine et codomaine dans lesquels vont être rangés les couples
```

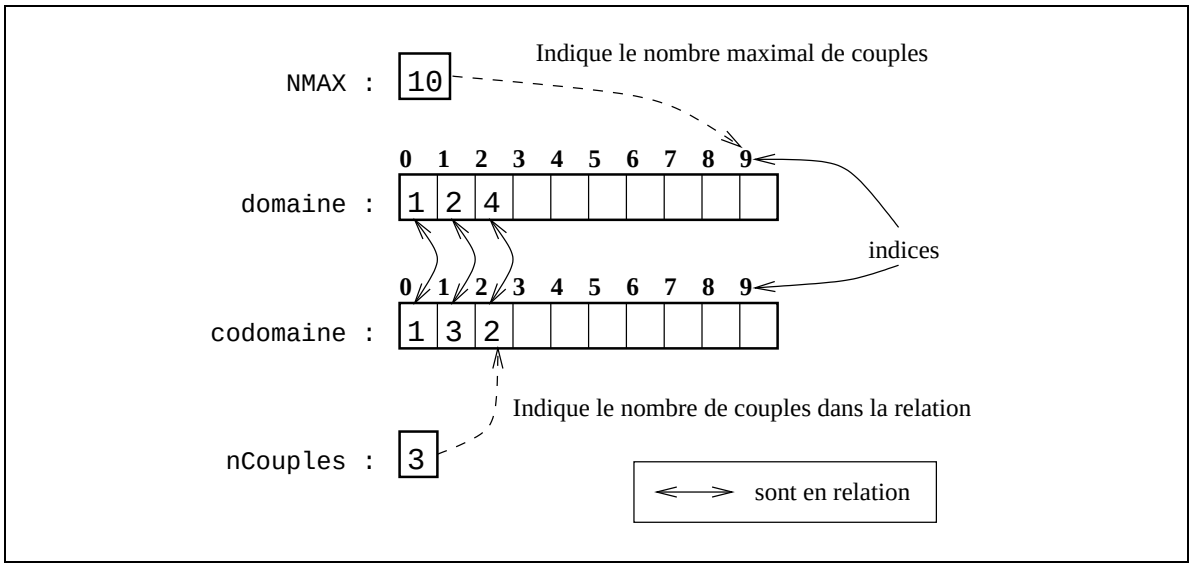


FIG. 6.11 – Représentation pour l'objet relation binaire.

```
 *  (a, b) de la relation.  Un couple est rangé dans les tableaux au
 *  même indice i, la valeur a dans le tableau domaine et b dans le
 *  tableau codomaine.  L'ordre de rangement des couples est sans
 *  importance.
 *  Les méthodes estReflexive, estSymetrique et estTransitive utilisent
 *  le fait que les variables elements et nombreElements de l'objet
 *  ensemble sont visibles directement puisque les deux classes font
 *  partie du même package.
 */

//  --- Constantes partagées par les instances ----------------------
//

/** Nombre maximal de couples dans la relation                      */
private static final int            NMAX = 100 ;

//  --- Variables d'instances ---------------------------------------
//

/** Tableau contenant les éléments du domaine                       */
Object[]            domaine ;
/** Tableau contenant les éléments du codomaine                     */
Object[]            codomaine ;
/** Nombre de couples dans la relation                              */
int                 nCouples ;
/** Ensemble sur lequel la relation est définie                     */
protected Ensemble  ensembleDeDefinition ;
```

La constante `NMAX` est une constante (`final`) de classe (`static`) dont la valeur est arbitrairement fixée à 100. L'intérêt d'utiliser une constante ici est de fixer ce choix à un endroit précis, et s'il doit être modifié lors de l'entretien de la classe, il suffira de modifier la valeur dans la déclaration.

Les tableaux `domaine` et `codomaine` ainsi que la variable `nCouples` sont déclarées comme des variables de visibilité de paquetage (par défaut, en l'absence d'autre qualificateur de visibilité), ce qui permettrait à un objet d'une autre classe du même paquetage de les accéder directement, ce qui peut s'avérer utile pour balayer efficacement tous les couples de la relation.

La variable `ensembleDeDefinition` est déclarée comme protégée, c'est-à-dire visible uniquement de la classe `RelationBinaire` ou de ses sous-classes. L'accès à sa valeur est donné par

la méthode `lEnsembleDeDefinition`, ce qui laisse toute latitude lors d'une refonte ou d'un sous-classage de la classe de modifier la façon de représenter l'ensemble de définition de la relation sans affecter les utilisateurs de la classe (si la méthode `lEnsembleDeDefinition` demeure intacte, bien sûr).

Ce choix de représentation fait, notre méthodologie demande alors d'appliquer les principes de développement des objets autour d'un invariant structurel de manière à capturer les contraintes de représentation. Nous avons déjà noter l'invariant fonctionnel qui précise que l'ensemble de définition doit être non-nul. Le choix de représentation par des tableaux induit un invariant représentationnel qui contraint les tailles des tableaux `domaine` et `codomaine` à être égales et la valeur de la variable `nCouples` à être inférieure ou égale à celle de la constante `NMAX`. Ces invariants sont introduits à la fin du commentaire Javadoc de la classe[6] :

```
 * <STRONG> Invariant structurel </STRONG>
 *
 *                                          // FONCTIONNEL
 * @invariant   lensembleDeDefinition() != null // ensemble bien défini
 *                                          // REPRÉSENTATIONNEL
 * @invariant   domaine.length == codomaine.length
 *                                          // nombres d'éléments domaine
 *                                          // et codomaine égaux
 * @invariant   nCouples <= NMAX            // borne max. observée
```

L'introduction des éléments de représentation demande également de revoir les contrats sur les méthodes de manière à en ternir compte. Cela va au-delà du fait que l'invariant structurel doit être vérifié à la fin de chaque méthode. Les pré- et postconditions des méthodes peuvent en effet prescrire des contraintes sur l'état dans lequel la méthode peut s'exécuter et sur les modifications produites par la méthode. Dans le cas de la classe `RelationBinaire`, cela intéresse la méthode d'initialisation et les modificateurs `ajoute` et `retire`.

Pour la méthode d'initialisation, son rôle va consister à créer les deux tableaux référés par `domaine` et `codomaine`, et initialiser la variable `nCouples` à 0 puisque la relation nouvellement créée est intialement vide. Il faut donc ajouter les éléments suivants à son contrat :

```
    * @post    domaine != null             // domaine défini
    * @post    codomaine != null           // codomaine défini
    * @post    nCouples == 0               // relation vide
```

Pour la méthode `ajoute`, une nouvelle précondition apparaît puisqu'on ne peut ajouter de couple à une relation dont les tableaux sont pleins. Ceci n'est bien sûr valable que si le couple n'est pas déjà en relation. Pour ce qui est des postconditions, de deux choses l'une : ou bien le couple était déjà en relation et alors la variable `nCouples` demeure inchangée, ou sinon, la variable `nCouples` voit sa valeur incrémentée de 1. Ceci nous donne les éléments de contrat suivants :

```
    * @pre     this.sontEnRelation(dom, codom) || nCouples < NMAX
    *                                       // pas plein
    * @post    (this.sontEnRelation(dom, codom)@pre
    *                                          && nCouples == nCouples@pre)
    *          || (!this.sontEnRelation(dom, codom)@pre
    *                                        && nCouples == nCouples@pre + 1)
    *                                       // couple ajouté
```

[6] L'outil *iContract* ne permettant pas de distinguer invariants représentationnel et fonctionnel, les deux apparaissent dans le commentaire Javadoc de la classe. Nous les séparons par un commentaire.

Enfin, pour la méthode `retire`, point n'est besoin de vérifier s'il y a au moins un couple dans la relation (i.e. `nCouples` supérieur à 0) car la précondition que le couple est en relation implique nécessairement si elle est vrai qu'il y a au moins un couple dans la relation. Par contre, de manière similaire à `ajoute`, on peut spécifier que la valeur de `nCouples` décroit de 1 après le retrait du couple :

```
* @post    nCouples == nCouples@pre - 1    // couple retiré
```

6.5.2 Mise en œuvre des méthodes

La mise en œuvre d'une méthode vise à construire, de manière incrémentale, une séquence d'énoncés qui va réaliser le travail attendu de cette méthode. Cette construction se base sur les valeurs des paramètres, qui doivent observer les préconditions de la méthode, et sur l'état de l'objet au moment de la réception du message. À partir de cette base, les énoncés du corps de la méthode vont calculer le résultat attendu de cette méthode et éventuellement apporter les modifications souhaitées à l'état de l'objet. Ce résultat et ces modifications doivent être tels que les postconditions de la méthode seront observées et tels que l'objet sera laissé dans un état vérifiant son invariant structurel.

Tout l'arsenal des énoncés vus au chapitre 4 peut être mis à contribution pour arriver à ces fins. La principale difficulté est ici de décomposer un calcul en énoncés qui, ajoutés les uns aux autres, vont produire le calcul et les effets désirés. Quelques méthodes, comme les accesseurs à des valeurs de variables d'instances, sont très faciles à écrire. C'est le cas par exemple pour la méthode `lEnsembleDeDefinition` de la classe `RelationBinaire`, qui se borne à retourner la valeur de la variable d'instance `ensembleDeDefinition` :

```
public Ensemble     lEnsembleDeDefinition()
{
    return ensembleDeDefinition ;
}
```

La programmation par objets a ceci d'intéressant que le découpage des responsabilités entre les objets fait en sorte que bien souvent les méthodes à écrire sont relativement simple. Dans certains cas cependant, la complexité intrinsèque de la programmation reprend ses droits, et il faut se donner un moyen d'attaquer le travail de manière efficace.

Décomposition descendante

Lorsque la tâche à réaliser par une méthode devient trop complexe pour être programmée en un ou deux énoncés évidents, l'approche la plus connue pour la mettre en œuvre est la décomposition descendante. L'idée générale de la décomposition descendante se résume en deux points :

1. Si le problème de programmation à résoudre est simple et se traduit par quelques énoncés élémentaires, alors produire directement ces énoncés.
2. Sinon, décomposer le problème en une séquence de sous-problèmes plus simples, puis recommencer le travail avec chacun des sous-problèmes.

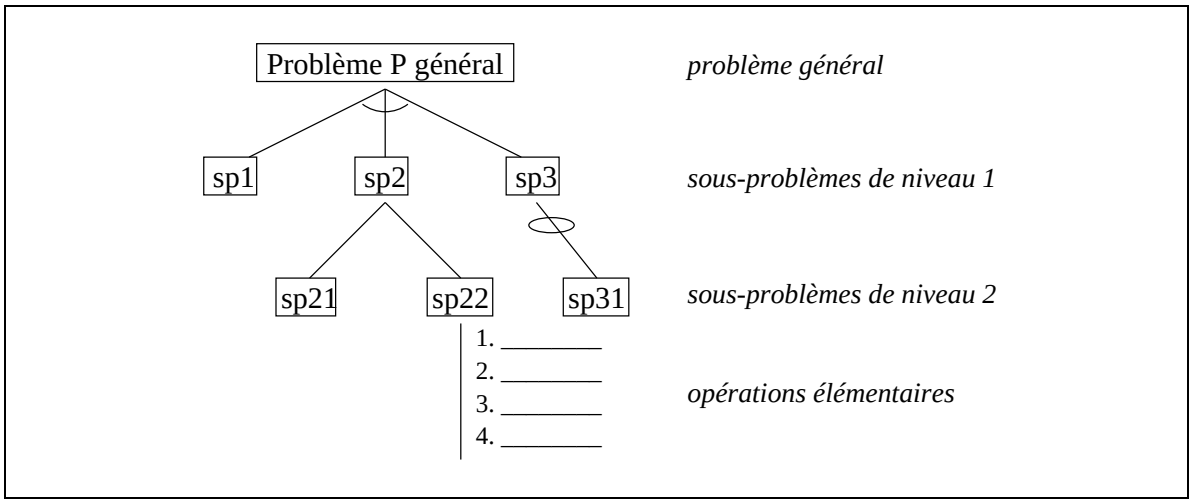


FIG. 6.12 – Illustration de la décomposition descendante.

Le processus de décomposition fait apparaître des sous-problèmes de plus en plus simples jusqu'à ce que ces derniers finissent par être solubles par quelques énoncés élémentaires. Une fois tous les sous-problèmes d'un problème résolus, leurs énoncés sont recomposées séquentiellement (la plupart du temps simplement placés les uns à la suite des autres) pour donner la solution au problème de départ. La figure 6.12 illustre ce processus de décomposition descendante sur un exemple générique volontairement limité à trois niveaux de décomposition. Il va de soi qu'en pratique le nombre de niveaux n'est pas borné à trois.

Plus précisément, la décomposition d'un problème en sous-problèmes peut être de nature séquentielle. Les liens du problème vers ses sous-problèmes sont alors joints par un arc de cercle (voir décomposition du problème P en sous-problèmes sp1, sp2 et sp3 de la figure 6.12). La décompostion peut aussi être de nature alternative, le problème étant résolu par l'un ou l'autre des sous-problèmes. Les liens ne sont alors pas liés par un arc de cercle (comme la décomposition de sp2 en sp21 ou sp22). Enfin, la décomposition peut être de nature itérative, le problème étant résolu par la répétition de la solution au sous-problème. Le lien est alors entouré d'un cercle (comme le passage de sp3 à sp31).

Exemple 55 Considérons le problème du calcul du nombre de jours entre deux dates représentées par leurs numéros de jours, de mois et d'années respectifs. On se rend vite compte que le nombre de cas à prendre en compte rend le problème insoluble directement (les deux dates sont-elles du même mois ? de la même année ? y a-t-il une ou plusieurs années bissextiles entre les deux ? etc.). Si les deux dates sont sur des années différentes, on peut décomposer le problème en un calcul du nombre de jours depuis la première date jusqu'à la fin de son année, du nombre de jours des années intermédiaires, et du nombre de jours depuis le début de son année jusqu'à la seconde date. De même, le nombre de jours dans les années intermédiaires peut se décomposer en la répétition du calcul du nombre de jours dans une année. Enfin, le nombre de jours dans une année peut se décomposer alternativement soit dans le calcul du nombre de jours dans une année normale, soit dans celui du nombre de jours dans une année bissextile. La décomposition obtenue est illustrée à la figure 6.13. □

La décomposition d'un problème en sous-problèmes est une tâche difficile, car toute décomposition ne mène pas nécessairement à une solution effective à un problème donné. C'est un tâche qui demande à la fois intuition et rigueur. L'intuition permet d'identifier les sous-problèmes plus simples qui se cachent derrière un problème complexe. La rigueur est essentielle

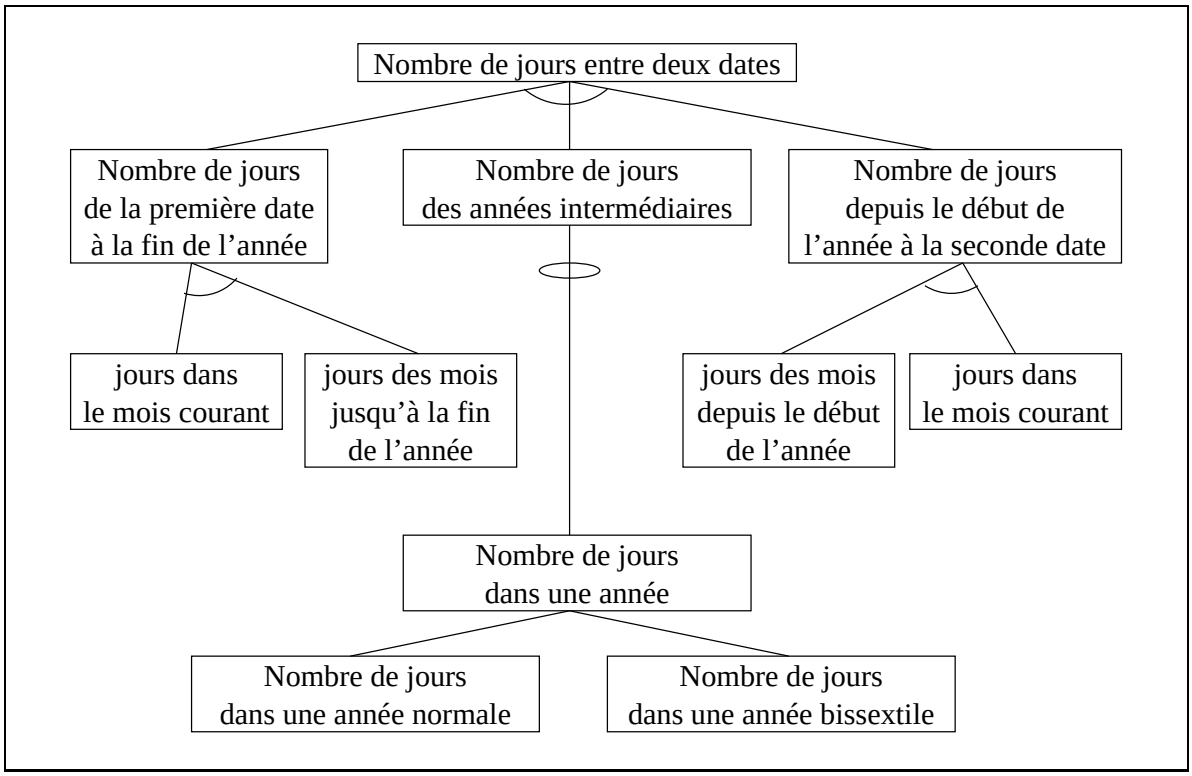


FIG. 6.13 – Décomposition du calcul du nombre de jours entre deux dates.

pour assurer que la composition des résultats des sous-problèmes va vien donner le résultat au problème global. Wirth propose trois questions à se poser à chaque étape de décomposition [Wir76] :

1. La résolution des sous-problèmes implique-t-elles la solution au problème entier ?
2. La succession d'actions [obtenue] a-t-elle un sens ? reflète-t-elle la structure du problème ?
3. Les sous-problèmes sont-ils plus simples ? plus proches des opérations primitives de la machine (du langage) ?

La recomposition des solutions aux sous-problèmes doit donner le résultat du problème global, ce qui demande d'être particulièrement attentif à la fois à l'exactitude des énoncés écrits mais aussi au fait que la composition de leurs résultats donnent bien le résultat global attendu. L'approche axiomatique développée dans les chapitres précédents peut être utilisée conjointement avec la décomposition descendante pour s'assurer que le résultat obtenu est bien celui attendu. Chaque sous-problème peut alors être doté d'une précondition et d'une postcondition, exprimant respectivement ce qui est attendu des étapes précédentes et ce qui est fourni aux étapes subséquentes.

Nous n'allons pas passer en revue chaque méthode de la classe `RelationBinaire`. Considérons simplement la méthode `retire`. L'objectif est de retirer un couple dont les éléments du domaine et du codomaine sont donnés en paramètres à la méthode. Compte tenu de la représentation de la relation sous la forme d'une collection de couples maintenus dans des tableaux «parallèles», un premier niveau de décomposition consiste d'abord à rechercher la valeur d'indice *ind* auquel le couple se trouve dans les tableaux `domaine` et `codomaine`, puis à prendre le dernier couple inscrit dans les tableaux pour le mettre à l'indice trouvé.

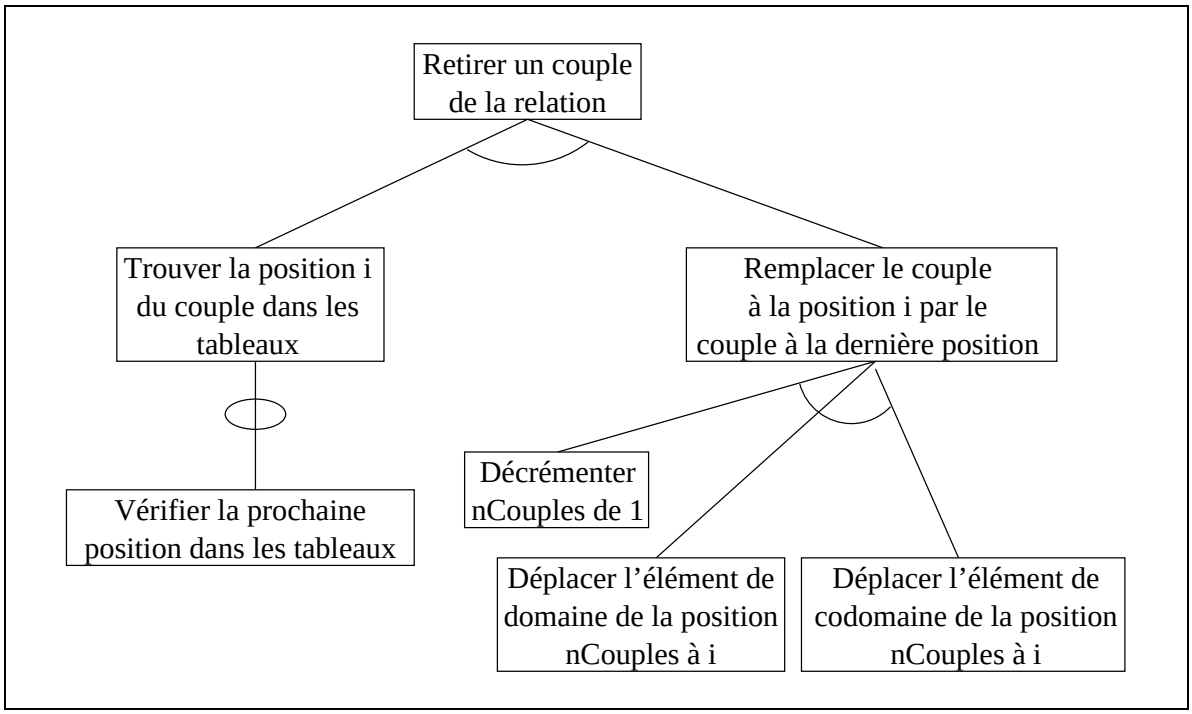


FIG. 6.14 – Décomposition de la méthode `retire`.

Si on considère chacun de ces sous-problèmes, la recherche de l'indice *ind* peut se décomposer en une répétition qui considère l'une après l'autre les positions dans les tableaux tant que la couple cherché n'est pas trouvé. Une assertion suffisante pour que la seconde partie de la méthode s'exécute correctement est :

$\texttt{i} = ind \ \wedge \ 0 \leq ind < \texttt{nCouples} \ \wedge \ \texttt{domaine}[ind] = \texttt{dom} \ \wedge \ \texttt{codomaine}[ind] = \texttt{codom}$

L'existence d'une valeur *ind* vérifiant cette assertion est garantie par le contrat de la méthode qui dit que le couple (`dom`., `codom`) est en relation. Pour le second sous-problème, dans la mesure où il y a au moins un couple dans la relation (ce qui est aussi garanti par le contrat), l'indice où se trouve le dernier couple dans les tableaux est donné par `nCouples - 1`. En décrémentant la valeur de `nCouples`, on fait d'une pierre deux coups : on obtient l'indice du dernier couple dans la relation et on réduit le nombre de couples dans la relation de 1, ce qui est attendu après le retrait.

On peut donc découper ce second sous-problème consistant à ramener le dernier couple à l'indice *ind* en trois sous-problèmes : décrémenter la valeur de `nCouples` pour obtenir l'indice du dernier couple dans la relation, transférer la valeur de domaine du dernier couple de la position `nCouples` à la position *ind*, puis transférer de la même manière la valeur de codomaine du dernier couple. La figure 6.14 illustre ce découpage de manière visuelle.

À partir des éléments de décomposition, on peut produire le code de la méthode :

```
public RelationBinaire      retire(
    Object dom,
    Object codom
    )
{
    int i ;
    boolean pas_trouve = true ;

    for (i = 0 ; i < nCouples && pas_trouve ; i++) {
```

```
            if (domaine[i].equals(dom) && codomaine[i].equals(codom)) {
                pas_trouve = false ;
            }                           // if (domaine[i].equals(dom) && ... )
        }                               // for (i = 0 ; ... )
        nCouples-- ;
        domaine[i-1] = domaine[nCouples] ;
        codomaine[i-1] = codomaine[nCouples] ;

        return this ;
    }
```

On note qu'à la sortie de la répétition, le couple a nécessairement été trouvé puisqu'il appartient à la relation par la contrainte du contrat de la méthode. Ainsi, on sort nécessairement de la répétition avec `pas_trouve` égal à faux. La valeur de `i` est alors la position cherchée plus 1, puisque la boucle `for` va faire l'incrémentation de `i` y compris pour la dernière itération où `pas_trouve` passera à faux. L'assertion précédente n'est donc pas tout à fait vérifiée, mais on peut prouver que l'assertion suivante l'est (voir exercices), or elle est parfaitement équivalente :

$$\texttt{i} = ind + 1 \ \wedge \ 0 \leq ind < \texttt{nCouples} \ \wedge \ \texttt{domaine}[ind] = \texttt{dom} \ \wedge \ \texttt{codomaine}[ind] = \texttt{codom}$$

Dans ce contexte, la position *ind* à laquelle le dernier couple doit être placé se trouve donc à $\texttt{i} - 1 = ind + 1 - 1 = ind$. Ceci explique donc l'utilisation de '`i - 1`' dans le code de la deuxième partie de la méthode.

Abstraction procédurale

Dans la mise en œuvre d'une classe, il n'est pas rare d'introduire une ou plusieurs méthodes servant à découper certaines des méthodes identifiées lors de l'analyse en une ou plusieurs sous-méthodes. Parce qu'elles servent essentiellement d'aide à des méthodes publiques et ne représentent pas des savoir-faire de l'objet, ces méthodes sont généralement déclarées comme privées.

Trois raisons essentielles poussent à l'introduction de telles méthodes privées : respect de la structure de la solution, partage de traitements qui se répètent et limitation de la taille des méthodes. L'introduction de méthodes pour mettre en œuvre les différents niveaux de décomposition obtenus fait en sorte que l'expression de la solution par le code reflète précisément la structure de la décomposition, ce qui est utile pour comprendre le programme et le maintenir. Il arrive que la décomposition de plusieurs méthodes mènent au même sous-problème. Plutôt que de répéter les lignes de code résolvant ce sous-problème à chaque fois, il est plus judicieux de définir une nouvelle méthode, et de l'appeler depuis chaque endroit où le calcul est nécessaire. Ceci dit, comme nous allons le voir ci-bas, avant d'introduire une méthode, il faut s'assurer que ces lignes de code correspondent bien à une opération cohérente du point de vue du contexte. Enfin, il est conseillé de ne pas écrire de méthodes qui soient trop longues (ne pas dépasser 25 lignes est une bonne règle), ce qui ne peut se faire qu'en découpant les méthodes longues en sous-méthodes.

La création d'une méthode implique trois phases :

1. Identification d'une séquence d'énoncés fortement cohérente et faiblement couplée qui forme une opération logique dans le contexte du problème à résoudre.
2. Paramétrisation de la séquence par les éléments les plus susceptibles d'être modifiés d'un appel à l'autre de la nouvelle méthode.
3. Généralisation de la séquence d'énoncés pour ne pas traiter de manière pointue un seul cas de figure pour l'opération concernée, mais bien le cas le plus général, de manière à pouvoir être réutilisée de plusieurs endroits dans la classe.

L'identification d'une séquence d'énoncés correspondant à un traitement susceptible de donner naissance à une procédure est le premier, mais pas le seul élément du passage à l'abstraction procédurale. Cette identification relève en partie de l'intuition. La question à laquelle on doit pouvoir répondre à ce moment est : est-ce que cette séquence d'énoncés correspond à une opération cohérente et en bonne partie indépendante du contexte dans laquelle cette séquence apparaît ? Le plus souvent, la réponse s'impose d'elle-même dans le contexte du problème. Lorsque les choses ne sont pas si claires, une bonne heuristique consiste à vérifier la cohérence et le couplage de la nouvelle méthode.

Une méthode est d'autant plus cohérente qu'elle traite peu de variables et de manière interdépendante. Par exemple, si la nouvelle méthode peut être divisée en plusieurs sous-parties qui travaillent de manière indépendantes les unes des autres sur des sous-ensembles de variables disjoints, alors la cohérence est faible et le découpage insatisfaisant. La cohérence est essentiellement une mesure de qualité interne à la nouvelle méthode.

Une méthode est faiblement couplée si elle partage très peu d'informations avec son contexte d'appel. Typiquement, deux questions servent à déterminer le couplage d'une méthode : nécessite-t-elle un grand nombre de paramètres ? et nécessite-t-elle d'avoir accès à plusieurs variables également accessibles depuis le site de son appel. Une méthode qui nécessite moins de trois paramètres, dont le corps utilise aucune variable extérieure à sa définition (uniquement les paramètres et variables locales) et qui retourne un résultat par un énoncé `return` est un bon exemple d'une méthode faiblement couplée.

À titre d'exemple, la séquence permettant de trouver l'indice auquel se trouve un couple dans une relation binaire apparaît de manière pratiquement identique dans deux méthodes de la classe `RelationBinaire` : `retire` et `sontEnRelation` (voir annexe B). Cette séquence correspond bien à une opération logique dans le contexte, puisqu'il s'agit de trouver la position d'un couple dans les tableaux `domaine` et `codomaine`. Cette séquence est donc bonne candidate pour la création d'une méthode `trouvePosition`. Cette opération nécessite la connaissance des valeurs de domaine et codomaine du couple recherché, et son résultat peut être de deux formes : soit la valeur d'indice cherchée si `pas_trouve` est égal à faux, et sinon `pas_trouve` égal à vrai si le couple n'appartient pas à la relation.

Vu de la méthode `retire`, le résultat de `trouvePosition` peut être une valeur d'indice, sachant que le contexte assure que le couple sera trouvé. Vu de `sontEnRelation`, il est possible que le couple ne soit pas trouvé. Cette observation nous incline à vouloir retourner deux résultat : la valeur d'indice et la valeur donné à `pas_trouve`. Pourtant, la généralisation entre les deux sites d'utilisation nous poussent à simplifier le couplage entre le site appelant et la méthode appelée.

Une technique simple permet de se passer de ce deuxième résultat. Si la méthode ne trouve pas le couple, on peut utiliser une valeur d'indice impossible pour encoder le fait que la méthode n'a pas trouvé le couple. Un indice négatif, par exemple -1, est impossible. On peut donc écrire la méthode `trouvePosition` de telle façon qu'elle retourne une valeur d'indice se situant entre 0 et la valeur de `nCouples` (exclusivement) si le couple est dans la relation et -1 sinon.

L'écriture de la méthode `trouvePosition` et la modification corollaire des méthodes `retire` et `sontEnRelation` sont laissées en exercice.

```
Test de la classe 'RelationBinaire'

    Séquence n.1 : construction et affichage

{}
Test OK

    Séquence n.2 : ajoute

{(1,5)}
Test OK

    Séquence n.3 : retire

{(1,5), (4,3)}
Test OK

    Séquence n.4 : sontEnRelation

1 est en relation avec 5 dans {(1,5), (4,3)} ?  true
3 est en relation avec 4 dans {(1,5), (4,3)} ?  false
1 est en relation avec 5 dans {} ?  false
Test OK

    Séquence n.5 : estReflexive

La relation {(1,5), (2,2), (4,3)} est-elle réflexive ?  false
La relation {(1,5), (1,1), (2,2), (4,3), (3,3), (4,4), (5,5)} est-elle réflexive ?  true
Test OK

    Séquence n.6 : estSymetrique

La relation {(1,5), (2,2), (4,3)} est-elle symétrique ?  false
La relation {(1,5), (5,1), (4,3), (3,4)} est-elle symétrique ?  true
Test OK

    Séquence n.7 : estTransitive

La relation {(1,5), (2,2), (5,3)} est-elle transitive ? false
La relation {(1,5), (2,2), (5,3), (1,3)} est-elle transitive ?  true
Test OK

    Séquence n.8 : estEquivalence

La relation {(1,5), (2,2), (5,3), (1,3)} est-elle une relation d'équivalence ?  false
La relation {(1,5), (2,2), (5,3), (1,3), (5,1), (3,5), (3,1), (1,1), (3,3), (4,4), (5,5)}
 est-elle une relation d'équivalence ?  true
Test OK

Fin du test de 'RelationBinaire'
```

FIG. 6.15 – Rapport de la classe de test `RelationBinaireT`.

6.6 Vérification

À ce point de la méthode de développement, il faut encore vérifier que la classe produite correspond bien aux attentes. Dans un premier temps, on vérifie que le code est correcte en exécutant la classe de test préparée à cet effet. Dans un second temps, on peut aussi vérifier que les contrats sont aussi corrects de manière à ce qu'ils captent effectivement le plus possible des erreurs et ne produisant pas de faux témoins, c'est-à-dire qu'ils ne se déclenchent pas sur des cas corrects.

6.6.1 Vérification par les tests

L'intérêt d'avoir préparé la séquence de tests avant la mise en œuvre de la classe prend ici tout son sens, dans la mesure où il suffit de compiler les deux classes et d'exécuter la classe de test pour vérifier le code de la classe. Rappelons cependant que la classe de test a été écrite avant tous les choix de mise en œuvre. Il peut donc devenir nécessaire de modifier la classe de test pour ajouter des cas de tests vérifiant des alternatives et des cas limites introduits par ces choix et le code correspondant.

L'exécution de la classe de test `RelationBinaireT` donne le rapport reproduit à la figure 6.15. Ce code avait été vérifié de longue date, donc le rapport reproduit ici ne vise qu'à illustrer un cas positif. Normalement, à cette étape, des erreurs peuvent apparaître, surtout si le code de la classe est complexe et si on a omis (pour des raisons pratiques) de développer complètement toutes les preuves axiomatiques. Par ailleurs, on note que la classe `RelationBinaire` utilisant la classe `Ensemble`, son test dépend intimement de cette dernière. Assez souvent, c'est lors de l'utilisation conjointe des classes que des erreurs apparaissent aux frontières.

Lorsque des erreurs sont mises en évidence par les tests, il faut bien sûr les corriger. La correction d'erreurs demande dans un premier temps d'identifier l'endroit où l'erreur apparaît dans le programme, puis d'identifier les causes de l'erreur. Une fois identifiées, les causes d'un résultat incorrect sont résolues par modification du programme, ce qui revient donc à de la programmation. A priori, c'est par inspection du programme que l'on peut arriver à identifier lieu et causes d'un résultat incorrect. Plusieurs des premières erreurs d'exécution sont trouvées comme cela, mais ce n'est cependant pas toujours évident (si ça l'était, il est probable que l'erreur n'aurait pas été commise). L'aide d'une trace du programme ou du raisonnement axiomatique, au moins partiel, s'avère ici très important.

Lorsque l'inspection du programme ne permet pas d'avancer, une pratique courante est alors d'utiliser l'ordinateur lui-même pour aider à trouver l'erreur. Le support d'un environnement de programmation sophistiqué, dont des outils d'aide appelés dévermineurs (littéral, de «*debugger*» en anglais). Ces outils permettent d'exécuter un programme pas à pas, énoncé par énoncé, et d'examiner l'état des variables à chaque étape. Ils servent donc essentiellement à localiser l'erreur, puis à comprendre ce qui ne tourne pas rond dans le programme.

En l'absence de tels outils, la technique simple consiste à tracer les programmes, c'est-à-dire faire imprimer à l'écran les valeurs de variables, par des énoncés d'impression ajoutés ajoutés au programme à cette fin dans les points d'observation jugés importants. C'est un procédé un peu fastidieux, mais qui donne généralement de bons résultats.

6.6.2 Vérification des contrats et des tests

Lorsque le code du programme a passé tous les tests et qu'il a donné satisfaction, on peut encore pousser plus loin la méthodologie de développement pour s'assurer que les contrats posés sur la classe et ses méthodes sont corrects. En effet, aussi intéressante qu'elle soit, l'approche contractuelle ne garantit pas que les contrats, écrits par des humains, sont plus exempts d'erreurs que le reste du programme. D'ailleurs, des utilisateurs courants de l'approche contractuelle évaluent eux-mêmes à égalité le nombre de déclenchements de contrats (détection de cas anormaux) dus à des erreurs de programme par rapport au nombre qui sont dus à des erreurs dans le contrat lui-même.

La difficulté de la vérification des contrats est que les cas de test devant déclencher la réaction du contrat entraînent généralement l'arrêt brutal du programme, puisque violation

de contrat engendre erreur. Il n'est donc pas possible[7] de construire et d'exécuter simplement une séquence de test qui vérifie que les contrats se déclenchent bien lorsqu'il le faut puisqu'à chaque déclenchement le programme s'arrête. Il faut donc faire chaque test individuellement, ce qui consomme passsablement de ressources.

D'autre part, pour vérifier les contrats, il faut vérifier les cas positifs et les cas négatifs. On peut considérer que les cas positifs, c'est-à-dire où les valeurs soumises au contrat vérifient bien la condition exprimée, ont déjà été vérifiés par les tests de la classe. Pour les cas négatifs, pour vérifier les préconditions aux méthodes, il faut leur soumettre des valeurs considérées comme mauvaises par leur contrat. Pour vérifier l'invariant structurel, il faut faire subir à l'objet des opérations qui vont volontairement le violer. Enfin, pour vérifier les postconditions des méthodes, il faut injecter volontairement des erreurs dans le corps des méthodes pour voir si le résultat produit déclenche bien la postcondition.

Donc, vérifier les contrats suppose d'introduire volontairement des erreurs dans les programmes, ce qui devient rapidement extrêmement coûteux en temps dans un processus (le test) qui en est lui même déjà gourmand. Des recherches sont actuellement menées pour produire automatiquement ce que l'on appelle des analyses de mutations, c'est-à-dire de prendre un programme et de lui faire subir des mutations «aléatoires» mais syntaxiquement correcte, et voir si les erreurs ainsi introduites sont détectées par les postconditions des méthodes et l'invariant structurel de l'objet. Le même genre de techniques de mutation est utilisé pour vérifier les tests, c'est-à-dire voir si la séquence de tests préparée met bien en évidence les erreurs qui sont introduites.[8].

Lorsque le code, les contrats et les tests eux-mêmes ont été systématiquement vérifiés, un bon degré de confiance peut être accordé à la classe ainsi obtenue.

6.7 Exercices

6.7.1. Utilisez l'approche axiomatique pour démontrer que l'assertion suivante :
$\texttt{i} = ind + 1 \ \wedge \ 0 \leq ind < \texttt{nCouples} \ \wedge \ \texttt{domaine}[ind] = \texttt{dom} \ \wedge \ \texttt{codomaine}[ind] = \texttt{codom}$
est bien vérifiée à la sortie de l'itération de la méthode `retire` de la classe `RelationBinaire`.

6.7.2. Écrire la méthode `trouvePosition` qui retourne l'indice d'un couple dans les tableaux représentation la relation binaire ou -1 si le couple n'y est pas. Modifier ensuite les méthodes `retire` et `sontEnRelation` pour utiliser cette méthode.

6.7.3. Considérez la classe `RelationBinaire` complète qui vous est donnée à l'annexe B et à partir du code des méthodes, rétroconcevez les schémas de décomposition descendante qui ont pu mener à ce code.

6.7.4. Un autre choix de représentation pour la classe `RelationBinaire` consiste à mémoriser les couples par une matrice relationnelle. À chaque élément de l'ensemble de définition est associé un indice de 0 à $n-1$. La matrice relationnelle de taille $n \times n$ est une matrice booléenne M contenant vrai en position (i, j) si l'élément de l'ensemble de définition correspondant à l'indice i est en relation avec celui d'indice j et faux sinon. Faire une deuxième mise en oeuvre complète de la classe `RelationBinaire` en utilisant ce choix de représentation. Quels sont les avantages et inconvénients comparatifs de ces deux choix de représentation ? (long)

[7] Modulo une utilisation extensive des exceptions et de leur capture, ce dont nous ne parlerons pas ici.

[8] Vérifier les tests a déjà quelque chose de surprenant, mais rien n'empêche ensuite de songer à vérifier la vérification des tests, et ainsi de suite `ad infinitum`

Chapitre 7

Entrées/sorties et interface graphique

7.1 Élément de savoir-faire :

7.2 Exercices

Annexe A

La classe Ensemble

```
//      Ensemble.java             implante des ensembles d'entiers simples

package calculEnsembliste ;

/**
 * <STRONG> Description </STRONG>
 *
 *      Implante des ensembles d'objets simples.
 *
 * <STRONG> Principes et fonctionnalités </STRONG>
 *
 *      Chaque élément ne peut apparaître qu'au plus une fois dans
 *      un ensemble.
 *
 * <STRONG> Date </STRONG>
 *
 *      5 janvier 1999
 *
 * <STRONG> Liste des mises à jour </STRONG>
 *
 *<table BORDER=2 COLS=2 WIDTH="100%" NOSAVE >
 *<tr>
 *<td> néant           </td><td>                                          </td>
 *</tr>
 *</table>
 *
 * <STRONG> Invariant structurel </STRONG>
 *
 * @invariant   nombreElements >= 0 && nombreElements <= MAX_ELEMENTS //
 * @invariant   elementsApparaissentUneFois()                         //
 *
 * @author      Jacques Malenfant
 */
public class    Ensemble
{
    /*
     * L'ensemble est représenté par un tableau.
     *
     */

    //  --- Constantes partagées par les instances ----------------------
    //

    /** Nombre maximal d'éléments dans un ensemble                */
    public static final int            MAX_ELEMENTS = 100 ;

    //  --- Variables d'instances ---------------------------------------
    //
```

```
/** Variable d'instance : le tableau des éléments             */
protected Object[]          elements ;
/** Variable d'instance : le nombre d'éléments dans l'ensemble */
protected int               nombreElements ;

//  --- Méthodes privées ----------------------------------------
//

// ------------------------------------------------
/**
 * calculeNbOccurences :    nombre d'occurence d'un élément dans elements
 *
 * @param   élément à considérer
 * @return  nombre d'occurence d'element dans elements
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     true
 * @post    return >= 0 && return <= nombreElements
 */
protected int       calculeNbOccurences(
    Object element
    )
{
    /* Balaie le tableau et compte le nombre d'occurence de l'élément
     */
    //    Variables locales
    int ret = 0 ;
    int i;

    //    Code de la méthode
    for ( i = 0 ; i < nombreElements ; i++ ) {
       if ( element.equals(elements[i]) ) {
           ret++ ;
       }                            // if ( element == elements[i] )
    }                               // for (i = 0 ; ...)

    return ret ;
}
// ------------------------------- calculeNbOccurences

// ------------------------------------------------
/**
 * vrai si tout élément apparaît une fois
 *
 *    Utilisé dans l'invariant
 *
 * @return  vrai si chaque élément apparaît une seule fois
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     true
 * @post    true
 */
protected boolean   elementsApparaissentUneFois()
{
    /*
     * Balaie le tableau et compte le nombre d'occurence de l'élément
     */
    //    Variables locales
    boolean ret = true ;
    int i;

    for ( i = 0 ; ret && i < nombreElements ; i++ ) {
        ret = ret && calculeNbOccurences(elements[i]) == 1 ;
    }                       // for ( i = 0 ; i < nombreElements ; i++ )

    return ret ;
```

```
}
// ------------------------ elementsApparaissentUneFois


//  --- Méthodes publiques ----------------------------------------
//

//  Les méthodes d'initialisation
//

// ---------------------------------------------------
/**
 * initialisation d'ensemble à vide
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     true
 * @post    this.estVide()  // ensemble vide créé
 */
public              Ensemble()
{
    /*
     * Crée le tableau de taille MAX_ELEMENTS et met le nombre
     * d'éléments à 0
     */
    elements = new Object[MAX_ELEMENTS] ;
    nombreElements = 0 ;
}
// ------------------------------------------- Ensemble

// ---------------------------------------------------
/**
 * initialisation d'ensemble à partir d'un tableau
 *
 *  Initialise l'ensemble à vide puis ajoute les éléments donnés
 *  dans l'argument.
 *
 * @param   tableau des éléments initiaux
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     elems != null && elems.length <= MAX_ELEMENTS
 *                                          // taille limitée
 * @post    forall Object e in new VisiteurTableau(elems, elems.length) |
 *              this.appartient(e)          // éléments dans l'ensemble
 */
public              Ensemble(
    Object[] elems
    )
{
    /*
     * Crée le tableau de taille MAX_ELEMENTS et met les éléments
     * initiaux dedans.
     */

    //    Variables locales
    int i;

    //    Code de la méthode
    elements = new Object[MAX_ELEMENTS] ;
    for ( i = 0 ; i < elems.length ; i++ ) {
        ajoute(elems[i]) ;
    }                                       // for (i = 0 ; ...)
}
// ------------------------------------------- Ensemble

// ---------------------------------------------------
/**
```

```
 * initialisation d'ensemble à partir d'un autre ensemble
 *
 *    Copie un à un les éléments donnés par l'argument.
 *
 * @param   autre ensemble à copier
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     autreEns != null
 * @post    this.equals(autreEns)
 */
public              Ensemble (
    Ensemble autreEns
    )
{
    /* Crée le tableau de taille MAX_ELEMENTS et met les éléments
     * de l'autre ensemble dedans.
     */
    //    Variables locales
    int i;

    //    Code de la méthode
    elements = new Object[MAX_ELEMENTS] ;
    nombreElements = autreEns.nombreElements ;
    for ( i = 0 ; i < nombreElements ; i++ ) {
        elements[i] = autreEns.elements[i] ;
    }                                       // for (i = 0 ; ...)

}
// ------------------------------------------ Ensemble

//  Les modificateurs
//

// ---------------------------------------------------
/**
 * ajoute un élément dans l'ensemble
 *
 *    Ajoute l'argument dans le receveur.  Si l'argument n'appartient pas
 *    à l'ensemble, il y est ajouté, sinon le receveur est inchangé.  Le
 *    receveur est retourné en résultat pour permettre les cascades.
 *
 * @param   élément à ajouter
 * @return  retourne le receveur
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     element != null
 * @pre     !this.appartient(element) implies
 *              this.nombreElements < MAX_ELEMENTS
 *                                          // si ajout pas plein
 * @post    this.appartient(element)        // élément dans l'ensemble
 * @post    !this.appartient(element)@pre implies
 *              (this.nombreElements - 1) == this.nombreElements@pre
 * @post    this.appartient(element)@pre implies
 *              this.nombreElements == this.nombreElements@pre
 *                                          // nombreElements ajusté
 */
public Ensemble     ajoute(
    Object element
    )
{
    /*
     * ajoute l'élément dans la première position libre du tableau
     * à condition de ne pas y apparaître déjà.
     */

    //    Code de la méthode
```

```
    if ( !appartient(element) ) {
        elements[nombreElements] = element ;
        nombreElements++ ;
    }                               // if ( !appartient(element) )

    return this ;
}
// ---------------------------------------------- ajoute

// ---------------------------------------------------
/**
 * retire un élément de l'ensemble
 *
 *    Retire l'argument de l'ensemble.  L'argument doit appartenir à
 *    l'ensemble, et alors l'ensemble contient un élément de moins
 *    à la fin de la méthode.  Le receveur est retourné en résultat
 *    pour permettre les cascades.
 *
 * @param   élément à retirer
 * @return  retourne le receveur
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     element != null && this.appartient(element)
 * @post    !this.appartient(element)       // élément plus dans l'ensemble
 * @post    (this.nombreElements + 1) ==
 *                  this.nombreElements@pre // nombreElements décrémenté
 */
public Ensemble     retire(
    Object element
    )
{
    /*
     * Retire l'élément du tableau à condition d'y apparaître déjà.
     * Tous les éléments suivants doivent être décalés d'une case
     * vers la gauche après le retrait.
     */
    //    Variables locales
    int     i;

    //    Code de la méthode
    i = 0 ;
    while ( i < nombreElements && !element.equals(elements[i]) ) {
        i++ ;
    }                       //  while ( element != elements[i] && ... )
    nombreElements-- ;
    elements[i] = elements[nombreElements] ;

    return this ;
}
// ---------------------------------------------- retire

// Les accesseurs
//

// ---------------------------------------------------
/**
 * appartient :     appartenance d'un élément à l'ensemble
 *
 *    Parcourt l'ensemble et retourne vrai si l'argument apparaît
 *    dans l'ensemble.
 *
 * @param   élément à vérifier
 * @return  true si l'élément appartient à l'ensemble
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     element != null
```

```
     * @post    return == (exists Object e in
     *                          new VisiteurTableau(elements, nombreElements) |
     *                       e.equals(element))
     */
    public boolean      appartient(
        Object element
        )
    {
        /*
         *   Implantation
         *
         * recherche l'élément dans le tableau d'éléments et retourne
         * vrai si l'élément est trouvé et faux sinon
         */
        //    Variables locales
        int     i;
        boolean ret = false ;

        //    Code de la méthode
        for ( i = 0 ; !ret && i < nombreElements ; i++ ) {
            ret = element.equals(elements[i]) ;
        }                                       // for (i = 0 ; ...)

        return ret ;
    }
    // ---------------------------------------- appartient

    // ---------------------------------------------------
    /**
     * vérifie l'appartenance des éléments à l'ensemble
     *
     * @param   éléments à vérifier
     * @return  true si tous les éléments appartiennent à l'ensemble
     *
     * <STRONG> Contrat </STRONG>
     *
     * @pre     elems != null
     * @post    return == (forall Object e in
     *                                 new VisiteurTableau(elems, elems.length) |
     *                           this.appartient(e))
     */
    public boolean      appartiennent(
        Object[] elems
        )
    {
        /* Implantation
         *
         * Appelle appartient pour chaque élément à vérifier
         */
        //    Variables locales
        int     i;
        boolean ret = true ;

        //    Code de la méthode
        for ( i = 0 ; ret && i < elems.length ; i++ ) {
            ret = (ret && appartient(elems[i])) ;
        }                       // for (i = 0 ; ret && i < elems.length ; i++)

        return ret ;
    }
    // ------------------------------------- appartiennent

    // ---------------------------------------------------
    /**
     * vérifie si le receveur est sous-ensemble de l'argument
     *
     *    Retourne vrai si tous les éléments du receveur appartiennent à
     *    l'argument.
```

```
 *
 * @param   ensemble qui doit contenir le receveur
 * @return  true si l'argument contient le receveur
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     autreEns != null
 * @post    return == (forall Object e in
 *                          new VisiteurTableau(elements, nombreElements) |
 *                       autreEns.appartient(e))
 */
public boolean      estSousEnsembleDe(
    Ensemble autreEns
    )
{
    /*
     * Le receveur est sous-ensemble de l'argument si tous ses éléments
     * appartiennent à l'agument.
     */
    //    Variables locales
    int     i ;
    boolean ret ;

    //    Code de la méthode
    ret = true ;
    for ( i = 0 ; ret && i < nombreElements ; i++ ) {
        ret = ret && autreEns.appartient(elements[i]) ;
    }                       // for (i = 0 ; i < nombreElements ; i++)

    return (ret) ;
}
// ---------------------------------- estSousEnsembleDe

// ----------------------------------------------------
/**
 * réalise l'union de deux ensembles
 *
 *    Retourne un nouvel objet Ensemble qui fait l'union du receveur
 *    et de l'argument.
 *
 * @param   ensemble avec lequel on doit faire l'union
 * @return  union du paramètre et du receveur
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     autreEns != null
 * @post    forall Object e in new VisiteurTableau(return.elements,
 *                                                 return.nombreElements) |
 *              this.appartient(e) || autreEns.appartient(e)
 */
public Ensemble     union (
    Ensemble autreEns
    )
{
    /* Crée une copie de autreEnsemble puis ajoute un à un les éléments
     * du receveur.
     */

    //    Variables locales
    Ensemble        ret ;
    int             i ;

    //    Code de la méthode
    ret = new Ensemble(autreEns) ;
    for ( i = 0 ; i < nombreElements ; i++ ) {
        ret.ajoute(elements[i]) ;
    }                                   // for (i = 0 ; ...)
```

```
    return ret ;
}
// ---------------------------------------------- union

// ---------------------------------------------------
/**
 * réalise l'intersection de deux ensembles
 *
 *    Retourne un nouvel objet Ensemble qui fait l'intersection
 *    du receveur et de l'argument.
 *
 * @param   second argument de l'intersection
 * @return  intersection du paramètre et du receveur
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     autreEns != null
 * @post    forall Object e in new VisiteurTableau(return.elements,
 *                                                 return.nombreElements) |
 *              this.appartient(e) && autreEns.appartient(e)
 */
public Ensemble     intersection (
    Ensemble autreEns
    )
{
    //    Variables locales
    Ensemble        ret ;
    int             i ;

    //    Code de la méthode
    ret = new Ensemble() ;

    for ( i = 0 ; i < nombreElements ; i++ ) {
        if ( autreEns.appartient(elements[i]) ) {
            ret.ajoute(elements[i]) ;
        }                   // if ( autreEns.appartient(elements(i)) )
    }                       // for (i = 0 ; i < nombreElements ; i++)

    return ret ;
}
// ------------------------------------- intersection

// ---------------------------------------------------
/**
 * soustraction ensembliste this \ autreEns
 *
 *    Retourne un nouvel objet Ensemble qui est le résultat de
 *    la soustraction ensembliste de l'argument du receveur, i.e.
 *    receveur - argument.
 *
 * @param   ensemble à soustraire
 * @return  receveur moins argument
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     autreEns != null
 * @post    forall Object e in new VisiteurTableau(return.elements,
 *                                                 return.nombreElements) |
 *              this.appartient(e) && !autreEns.appartient(e)
 */
public Ensemble             moins(
    Ensemble autreEns
    )
{
    /*
     * Le résultat contient tous les éléments du receveur sauf
     * ceux qui sont dans l'argument
     */
```

```
    //    Variables locales
    Ensemble        ret ;
    int             i ;

    //    Code de la méthode
    ret = new Ensemble() ;
    for ( i = 0 ; i < nombreElements ; i++ ) {
        if ( !autreEns.appartient(elements[i]) ) {
            ret.ajoute(elements[i]) ;
        }                   // if ( !autreEns.appartient(elements[i]) )
    }                       // for ( i = 0 ; i < nombreElements ; i++ )

    return ret ;
}
// ------------------------------------------- moins

// ---------------------------------------------------
/**
 * test l'égalité de deux ensembles
 *
 *    Retourne vrai si les deux ensembles contiennnent les mêmes
 *    éléments.
 *
 * @param   ensemble à comparer
 * @return  true si les deux ensembles sont égaux
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     autreEns != null
 * @post    return == ((forall Object e in
 *                            new VisiteurTableau(this.elements,
 *                                                this.nombreElements) |
 *                        autreEns.appartient(e))
 *                    &&
 *                    (forall Object e in
 *                            new VisiteurTableau(autreEns.elements,
 *                                                autreEns.nombreElements) |
 *                        this.appartient(e)))
 */
public boolean      equals(
    Ensemble autreEns
    )
{
    /*
     * Deux ensembles sont égaux qs'ils sont sous-ensembles l'un
     * de l'autre.
     */

    return estSousEnsembleDe(autreEns)
                && autreEns.estSousEnsembleDe(this) ;
}
// ------------------------------------------- equals

// ---------------------------------------------------
/**
 * vérifier si l'ensemble est plein
 *
 *    Retourne vrai si le receveur représente l'ensemble vide.
 *
 * @return  true si l'ensemble ne peut contenir plus d'éléments
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     true
 * @post    true
 */
public boolean      estPlein()
```

```
{
    /* Un ensemble plein contient MAX_ELEMENTS éléments
     */

    return nombreElements == MAX_ELEMENTS ;
}
// ------------------------------------------ estPlein

// --------------------------------------------------
/**
 * estVide() :              vérifier si l'ensemble est vide
 *
 *    Retourne vrai si le receveur représente l'ensemble vide.
 *
 * @return  true si l'ensemble ne contient aucun élément
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     true
 * @post    return == !(exists Object e in
 *                                new VisiteurTableau(this.elements,
 *                                                    this.nombreElements) |
 *                           this.appartient(e))
 */
public boolean              estVide()
{
    /*
     * Un ensemble vide contient 0 éléments
     */

    return nombreElements == 0 ;
}
// ------------------------------------------- estVide

// --------------------------------------------------
/**
 * toString :       chaîne présentant les éléments de l'ensemble
 *
 *  Retourne une chaîne présentant le contenu de l'ensemble sous la
 *  forme suivante :
 *  <OL>
 *  <LI> ensemble vide : {}
 *  <LI> ensemble contenant les éléments 1,2, et 3 : {1,2,3}
 *  <OL>
 *  L'ordre dans lequel les éléments apparaissent n'a pas d'importance,
 *  i.e. dans le second exemple, on pourrait tout aussi bien obtenir
 *  {3,1,2}.
 *
 * @return  chaine représentant le contenu de l'ensemble
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     true
 * @post    true
 */
public String               toString()
{
    /*
     * La principale difficulté est le traitement du dernier élément
     * qui ne doit pas être suivi d'une virgule.
     */

    //    Variables locales
    int     i ;
    String  ret ;

    //    Code de la méthode
    ret = "{" ;
```

```
        i = 0 ;
        while ( i < (nombreElements - 1) ) {
            ret = ret + elements[i] + "," ;
            i++ ;
        }                                   // while ( i < nombreElements - 1 )
        if ( i == (nombreElements - 1) ) {
            ret = ret + elements[i] ;
        }                                   //  if ( i < nombreElements )
        ret = ret + "}" ;

        return ret ;
    }
    // ------------------------------------------ toString

}
// ------------------------------------------------------------ Classe Ensemble
```

Annexe B

La classe `RelationBinaire`

```
//      RelationBinaire.java -- relations binaires sur ensembles d'objets

package calculEnsembliste ;

/**
 * <STRONG> Description </STRONG>
 *
 *      La classe RelationBinaire permet de représenter une relation binaire
 *      sur un certain ensemble, lequel est représenté par un objet instance
 *      de la classe Ensemble du même package.
 *
 * <STRONG> Principes et fonctionnalités </STRONG>
 *
 *      Une relation binaire R sur un ensemble E est un sous-ensemble S du
 *      produit cartésien E x E de l'ensemble E avec lui-même tel que si le
 *      couple (a, b) appartient à S, alors on dit que a est en relation R
 *      avec b, noté a R b.  La classe RelationBinaire permet de créer des
 *      objets représentant de telles relations qui sont construites sur un
 *      objet e représentant l'ensemble E sur lequelle elle est définie, puis
 *      par ajout et retrait de couples (a, b) à la relation.  L'objet
 *      représentant une relation binaire R sait répondre à des messages pour
 *      savoir si deux éléments de l'ensemble E sont en relation par R.  Il
 *      sait également répondre à des messages demandant si la relation R
 *      est réflexive, symétrique, transitive, ou d'équivalence.
 *
 * <STRONG> Date </STRONG>
 *
 *      7 septembre 2000
 *
 * <STRONG> Liste des mises à jour </STRONG>
 *
 *<table BORDER=2 COLS=2 WIDTH="100%" NOSAVE ><tr>
 *<td> néant           </td><td>                                          </td>
 *</tr></table>
 *
 * <STRONG> Invariant structurel </STRONG>
 *
 *                                              // FONCTIONNEL
 * @invariant   lensembleDeDefinition() != null // ensemble bien défini
 *                                              // REPRÉSENTATIONNEL
 * @invariant   domaine.length == codomaine.length
 *                                              // nombres d'éléments domaine
 *                                              // et codomaine égaux
 * @invariant   nCouples <= NMAX                // borne max. observée
 *
 * @author      Jacques Malenfant
 */
public class RelationBinaire
{
```

```
/*
 *  Implantation
 *
 *  La relation binaire est représentée par deux tableaux d'entiers,
 *  domaine et codomaine dans lesquels vont être rangés les couples
 *  (a, b) de la relation.  Un couple est rangé dans les tableaux au
 *  même indice i, la valeur a dans le tableau domaine et b dans le
 *  tableau codomaine.  L'ordre de rangement des couples est sans
 *  importance.
 *  Les méthodes estReflexive, estSymetrique et estTransitive utilisent
 *  le fait que les variables elements et nombreElements de l'objet
 *  ensemble sont visibles directement puisque les deux classes font
 *  partie du même package.
 */

//  --- Constantes partagées par les instances ----------------------
//

/** Nombre maximal de couples dans la relation                      */
private static final int            NMAX = 100 ;

//  --- Variables d'instances ---------------------------------------
//

/** Tableau contenant les éléments du domaine                       */
Object[]            domaine ;
/** Tableau contenant les éléments du codomaine                     */
Object[]            codomaine ;
/** Nombre de couples dans la relation                              */
int                 nCouples ;
/** Ensemble sur lequel la relation est définie                     */
protected Ensemble  ensembleDeDefinition ;

//  --- Méthodes publiques ------------------------------------------
//

//  Les méthodes d'initialisation
//

// ---------------------------------------------------
/**
 * initialise la relation à vide
 *
 *    Une fois initialisée, l'ensemble de définition de la relation doit
 *    être défini.
 *
 * @param   ensemble sur lequel la relation est définie
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     e != null                       // ensemble bien défini
 * @post    lEnsembleDeDefinition() == e    // ensemble mémorisé
 * @post    domaine != null                 // domaine défini
 * @post    codomaine != null               // codomaine défini
 * @post    nCouples == 0                   // relation vide
 */
public              RelationBinaire(
    Ensemble e
    )
{
    ensembleDeDefinition = e ;
    domaine = new Object[NMAX] ;
    codomaine = new Object[NMAX] ;
    nCouples = 0 ;
}
// ---------------------------------- RelationBinaire

//  Les modificateurs
```

```
//

// ------------------------------------------------
/**
 * ajoute un couple à la relation
 *
 *    Cette méthode ajoute un couple à la relation, si celui-ci n'y
 *    appartient pas déjà.  Après l'exécution, l'exécution de la
 *    méthode sontEnRelation sur les éléments du couple doit retourner
 *    vrai.  Le receveur est retourné en résultat pour permettre les
 *    cascades.
 *
 * @param   élément du domaine
 * @param   élément du codomaine
 * @return  retourne le receveur
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     lensembleDeDefinition().appartient(dom)
 *                                          // dom dans l'ensemble
 * @pre     lensembleDeDefinition().appartient(codom)
 *                                          // codom dans l'ensemble
 * @pre     this.sontEnRelation(dom, codom) || nCouples < NMAX
 *                                          // pas plein
 * @post    sontEnRelation(dom, codom)      // relation établie
 * @post    (this.sontEnRelation(dom, codom)@pre
 *                                             && nCouples == nCouples@pre)
 *          || (!this.sontEnRelation(dom, codom)@pre
 *                                           && nCouples == nCouples@pre + 1)
 *                                           // couple ajouté
 * @post    this == return                  // retourne soi-même
 */
public RelationBinaire      ajoute(
    Object dom,
    Object codom
    )
{
    if (!sontEnRelation(dom, codom)) {
        domaine[nCouples] = dom ;
        codomaine[nCouples] = codom ;
        nCouples++ ;
    }                               // if (!sontEnRelation(dom, codom))

    return this ;
}
// ---------------------------------------------- ajoute

// ------------------------------------------------
/**
 * retire un couple à la relation
 *
 *    Cette méthode retire un couple de la relation à condition que
 *    ce couple en fasse effectivement déjà partie.  Après l'exécution
 *    de cette méthode sur un couple donnée, l'appel de la méthode
 *    sontEnRelation sur ce couple doit retourner faux.  Le receveur
 *    est retourné en résultat pour permettre les cascades.
 *
 * @param   élément du domaine
 * @param   élément du codomaine
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     lensembleDeDefinition().appartient(dom)
 *                                          // dom dans l'ensemble
 * @pre     lensembleDeDefinition().appartient(codom)
 *                                          // codom dans l'ensemble
 * @pre     sontEnRelation(dom, codom)      // couple existe
 * @post    !sontEnRelation(dom, codom)     // relation brisée
```

```
     * @post    nCouples == nCouples@pre - 1    // couple retiré
     * @post    this == return                  // retourne soi-même
     */
    public RelationBinaire      retire(
        Object dom,
        Object codom
        )
    {
        int i ;
        boolean pas_trouve = true ;

        for (i = 0 ; i < nCouples && pas_trouve ; i++) {
            if (domaine[i] == dom && codomaine[i] == codom) {
                pas_trouve = false ;
            }                           // if (domaine[i] == dom && ... )
        }                               // for (i = 0 ; ... )
        nCouples-- ;
        domaine[i-1] = domaine[nCouples] ;
        codomaine[i-1] = codomaine[nCouples] ;

        return this ;
    }
    // ------------------------------------------- retire

    // Les accesseurs
    //

    // ----------------------------------------------------
    /**
     * retourne l'ensemble de définition
     *
     * @return  l'ensemble de définition de la relation
     *
     * <STRONG> Contrat </STRONG>
     *
     * @post ensembleDeDefinition == return             // ensemble retourné
     */
    public Ensemble     lEnsembleDeDefinition()
    {
        return ensembleDeDefinition ;
    }
    // ------------------------------ lEnsembleDeDefinition

    // ----------------------------------------------------
    /**
     * vérifie si les deux paramètres sont en relation
     *
     * @param   élément du domaine
     * @param   élément du codomaine
     * @return  vrai si dom est en relation avec codom
     *
     * <STRONG> Contrat </STRONG>
     *
     * @pre     lensembleDeDefinition().appartient(dom)
     *                                          // dom dans l'ensemble
     * @pre     lensembleDeDefinition().appartient(codom)
     *                                          // codom dans l'ensemble
     */
    public boolean      sontEnRelation(
        Object dom,
        Object codom
        )
    {
        /*
         *  dom est en relation avec codom si dom apparaît dans le tableau
         *  domaine et que codom apparaît alors au même indice dans le
         *  tableau codomaine.
         */
```

```
    // Variables locales
    int     i ;
    boolean pas_trouve = true ;

    // Code de la méthode
    for (i = 0 ; i < nCouples && pas_trouve ; i++) {
        if (domaine[i] == dom && codomaine[i] == codom) {
            pas_trouve = false ;
        }                           // if (domaine[i] == dom && ... )
    }                               // for (i = 0 ; ... )
    return !pas_trouve ;

}
// ------------------------------------ sontEnRelation

// ---------------------------------------------------
/**
 * vérifie si la relation est réflexive
 *
 *    Une relation binaire R est réflexive si pour tout élément e, e
 *    est en relation avec lui-même.
 *
 * @return  vrai si la relation est réflexive et faux sinon
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     true
 * @post    return == (forall Object e in
 *                                    lEnsembleDeDefinition().enumerate() |
 *                         this.sontEnRelation(e, e))
 */
public boolean      estReflexive()
{
    // Variables locales
    int i ;
    boolean ret = true ;

    // Code de la méthode
    for (i = 0 ; i < ensembleDeDefinition.nombreElements && ret ; i++) {
        ret = ret && sontEnRelation(ensembleDeDefinition.elements[i],
                                    ensembleDeDefinition.elements[i]) ;
    }                       // for ( i = 0 ; ... )
    return ret ;
}
// ---------------------------------------- estReflexive

// ---------------------------------------------------
/**
 * vérifie si la relation est symétrique
 *
 *    Une relation binaire R est symétrique si pour tous éléments a
 *    et b, si a est en relation avec b, alors b est en relation avec a.
 *
 * @return  vrai si la relation est symétrique et faux sinon
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     true
 * @post    return == (forall Object e1 in
 *                                    lEnsembleDeDefinition().enumerate() |
 *                         (forall Object e2 in
 *                                    lEnsembleDeDefinition().enumerate() |
 *                             this.sontEnRelation(e1, e2)
 *                                 implies this.sontEnRelation(e2, e1)))
 */
public boolean      estSymetrique()
{
```

```
    // Variables locales
    int i ;
    boolean ret = true ;

    // Code de la méthode
    for (i = 0 ; i < nCouples && ret ; i++) {
        ret = ret && sontEnRelation(codomaine[i], domaine[i]) ;
    }                       // for ( i = 0 ; ... )
    return ret ;
}
// ------------------------------------ estSymetrique

// ----------------------------------------------------
/**
 * vérifie si la relation est transitive
 *
 *    Une relation est transitive si et seulement si pour tous éléments a,
 *    b et c, si a est en relation avec b et b est en relation avec c,
 *    alors a est en relation avec c.
 *
 * @return  vrai si la relation est transitive et faux sinon
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     true
 * @post    return == (forall Object e1 in
 *                                  lEnsembleDeDefinition().enumerate() |
 *                        (forall Object e2 in
 *                                  lEnsembleDeDefinition().enumerate() |
 *                            (forall Object e3 in
 *                                  lEnsembleDeDefinition().enumerate() |
 *                                (this.sontEnRelation(e1, e2)
 *                                          && this.sontEnRelation(e2, e3)
 *                                    implies
 *                                        this.sontEnRelation(e1, e3)))))
 */
public boolean      estTransitive()
{
    // Variables locales
    int i, j ;
    boolean ret = true ;

    //Code de la méthode
    for (i = 0 ; i < nCouples && ret ; i++) {
       for (j = 0 ; j < nCouples && ret ; j++) {
           if (codomaine[i] == domaine[j]) {
              ret = ret && sontEnRelation(domaine[i],
                                          codomaine[j]) ;
           }                        // if (codomaine[i] == domaine[j])
       }                            // for (j = 0 ; ... )
    }                               // for (i = 0 ; ... )
    return ret ;
}
// ------------------------------------ estTransitive

// ----------------------------------------------------
/**
 * vérifie si la relation est une relation d'équivalence
 *
 *    Une relation est une relation d'équivamence si elle est réflexive,
 *    symétrique et transitive.
 *
 * @return  vrai si la relation est une équivalence et faux sinon
 *
 * <STRONG> Contrat </STRONG>
 *
 * @pre     true
 * @post    true
```

```
     */
    public boolean      estEquivalence() {

        return this.estReflexive() && this.estSymetrique()
                       && this.estTransitive() ;
    }
    // ------------------------------------ estEquivalence

    // ---------------------------------------------------
    /**
     * toString :       retourne une représentation chaîne de caractères
     *
     *    Cette méthode retourne une représentation de la relation sous la
     *    forme d'une chaîne de caractères obéissant au format suivant :
     *    <OL>
     *    <LI> relation vide : {}
     *    <LI> relation contenant les couples (1,2), et (2,3) : {(1,2), (2,3)}
     *    <OL>
     *    L'ordre dans lequel les couples apparaissent n'a pas d'importance,
     *    i.e. dans le second exemple, on pourrait tout aussi bien obtenir
     *    {(2,3), (1,2)}.
     *
     * @return  chaîne représentant le contenu de la relation
     *
     * <STRONG> Contrat </STRONG>
     *
     * @post    return != null                  // chaîne existe
     */
    public String       toString()
    {
        /*
         * La principale difficulté est le traitement du dernier élément
         * qui ne doit pas être suivi d'une virgule.
         */

        // Variables locales
        int i ;
        String ret = "{" ;

        // Code de la méthode
        i = 0 ;
        while ( i < (nCouples - 1) ) {
            ret = ret + "(" + domaine[i] + "," + codomaine[i] + "), " ;
            i++ ;
        }                                       // while ( i < (nCouples - 1) )
        if ( i == (nCouples - 1) ) {
            ret = ret + "(" + domaine[i] + "," + codomaine[i] + ")}" ;
        } else {
            ret = ret + "}" ;
        }                                       //  if ( i == (nCouples - 1) )

        return ret ;
    }
    // ------------------------------------------- toString

}
// ---------------------------------------------------- Classe RelationBinaire
```

Bibliographie

[AW98] D. Arnow et G. Weiss. *Introduction to Programming Using Java.* Addison-Wesley, 1998.

[Bud98] T. Budd. *Object-Oriented Programming with Java.* Addison-Wesley, 1998.

[Cha96] J. Chazarain. *Programmer avec Scheme.* International Thompson Publishing, 1996.

[CMM+98] G. Clavel, N. Mirouze, S. Munerot, E. Pichon, et M. Soukal. *Java — La synthèse.* InterÉditions, 2 édition, 1998.

[DD00] H.M. Deitel et P.J. Deitel. *Comment programmer en Java.* Éditions Reynald Goulet inc., 3 édition, 2000.

[DH95] R. Decker et S. Hirshfield. *The Object Concept – an introduction to computer programming using C++.* International Thomson Publishing, 1995.

[Eck98] B. Eckel. *Thinking in Java.* Prentice-Hall, 1998.

[Fla97] D. Flanagan. *Java in a nutshell.* O'Reilly & associates, 1997.

[Gra00] V. Granet. *Algorithmique et programmation en Java.* Dunod, 2000.

[Mey97] B. Meyer. *Object Oriented Software Construction.* Prentice Hall, 2nd édition, 1997.

[ML95] V.S. Manis et J.L. Little. *The Schematics of Computation.* Prentice-Hall, 1995.

[Mor00] R. Morelli. *Java, Java, Java — Object-Oriented Problem Solving.* Prentice-Hall, 2000.

[Pap80] S. Papert. *Mind-Storms — Children, Computers and Powerful Ideas.* Basic Books, 1980.

[SA98] Sinan Si Alhir. *UML in a nutshell.* O'Reilly & associates, 1998.

[Sch00] D. Schmidt. *Programming Principles in Java.* Kansas State University, '`http://www.cis.ksu.edu/~schmidt/CIS200`', 2000.

[Sha98] R.L. Shackelford. *Introduction to Computing and Algorithms.* Addison-Wesley, 1998.

[Wei98] M.A. Weiss. *Data Structures & Problem Solving Using Java.* Addison-Wesley, 1998.

[Wir76] Niklaus Wirth. *Algorithms + Data Structures = Programs.* Prentice-Hall, Englewood Cliffs, NJ, 1976.

[WR98] R. Winder et G. Roberts. *Developing Java Software.* John Wiley & sons, 1998.