UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

MÉMOIRE PRÉSENTÉ À

L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI
COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN INFORMATIQUE

OFFERTE À

L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI
EN VERTU D'UN PROTOCOLE
AVEC L'UNIVERSITÉ DU QUÉBEC À MONTRÉAL

PAR

AYMEN SIOUD

GESTION DE CYCLE DE VIE DES OBJETS PAR ASPECTS POUR C++

JUIN 2006

Bibliothèque Paul-Émile-Boulet

UQAC Université du Québec à Chicoutimi

Mise en garde/Advice

Afin de rendre accessible au plus grand nombre le résultat des travaux de recherche menés par ses étudiants gradués et dans l'esprit des règles qui régissent le dépôt et la diffusion des mémoires et thèses produits dans cette Institution, **l'Université du Québec à Chicoutimi (UQAC)** est fière de rendre accessible une version complète et gratuite de cette œuvre.

Motivated by a desire to make the results of its graduate students' research accessible to all, and in accordance with the rules governing the acceptation and diffusion of dissertations and theses in this Institution, the **Université du Québec à Chicoutimi (UQAC)** is proud to make a complete version of this work available at no cost to the reader.

L'auteur conserve néanmoins la propriété du droit d'auteur qui protège ce mémoire ou cette thèse. Ni le mémoire ou la thèse ni des extraits substantiels de ceux-ci ne peuvent être imprimés ou autrement reproduits sans son autorisation.

The author retains ownership of the copyright of this dissertation or thesis. Neither the dissertation or thesis, nor substantial extracts from it, may be printed or otherwise reproduced without the author's permission.

Sommaire

Les langages tels Java, Simula, Eiffel, Modula III sont des langages orientés objet qui ont gagné leur popularité grâce à la sûreté du traitement des exceptions et à là politique de sécurité qu'ils utilisent, notamment pour la gestion mémoire. En effet, Meyer a placé la gestion automatique de la mémoire en troisième position dans les "sept commandements" de la programmation orientée objet. L'entité utilisée pour assurer la gestion automatique de la mémoire est appelée *Garbage Collector* ou ramasse-miettes. Certains langages comme C, C^{++}, Ada, Pascal et bien d'autres utilisent une libération explicite de la mémoire. L'avantage majeur de cette libération par rapport au *Garbage Collector* est sans doute l'emploi des pointeurs, outil très important permettant un accès direct à certaines zones mémoires et une utilisation plus optimale de l'espace mémoire.

Le C++ est l'un des langages de programmation les plus utilisés actuellement. Il est à la fois facile à utiliser et très efficace. Les caractéristiques du C++ en font un langage idéal pour certains types de projets. Il est incontournable dans la réalisation des grands programmes. Les optimisations des compilateurs actuels en font également un langage de prédilection pour ceux qui recherchent les performances. Puisqu'il est une extension de l'ANSI-C, C++ utilise une gestion explicite de la mémoire avec les *delete, new, free* et *malloc*. Pour Ellis et Stroustrup, la gestion de mémoire explicite via le *Garbage Collector* n'est pas une composante du langage C++. Nous proposons dans notre travail de recherche un outil assurant une gestion implicite de la mémoire basé sur la programmation aspect, notamment avec l'extension AspectC++ qui est un préprocesseur pour un compilateur C++ usuel. L'idée est d'implémenter via AspectC++ des compteurs de références pour les objets nouvellement créés. Il s'agit d'attribuer un compteur de références à un objet, d'incrémenter ce compteur chaque fois qu'un autre objet crée une référence vers le premier objet et de décrémenter ce compteur chaque fois qu'une référence est supprimée. L'objet sera détruit dès que son compteur associé sera à zéro.

Table des matières

Liste des figures

Liste des tableaux

À Sana
À mes parents, mes frères et ma sœur, mes beaux-parents
À mes ami(e)s

Remerciements

Tout d'abord, je remercie **Dieu** qui m'a donné le courage et la force de réaliser ce travail et qui a exaucé mes prières en me donnant la chance de pouvoir vous le remettre aujourd'hui.

Ensuite, je remercie mes parents SIOUD Mohamed et Naïma qui ont toujours été là pour moi, qui m'ont toujours fait confiance, qui m'ont fourni tout ce dont j'avais besoin pour que je sois dans les meilleures conditions, allant même jusqu'à se sacrifier. Je leur dédie donc ce projet, comme modeste marque de l'amour que je leur porte pour toujours et qui j'espère, sera un jour à la hauteur de celui avec lequel ils m'ont comblé toute ma vie. Merci aussi à mes frères, Akram et Wadii, à ma sœur Sarra et à tout le reste de ma famille, surtout mes oncles Youssef et Nourredine. Que **Dieu** les garde et leur donne autant de bonheur qu'ils m'en ont donné. À la mémoire de mes grands-parents disparus, qui j'espère de là-haut, seront fiers de moi. Que **Dieu** les protège.

Je remercie aussi la femme et l'amour de ma vie; celle qui m'a soutenu quand j'en avais réellement besoin, celle qui a comblé ma vie par sa gentillesse et sa bonté, celle qui a été à mes côtés durant la majorité de la réalisation de ce travail. Je dédie ce travail, en tant que premier édifice de notre vie, à ma tendre épouse Sana.

Je remercie également mes beaux-parents, Tata Sonia et Tonton Mohamed, qui n'ont pas arrêté de m'encourager et de me soutenir durant cette dernière année. J'ai beaucoup apprécié leur soutien moral, surtout qu'ils m'ont offert le plus beau cadeau de ma vie, leur fille Sana.

J'exprime ma sincère gratitude à mon directeur de recherche M. Marc Gravel et à mon co-directeur M. Hamid Mcheick tant sur le plan du mémoire en lui-même que sur le plan du soutien moral afin que je puisse le terminer. Un très grand merci et chapeau bas pour votre conscience professionnelle, votre générosité et votre disponibilité.

De même, je remercie aussi tout le corps professoral et administratif du département d'Informatique et de Mathématiques de l'Université du Québec à Chicoutimi pour tous les enrichissements que j'ai eus ainsi que leur aide inestimable.

Finalement, je remercie tous mes amis et tous ceux que j'aime, Amine, Anis, Hamadi, Tarek, Aymen mon beau-frère, Charaf, Ouelid et j'en oublie certainement qui me le pardonneront. Je dois tellement à ces personnes qui me transmettent de la joie de vivre tout le temps et qui sont à mes côtés quand j'en ai besoin.

Chapitre 1 : Introduction

Les systèmes d'informations sont devenus au fil du temps l'une des principales clés de réussite de l'entreprise moderne, une entreprise qui connaît de plus en plus d'expansion et de diversification. Au centre de tout système d'information, le logiciel doit assurer plusieurs fonctionnalités pour atteindre une efficacité maximale. Ces fonctionnalités peuvent varier de la comptabilité à la sécurité, en passant par la gestion, l'archivage, la production, la planification, etc. Chaque fonctionnalité peut être assurée par une ou plusieurs entités, où une entité peut être représentée par un sous-programme, une procédure, une fonction, ou un logiciel autonome. L'avènement de la programmation orientée objet a facilité la modularité et la réutilisation des entités du logiciel [Malenfant, 1995] qui sont devenues des objets qui interagissent entre eux. Ici, un objet est un concept informatique permettant de modéliser quelque chose de concret ou d'abstrait de la fonction du logiciel. Que les objets représentent une information persistante ou une information volatile, voire temporaire, ils suivent le même cycle de vie : création, utilisation et terminaison.

Traditionnellement, le cycle de vie d'objets est géré par le programmeur en utilisant les fonctions d'allocations et de libérations d'espace mémoire et on parle alors de gestion explicite de la mémoire. Par exemple, en C/C++, on utilise les fonctions *alloc()*, *malloc()*, l'opérateur *new*, *free()* et les méthodes constructeur et destructeur pour des objets. Dans d'autres approches, le cycle de vie d'un objet est géré implicitement par les langages de programmation via le *Garbage Collector*. Cette action s'appelle la *Garbage Collection* et on parle, dans ce cas, de gestion implicite de la mémoire. Le *Garbage Collector* ou ramasse-miettes est une entité autonome qui réclame de l'espace mémoire dès que le programme en requiert [Jones et Lins, 1996]. Le *Garbage Collector* a été utilisé au début pour des plateformes non distribuées et il a ensuite migré vers des plateformes distribuées. Il existe plusieurs approches de *Garbage Collection* regroupées en deux grandes catégories : des *Garbage Collector* directs et d'autres indirects. Les *Garbage Collector* directs sont basés sur l'approche des compteurs de références alors que toutes les autres approches sont considérées comme *Garbage Collectior* indirects. L'approche

de compteurs de références a été utilisée pour la gestion de mémoire automatique des objets pour des systèmes locaux et distribués. Pour chaque objet, un compteur lui est attribué et il est incrémenté chaque fois qu'un autre objet réfère au premier [Jones et Lins, 1996]. Tant que ce compteur n'est pas à zéro, cet objet devra persister en mémoire. Le compteur de références a été utilisé pour la première fois par Collins en 1960 pour déterminer les portions de programmes qui sont encore utilisées. La plateforme COM/DCOM [Frank, 1997] et certaines versions de la plateforme CORBA (Orbix 3.0) qui offrent une utilisation d'objets distribués proposent une gestion du cycle de vie d'objets en utilisant l'approche des compteurs de références. L'approche de *leasing* est une autre approche utilisée pour la gestion de cycle de vie d'objets au sein des plateformes EJB-J2SE [Hart, 2002][Tanembaum et Van Steen, 2002] et la plateforme CORBA-Orbix6.2 [IONA, 2006]. Le principe de *leasing*, bien que très simple, est très efficace. Au lieu d'allouer une ressource jusqu'à ce que le requérant déclare explicitement ne plus en avoir besoin, on alloue la ressource pour une période donnée appelée bail. Cette période peut être négociée entre les deux parties ou imposée par le gestionnaire de la ressource. Bien entendu, le bail peut être renouvelé *ad libitum* si les circonstances le permettent. En cas de problème, la ressource n'est bloquée que le temps du bail et redevient disponible à l'expiration de celui-ci. Ces deux techniques, bien que très utilisées, ont plusieurs problèmes et ont démontré des failles dans plusieurs langages de programmation et plateformes [Jones et Lins, 1996] [Tanembaum et Van Steen, 2002]. En effet, le principal problème concernant les compteurs de références est sans doute son incapacité à gérer les cycles [Jones et Lins, 1996]. Concernant le *leasing*, puisque les baux sont basés sur des heuristiques, ils sont rarement égaux au temps réel d'utilisation. Des améliorations ont été apportées à la technique de *leasing* pour pallier à certaines failles [IONA, 2006], notamment dans les systèmes distribués. Il existe, outre la technique des compteurs de références, d'autres techniques de *Garbage Collection* qui se sont avérées très efficaces pour certains langages comme les techniques *Mark and Compact* et *incrémental* [Jones et Lins, 1996] qui seront présentées en détail au deuxième chapitre.

Toutes ces techniques ont permis à la gestion implicite de la mémoire de généraliser la plupart des langages orientés objet. En effet, Meyer [1988] a placé la gestion implicite de la mémoire en troisième position dans ses "sept commandements" de la programmation orientée objet. Selon lui, le *Garbage Collector* facilite la tâche des programmeurs qui n'ont plus à se soucier de la libération de la mémoire. De son côté, Ellis [1993] a établi que les programmeurs passent plus des deux cinquième du temps accordé à l'élaboration du logiciel à déboguer les erreurs liées à la gestion de la mémoire pour les langages à gestion mémoire explicite comme C++. Certains chercheurs comme Detlefs [1992], Boehm et Weiser [1988] ou encore Ferreira [1991] ont intégré un *Garbage Collector* au sein de C++ pour remédier à ces problèmes reliés au débogage de la mémoire. Ils ont utilisé à cet effet plusieurs techniques comme les *Garbage Collector* conservatifs [Boehm et Weiser, 1988], les *smart pointers* qui sont des classes qui imitent le comportement des pointeurs [Alexandrescu, 2001] ou encore les compteurs de références [Ellis, 1993]. Detlefs [1992], puis Ellis et Detlefs [1993] ont énuméré un ensemble de problèmes et de contraintes qu'il faudrait surpasser pour intégrer un *Garbage Collector* à C++ comme la coexistence de la gestion implicite et explicite de la mémoire. Toutes les tentatives d'intégration d'un *Garbage Collector* n'ont pu surpasser ces problèmes et ces contraintes. Les travaux de Detlefs [1992] et de Boehm [2002] restent cependant optimistes puisqu'ils ont pu résoudre la majorité des problèmes rencontrés à l'intégration d'un *Garbage Collector.*

Le cycle de vie des objets se décompose en quatre phases tel que décrit dans la Figure 1-1 :

- L'application crée les objets.
- L'application leur alloue l'espace nécessaire.
- L'application utilise les objets.
- L'application libère l'espace occupé par ces objets.

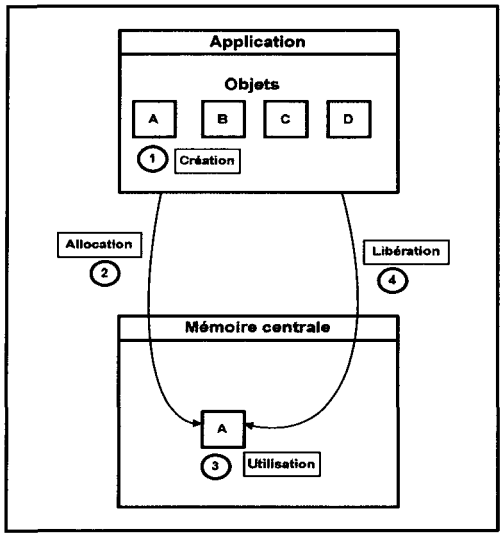

Figure 1-1 : Cycle de vie d'un objet

Dans ce travail de recherche, un outil assurant la gestion implicite de la mémoire au sein de C++ sera proposé et évalué. La programmation par aspects [Kicsales *et al.*, 1997], incarnant la séparation des préoccupations [Tarr *et al.*, 1999] (*Separation of concerns* en anglais) et formant un support méthodologique de modélisation et de programmation, sera utilisée à cet effet. Dans le code source d'une classe écrite avec un langage orienté objet comme C++ ou Java, plusieurs lignes sont consacrées à la synchronisation des accès aux ressources, à la sauvegarde des données (la persistance), au traitement des exceptions, à la vérification des paramètres entrants, etc. On parle alors de *préoccupations entrecroisées* (*Crosssutting concerns* en anglais) [Kicsales *et al.*, 1997]. Ces lignes de code ont tendance à être réécrites dans la plupart des méthodes des classes d'une application. Il en résulte que les classes sont peu réutilisables telles quelles par d'autres applications et sont, pour ainsi dire, *couplées* avec l'application. Il en résulte aussi que le changement de politique sur un de ces aspects de l'application (par exemple, si on décide de ne plus enregistrer certaines informations) oblige la modification de nombreuses lignes de code éparpillées dans le code source. Cet éparpillement pourrait décrire un problème de localité textuelle: tout code concernant le même aspect devrait être regroupé dans un même endroit ; cela se rapproche bien sûr du concept de modularité. La programmation par aspects permettra donc de minimiser le couplage et d'augmenter la cohérence lorsque les méthodes classiques de l'objet atteignent leurs

limites. Idéalement, chaque tâche spécifique devrait être la seule responsable d'une composante ou d'un groupe de composantes éventuellement rassemblées au sein d'un groupement logique. Cependant, la gestion mémoire au sein de C++ est gérée par plusieurs classes et elle est couplée avec son application. Ce travail de recherche consiste dans une première étape à composer un aspect technique qui regroupe la gestion mémoire d'une application donnée. Dans une deuxième étape, cet aspect sera intégré avec le reste des aspects fonctionnels tout en gardant l'intégrité des différentes fonctionnalités du compilateur. Notre outil est conçu pour les applications légataires qui utilisent déjà l'outil explicite pour la gestion de la mémoire en C++.

Ce travail de recherche est divisé en cinq chapitres. Après l'introduction présentée dans ce chapitre, le chapitre 2 présentera une revue de la littérature pertinente sur les différentes techniques utilisées pour la gestion mémoire explicite et implicite en systèmes logiciels ainsi que les différentes techniques employées au sein de C++ pour assurer la gestion mémoire. Le chapitre 3 présentera, pour sa part, la séparation des préoccupations et, plus particulièrement, la programmation par aspects pour aboutir par la suite aux objectifs de la recherche. Le chapitre 4 présentera une description détaillée de l'outil de gestion mémoire proposé et son évaluation. Enfin, le mémoire se termine par une conclusion et quelques aspects d'une recherche future seront proposés.

Chapitre 2 : Gestion du cycle de vie des objets

Nous décrivons dans ce chapitre, dans une première étape, les cycles de vies implicites et explicites des objets. Dans une seconde partie, nous décrivons le cycle de vie des objets au sein du langage C++.

2.1 Gestion du cycle de vie des objets

2.1.1. Gestion du cycle de vie des objets locaux

Lors du chargement d'une application en mémoire centrale, la création et la gestion des objets, des variables et des structures de données nécessaires à cette application sont invisibles à l'utilisateur. Ces variables sont créées aux besoins de l'application. Elles pourront être modifiées au cours du déroulement de l'application et elles seront détruites à l'achèvement de l'application. La création, la gestion et la destruction de ces variables et structures forment ce qu'on appelle le cycle de vie d'une variable [Bray, 1977]. Ainsi, à son appel, une variable est initialisée en mémoire centrale et elle occupera un espace mémoire qui dépendra de sa taille. À la fin de l'application, elle devra être détruite pour récupérer cet espace mémoire occupé puisque cette variable n'est devenue d'aucune utilité. On parle alors de gestion mémoire de la mémoire centrale ou plus simplement de gestion mémoire. Cette dernière nous permet donc d'être sûrs que l'espace mémoire n'est pas détenu par des objets ou des variables qui ne sont plus d'aucune nécessité pour nos applications. Cet espace mémoire occupé inutilement pourrait être utile pour d'autres objets ou variables de la même application ou d'une autre [Metropolis *et al.*, 1980]. Nous ne pouvons parler de cycle de vie d'objets ou de variables sans parler de gestion mémoire, car le cycle de vie d'une variable ou d'un objet sera incomplet si ces derniers ne seront pas détruits [Bray, 1977]. Le cycle de vie des objets n'est pas géré par l'utilisateur ni par le système où évolue l'application, mais il est géré par le langage de programmation avec lequel l'application a été implantée. Quand un objet n'est plus sous le contrôle d'une application sans avoir libéré l'espace qu'il occupait, la mémoire est

perdue inutilement. Cette perte de mémoire est appelée *fuite mémoire* [Hirzel *et al.*, 2002], elle peut être due à une mauvaise manipulation de l'utilisateur par exemple. La majorité des langages de programmation, voire la totalité, ont essayé de résoudre le problème de cette *fuite mémoire*.

La gestion de la mémoire est la succession de deux étapes différentes que chaque langage de programmation doit assurer : une étape d'allocation de la mémoire et une autre de libération. La première consiste à allouer de la mémoire pour les variables ou les objets et la seconde libère la mémoire allouée qui n'est pas utilisée. La gestion mémoire a vu deux périodes distinctes depuis l'utilisation de la mémoire centrale pour les applications [Bray, 1977]. On parlait au début des années cinquante d'une gestion statique de la mémoire, qui est passée à une gestion dynamique de la mémoire vers le début des années soixante [Metropolis *et al.*, 1980].

Dans le cas d'une gestion statique, toutes les données sont délimitées en mémoire. Les limites des différentes structures et variables ne changent pas pendant l'exécution. Les principaux inconvénients sont, d'une part, que la taille des structures de données doit être connue d'avance et qu'on ne pourra pas changer ou ajouter un champ dans une structure, par exemple. D'autre part, il ne peut y avoir de procédures récursives. Ces inconvénients, surtout l'absence de la récursivité [Bray, 1977], ont poussé les chercheurs à adopter une nouvelle stratégie pour l'utilisation de l'espace mémoire. Une nouvelle technique de gestion de l'espace mémoire est apparue avec les langages structurés, notamment avec Algol 58 et Atlas Autocode [Baecker, 1970] et on entra alors dans l'ère de la gestion mémoire dynamique. Celle-ci se décompose en deux branches et ce, selon la structuration de l'espace mémoire à gérer. Une gestion mémoire par *stack*, employée par Algol 58 [Baecker, 1970], utilise un tas pour y empiler les données au fur et à mesure de leur création. La deuxième structure utilisée est une *heap* [Hirzel *et al.*, 2002]. La gestion mémoire par *heap* consiste à allouer les données selon un ordre quelconque dans cette structure. La récursivité y est aussi possible et contrairement à la technique par *stack*, les données et les procédures peuvent survivre même si leurs prédécesseurs sont détruits.

Si l'allocation se fait d'une manière manuelle par la création des objets et la déclaration des pointeurs et des variables, la libération de l'espace mémoire peut se faire de deux manières : soit manuellement ou automatiquement. Dans le cas d'une

désallocation manuelle, c'est le programmeur qui se charge de ce travail explicitement. On peut citer les fonctions *delete* et *free* utilisées par le langage de programmation C++. Cette technique est utilisée par C, C++, Ada, Pascal, etc. Dans le cas d'une désallocation automatique, la mémoire est implicitement récupérée. C'est au langage de programmation de gérer la récupération de la mémoire. Plusieurs langages utilisent cette technique, parmi lesquels on cite Java, Simula, Eiffel, Modula III, etc. La récupération implicite de la mémoire au sein de ces langages est assurée par une entité appelée *Garbage Collector*. Dans la partie suivante, la gestion implicite de la mémoire via le *Garbage Collector* sera décrite, le tout suivi d'explications sur la gestion explicite de la mémoire.

2.1.2. Gestion de mémoire implicite

2.1.2.1 Introduction

Les langages de programmation tels Java, Lisp, Eiffel et Modula III sont des langages orientés objet qui ont gagné leur popularité grâce à une nouvelle philosophie de programmation mais aussi grâce à la sûreté du traitement des exceptions et de la politique de sécurité qu'ils utilisent. Meyer [1988] a placé la gestion automatique de la mémoire en troisième position dans les "sept commandements" de la programmation orientée objet. En effet, selon Meyer [1988], une gestion dynamique de la mémoire via une désallocation automatique est un bon outil pour assurer de bonnes performances des langages de programmation. L'entité utilisée pour assurer la gestion automatique de la mémoire est appelée *Garbage Collector* ou ramasse-miettes. Les termes *Garbage* et *miettes* désignent les objets qui ne sont pas référencés, c'est-à-dire ceux que le programme en cours d'exécution ne pourra jamais atteindre. La présence de ces *Garbage* en mémoire est inutile et peut causer un encombrement de la mémoire pour d'autres objets exigés par le programme. Avec la gestion de mémoire implicite, le développeur n'a pas à se soucier de libérer la mémoire utilisée par les objets inutilisés puisqu'un collecteur de déchets teste en permanence chaque objet pour s'assurer qu'il est toujours utilisé.

L'une des caractéristiques d'un *Garbage Collector* est qu'il doit évoluer de manière transparente à l'utilisateur. Il a été créé essentiellement pour répondre aux problèmes de

désallocation implicite, mais solutionne aussi certains problèmes de langage et de matériel. En effet, une gestion manuelle de la mémoire peut causer une occupation inutile de l'espace mémoire et une perte de références pouvant entraîner des perturbations lors de l'exécution du programme et engendrer des problèmes tels que :

- Un espace mémoire occupé inutilement : quelques blocs mémoires alloués peuvent devenir, à la suite de différentes actions, inaccessibles. Par exemple, si une tête de liste pointe vers aucune adresse, soit vers NULL, alors tous les membres de la liste deviendront inaccessibles étant donné que la tête de liste a perdu la référence vers les éléments suivants. Le reste des éléments de la liste ne sont plus utiles, mais ils occupent encore de l'espace mémoire.
- Une perte de références : si le deuxième membre d'une liste est désalloué, le troisième ainsi que le reste de la liste deviendra du *Garbage*. Le pointeur du premier membre vers le deuxième référencera vers de l'espace mémoire désalloué.

La fonction du *Garbage Collector* ne se limite pas à la libération de la mémoire seulement, mais c'est par son biais que se fait l'allocation via une entité appelée *mutateur* [Abdullahi et Ringwood, 1998]. Un mutateur est une entité du *Garbage Collector* qui s'occupe de l'allocation de la mémoire et fait partie intégrante du langage de programmation. La libération se fait, quant à elle, via une deuxième entité appelée le *collecteur* [Abdullahi et Ringwood, 1998]. Ces deux entités forment ensemble le *Garbage Collector*. Dans ce qui suit, nous nous intéressons seulement au *collecteur*, ainsi le *Garbage Collector* sera réduit à la tâche de récupération de la mémoire.

Les techniques de *Garbage Collection* (l'action reliée au *Garbage Collector*) sont nombreuses. Il s'agit de différents algorithmes et approches pour récupérer l'espace mémoire occupé par les objets et il existe deux principales classifications pour ces algorithmes. La première classification regroupe les techniques de *Garbage Collection* en deux catégories : des *Garbage Collector* directs et d'autres indirects. Les *Garbage Collector* directs représentent les *Garbage Collection* qui se basent sur l'approche des compteurs de références. Toute autre approche est considérée comme *Garbage Collection* indirect. L'approche des compteurs de références est utilisée pour la gestion de mémoire automatique des objets. Un compteur est attribué à chaque objet. Ce compteur est

incrémenté chaque fois qu'un autre objet réfère au premier et, tant que ce compteur n'est pas à zéro, cet objet devra persister en mémoire. Le compteur de références a été utilisé pour la première fois par Collins [1960] et a reçu de nombreuses améliorations qui seront décrites ultérieurement dans ce travail.

Une autre classification regroupe les techniques de *Garbage Collection* en deux familles. La première famille contient les algorithmes les plus utilisés et qui sont à la base de l'élaboration de la majorité des autres algorithmes [Jones et Lins, 1996]. Ces algorithmes sont appelés algorithmes de base et regroupent les algorithmes de compteurs de références, les algorithmes *Mark and Sweep* et les algorithmes de recopie. La deuxième famille regroupe des algorithmes qui sont des variantes des algorithmes de base comme les algorithmes conservatifs ou les algorithmes qui ont été créés pour répondre à certains problèmes rencontrés dans les algorithmes de base comme le *Mark and Compact.* Il est important de signaler que le choix de l'algorithme de *Garbage Collection* est très important parce qu'il a une influence considérable sur les performances du langage de programmation [Levanoni et Petrank, 2001].

Usant d'une approche quelconque, un algorithme de *Garbage Collection* a généralement deux tâches principales :

- déterminer et détecter les objets à collecter;
- récupérer l'espace libéré et le mettre à la disposition du programme.

Pour réaliser ces deux tâches, le *Garbage Collector* sera amené à déterminer si un objet est utile ou non au programme. S'il est utile, on parlera d'objet « vivant » et, dans le cas contraire, on parlera alors d'objet « mort » (*Garbage*). Dans une première étape, le *Garbage Collector* est amené à définir un ensemble de racines et à déterminer l'atteignabilité des objets à partir de ces racines. Les racines représentent les variables globales, celles de la pile et les registres [Jones et Lins, 1996]. En pratique, la racine est constituée des variables globales, des piles des programmes qui sont exécutés et des registres CPU tel qu'illustré à la Figure 2-1.

Un objet est accessible, atteignable ou vivant [Abdullahi et Ringwood, 1998] s'il existe un chemin de références (pointeurs) depuis les racines par lequel le programme peut y accéder. À la Figure 2-1, les objets A, C et D sont accessibles, mais les objets E et F ne le sont pas. Un objet référé par une racine est accessible et sera donc un objet vivant.

De plus, n'importe quel objet référé par un objet vivant est aussi un objet accessible comme c'est le cas de l'objet B. Le *Garbage Collector* construit ce qu'on appelle le graphe d'objets atteignables qui est formé de tous les objets accessibles à partir de toutes les racines. Le programme est capable d'accéder à tout objet atteignable et ces objets doivent alors demeurer en mémoire. Tous les objets qui ne sont pas accessibles par le programme seront alors collectés par le *Garbage Collector* puisqu'ils sont considérés comme des miettes. Dans l'exemple présenté à la Figure 2-1, E et F ne sont pas accessibles ni directement, ni par l'intermédiaire d'autres objets et ils sont considérés comme des miettes.

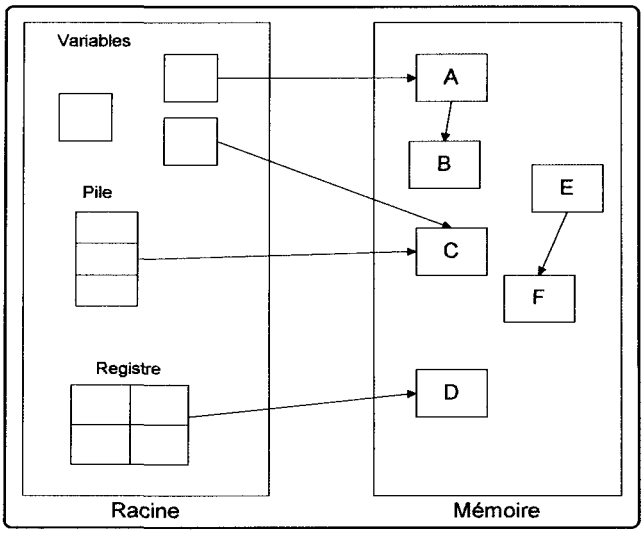

Figure 2-1 : Disposition de la mémoire

Concernant les miettes, toutes les techniques de *Garbage Collection* supposent que ces dernières doivent rester stables en mémoire et qu'elles ne peuvent, en d'autres termes, en aucun cas redevenir vivantes. Le choix de l'algorithme de *Garbage Collection* est très important parce qu'il a une influence considérable sur les performances de l'application en général et sur le langage de programmation en particulier [Levanoni et Petrank, 2001]. Dans une première partie, nous élaborerons une étude des trois algorithmes de base. Dans une deuxième partie, nous ferons une étude non exhaustive des autres techniques.

2.1.2.2 Les Garbage Collector de base

a - *Garbage Collector* par Compteur de références

Le *Garbage Collector* par compteur de références compte le nombre de références qui pointent vers les objets. En pratique, ce compteur est un entier qui est stocké dans l'entête de l'objet lui-même. Le compteur de références est incrémenté chaque fois que sa référence est dupliquée. À l'inverse, lorsqu'une référence d'un objet est détruite, le compteur associé est décrémenté. Le *Garbage Collector* intervient et collecte les objets pour lesquels le compteur de références est égal à 0. Donc, la disparition d'un objet peut entraîner la disparition d'autres objets. En effet, à la Figure 2-2 qui présente une mémoire à son état initial avec des racines et six objets, si la référence vers l'objet A qui est en ligne discontinue venait à être effacée, le compteur de cet objet passerait à 0 et devrait être collecté. De même, les compteurs respectifs des objets C et E seraient décrémentés et passeraient à 0 pour C et à 1 pour E. La réaction n'est pas encore finie, car C serait à son tour collecté et les compteurs de B et D passeraient également à 0 et seraient à leur tour collectés. La Figure 2-3 présente la mémoire à son état final.

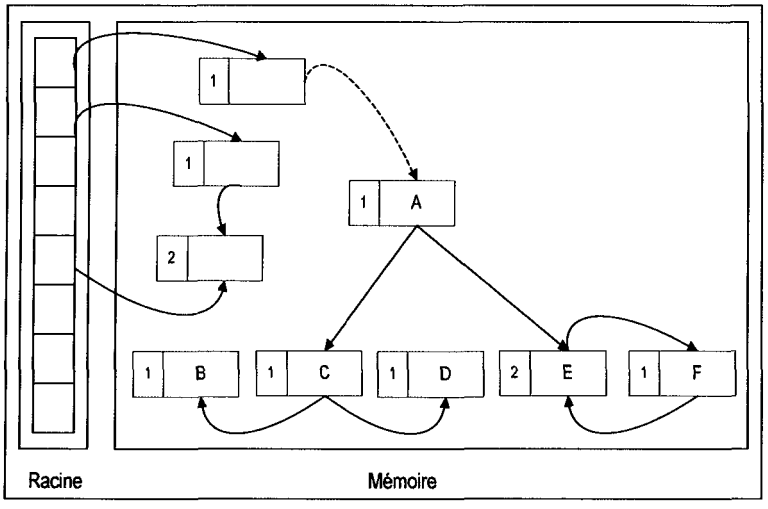

Figure 2-2 : Compteurs de références – Mémoire à l'état initial

Les objets recyclés sont généralement chaînés dans une liste d'objets libres. Cette liste donne les emplacements de mémoire libre ainsi que la taille de chaque emplacement. Lorsqu'un programme demande à allouer un objet, le système extrait de cette liste le

meilleur emplacement susceptible d'accueillir l'objet. Le système gère souvent plusieurs listes et non pas une seule, ce qui permet une accélération de la recherche de l'emplacement optimal.

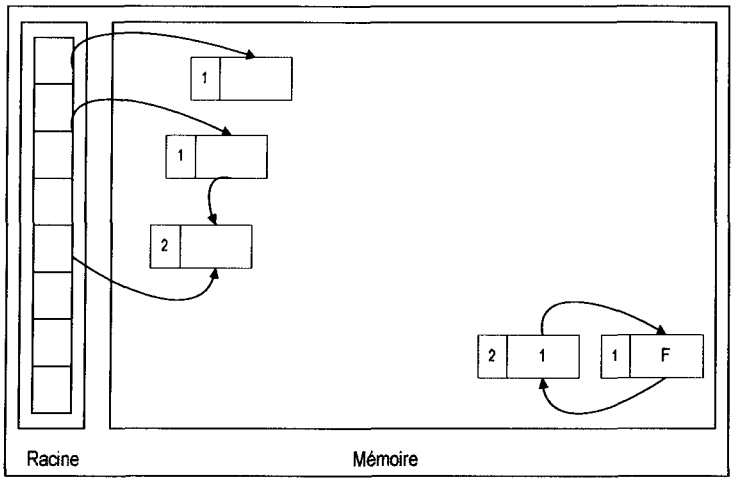

Figure 2-3 : Compteurs de références – Mémoire à l'état final

Les avantages de cet algorithme sont :

- Le recyclage se fait dès qu'un objet devient miette;
- Le principe est incrémental, c'est-à-dire que le *Garbage Collector* fonctionne au fur et à mesure que s'exécutent le ou les programmes de l'utilisateur.

Par contre, les inconvénients que nous pouvons citer sont :

- Le *Garbage Collector* ne peut recycler les structures cycliques d'objets non atteignables, comme les objets E et F de la Figure 2-3. Ces objets sont non atteignables, mais leurs compteurs ne sont pas à 0, ils ne seront pas collectés.
- Le compteur étant généralement un entier, il ne faut pas qu'il déborde, sinon, il pourrait entraîner le recyclage d'objets vivants ou le non-recyclage des miettes.
- Ce type de *Garbage Collector* n'est pas rapide. Cette consommation temporaire assez élevée est due à la lourde gestion des compteurs [Jones et Lins, 1996]. Il faut, chaque fois, incrémenter ou décrémenter et tester si égal à zéro. Par exemple, si le premier objet d'un arbre est libéré, ce sont tous les compteurs de la branche associée qui sont décrémentés.

- De la gestion par liste chaînée des objets morts en vue de réallocation, il en résulte forcément une fragmentation de la mémoire quelle que soit la technique d'attribution employée.

Le coût de ce type de *Garbage Collector* est directement proportionnel à l'activité des programmes utilisateurs, incluant une grande constante de proportionnalité : dès qu'un pointeur est créé ou détruit, il faut appeler le *Garbage Collector.* À chaque décrémentation de compteurs, le *Garbage Collector* est sollicité pour vérifier s'il n'y aurait pas des objets dont le compteur de références est passé à 0 et par conséquent collecter ces objets [Jones et Lins, 1996].

Cet algorithme a été utilisé pour la première fois par Georges Collins [1960] pour tester les parties non utilisées dans une application dédiée à la manipulation des listes. Cependant la première description a été faite par Gelernter *et al.* [1960]. Cette technique a été utilisée par les premières versions de Smakltalk en 1983 [Goldberg et Robson, 1983], Interlisp et Modula II [DeTreville, 1990] en 1982 également, mais aussi par les utilitaires *awk* et *perl* de Unix en 1988 [Aho *et al.*, 1988]. C'est Weizenbaum [1963] qui a introduit l'utilisation des listes pour récupérer la mémoire collectée. Plusieurs variantes de cette technique ont été utilisées pour divers systèmes et ce, pour essayer de remédier aux inconvénients cités plus haut. En particulier, à celui lié à une consommation temporelle assez importante. On peut citer l'algorithme de compteurs de références déféré de Deutsch et Bobrow [1976] où le *Garbage Collector* n'est appelé que lorsqu'il y a des décrémentations liées aux variables référencées directement aux racines. L'algorithme de Christopher [1984], qui fut utilisé pour Fortran, est appelé périodiquement pour les nœuds aux compteurs différents de 0. Pour différencier les objets en mémoire, cet algorithme attribua une couleur à chaque objet : noir, blanc ou gris. Les objets « blancs » étaient considérés comme étant des miettes. La coloration des objets, qui a été introduit par Dijkstra *et al.* [1978], permettait à l'application d'évoluer en parallèle avec le *Garbage Collector.* Bobrow [1980] a utilisé une technique qui a permis de surmonter le problème lié aux cycles. Il a utilisé le principe suivant : un cycle ne peut être collecté que s'il n'a pas de référence externe. Le cycle formé par les objets E et F de la Figure 2-3 pourra être collecté puisqu'il ne présente pas de références externes. Les travaux de Shapiro et *al.*[1992] ont aussi contribué à améliorer les performances de la

technique des compteurs de références en surmontant le problème des cycles et en l'adoptant aux systèmes distribués comme dans le cas des travaux de Lins et Jones [1993].

b - *Garbage Collector Mark and Sweep*

Le principe de ce *Garbage Collector* est très simple, car il trace un graphe des objets en débutant par les racines et marque tous les objets visités. Les objets ne sont pas collectés lors de la première visite du *Garbage Collector*, mais lors de la deuxième. Ce dernier marque les objets accessibles à partir des racines. L'action de marquer des objets se fait soit par des drapeaux (*flag*) à l'intérieur de l'objet, soit dans une autre variable qui est généralement un tableau où l'on parlera alors de *bitmap* [Jones et Lins, 1996]. Dans une deuxième étape, le *Garbage Collector* parcourt toute la mémoire en libérant les objets non marqués. La mémoire récupérée peut être maintenue sous forme de liste chaînée.

Lors d'une première phase de marquage (*Mark*) à partir des racines, le *Garbage Collector* marquera tous les objets qu'il rencontre en traçant le graphe d'objets. À la Figure 2-4, les objets A, B et C seront marqués dans un champ supplémentaire dédié à l'utilisation du *Garbage Collector.* Dans une deuxième étape (la phase *Sweep*), le *Garbage Collector* balayera tout l'espace mémoire et visitera toutes les cellules mémoires pour vérifier si elles sont marquées ou non. Si elles ne le sont pas, le *Garbage Collector* les enchaînera à l'ensemble de l'espace libre de la mémoire. Ainsi, dans cet exemple, les objets D, E et F seront chaînés au reste de l'espace libre. Comme nous pouvons le remarquer, cette technique peut surmonter le problème des cycles (l'objet D et E) et recycler les objets inaccessibles, ce que ne faisait pas le *Garbage Collector* par comptage de références. Par contre, les inconvénients suivants peuvent être cités :

- Le fait de collecter des objets dispersés en mémoire peut causer une fragmentation de la mémoire et, par la suite, dégrader les performances de l'application [Jones et Lins, 1996].
- Le coût du *Garbage Collector* est directement proportionnel à la quantité de mémoire dont on dispose puisque tous les processus sont stoppés lors de l'activation du *Garbage Collector.*

- Puisque les objets ne sont jamais déplacés, il se pose le problème de la localité des objets : ce type de *Garbage Collector* n'est donc pas utilisable dans un système basé sur la mémoire virtuelle.

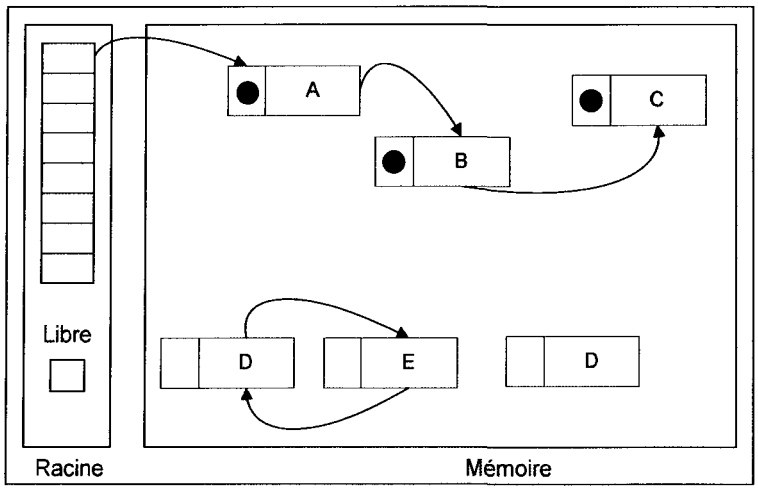

Figure 2-4 : *Mark and Sweep*

Les premières descriptions de cet algorithme sont apparues en 1960, à la suite des travaux de Mc Carthy [1960] qui l'utilisait pour calculer les temps de calcul des fonctions récursives. Thorelli [1972], lui, a préposer d'ajouter l'espace mémoire récupéré des objets collectés à une liste d'espace libre. Baecker [1972] a réduit le coût de la phase de marquage des objets en marquant toute une page mémoire au lieu de marquer les cellules mémoires. Boehm et Weiser [1988] ont, quant à eux, proposé une organisation de la mémoire formée par des blocs contigus de mémoire afin de limiter la fragmentation. Ils ont aussi utilisé une file pour allouer l'espace aux objets et donner des priorités. Knuth [1973] a utilisé une file circulaire pour allouer l'espace aux objets et il a ainsi pu réduire le temps de la phase de *sweep*. Hughes [1982], pour sa part, a pu paralléliser la phase de l'allocation des objets et celle de marquage. Enfin, Zorn [1989] a utilisé des listes pour paralléliser les deux tâches ainsi qu'un tableau pour les *bitmap*. Les problèmes avec cet algorithme, qui est aussi celui de la technique des compteurs de références, est la fragmentation [Jones et Lins, 1996]. Ainsi, les espaces mémoires des objets libérés ne sont pas contigus et leur utilisation ne se fera pas de manière optimale [Jones et Lins,

1996]. Un autre algorithme sera alors utilisé, soit celui *Mark and Compact*, dont la description sera faite ultérieurement à la section 2.1.2.3.

c - *Garbage Collector* par recopie

Dans ce type d'algorithme, le tas est divisé en deux zones mémoires égales et contiguës : une zone de départ (*fromspace*) et une zone d'arrivée (*tospace*). Ce *Garbage Collector* ne sépare pas les phases de marquage et de balayage. Durant l'exécution normale des programmes, seul le « *fromspace* » est utilisé. On recopie linéairement les cellules accessibles depuis les racines de la zone de départ vers la zone d'arrivée. Les objets sont donc placés l'un à côté de l'autre de façon à obtenir un demi-espace défragmenté à la fin de la copie et, par la suite, les fonctions des deux demi-espaces sont échangées.

Dans la Figure 2-5, les objets et les références sont représentés en pointillés dans le « *fromspace* ». Lors du passage du *Garbage Collector*, les objets accessibles à partir du « *fromspace* » sont copiés un à un dans « *tospace* » en effectuant une mise à jour des références en leur attribuant leurs nouvelles valeurs dans le « *tospace* ». Finalement, le rôle des espaces est permuté, ainsi le « *fromspace* » devient le « *tospace* » et vice versa.

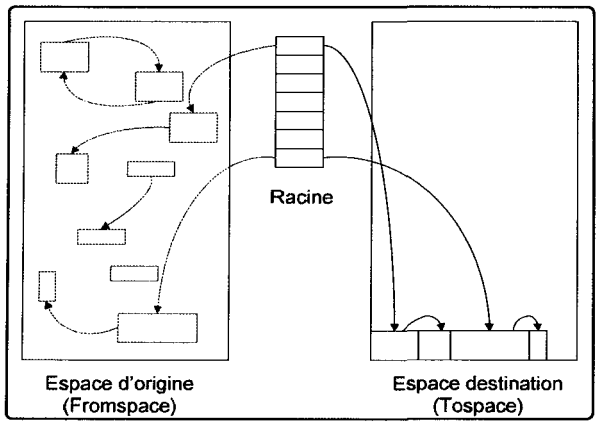

Figure 2-5 : *Garbage Collector* par recopie

Les avantages d'un tel *Garbage Collector* sont :

- Son coût temporel est intéressant, car il est directement proportionnel au nombre d'objets « vivants » à gérer et non à la taille totale de la mémoire ni à la quantité de miettes à gérer.
- La fragmentation mémoire n'existe plus lors de l'utilisation de cet algorithme.
- L'allocation est plus rapide, car il n'y a plus de liste chaînée d'objets « morts ».
- Il y a utilisation optimale de la mémoire. Le nombre de pages virtuelles diminue parce que les données sont compactées. Les objets sont accolés, donc il y a de fortes raisons pour que deux objets qui vont de pair soient sauvegardés dans la même page virtuelle.

Les inconvénients que nous pourrions citer sont :

- Les performances de ce type de *Garbage Collector* sont bonnes [Jones et Lins, 1996] si et seulement si peu d'objets sont copiés d'un espace vers un autre. S'il y a beaucoup de miettes et peu d'objets vivants, il peut y avoir une recopie inutile d'objets ayant une longue durée de vie.
- La taille mémoire physique (réelle) est divisée par deux, ce qui contribue à réduire l'espace d'utilisation de cette mémoire.

Depuis que Cheney [1970] a découvert une méthode itérative pour appliquer un *Garbage Collector* par recopie, la popularité de ces algorithmes a considérablement augmenté [Jones et Lins, 1996]. L'algorithme de Cheney a aussi utilisé la technique de « à la volée » utilisant les trois couleurs de Dijkstra *et al.* [1978]. L'avantage de cet algorithme est le fait que son coût est proportionnel au nombre d'objets « survivants » après chaque passage du *Garbage Collector* et non à la taille de l'espace mémoire comme dans le cas de l'algorithme *Mark and Compact.* Pour plusieurs applications et langages, le nombre d'objets « survivants » est très réduit [Hudak et Keller, 1982] [Zorn 1989] [Appel 1992]. C'est Fenichel et Yochelson [1969] qui, en utilisant un algorithme récursif, ont introduit pour la première fois l'utilisation des deux semi-espaces. La division de l'espace mémoire en plusieurs parties et la recopie ont ensuite été utilisées à maintes reprises [Stamos 1984] [Courts 1988] [Wilson, 1990]. Ce type d'algorithme sera présenté à la section suivante. Celui de Cheney [1970] reste cependant le plus élégant et le moins coûteux en consommation temporelle [Jones et Lins, 1996]. Jones et Lins [1996] ont

effectué une étude comparative entre les algorithmes *Mark and Sweep* et les algorithmes de recopie et qui ont tous les deux une même borne asymptotique.

Le Tableau 1 résume les complexités des algorithmes *Mark and Sweep* et de recopie. Nous pouvons remarquer que les coûts de traçage et d'allocation sont constants et égaux pour les deux algorithmes, ces coûts de la phase du *Sweep* et de la recopie sont en fonction de la taille de la mémoire et du nombre d'objets à recopier. La comparaison ne contient pas la technique des compteurs de références parce que cette dernière est une technique directe de *Garbage Collection*, alors que celle de recopie et *Mark and Sweep* sont des techniques indirectes. Le traçage représente ici la phase de marquage.

Tableau 1 : Complexité asymptotique des phases *Mark and Sweep* et du *Garbage Collector* par recopie. *M* représente la taille de l'espace mémoire, *R* est la taille de la mémoire utilisée. Issu de ftp://parcftp.xerox.com/pub/garbage/complexity.ps Hans Boehm [1995]

Méthode	*Mark and Sweep*	Recopie
Initialisation	**négligeable**	**négligeable**
Traçage	***O*(R)**	***O*(R)**
Phase de sweep	**-**	**-**
Allocation	***O*(M-R)**	***O*(M-R)**

2.1.2.3 Autres techniques de Garbage Collection

a- Les collecteurs incrémentaux

Dans ce type d'algorithme, on décompose l'espace mémoire en plusieurs parties contiguës et les objets seront répartis sur les différentes zones. On parle alors d'algorithmes concurrents ou parallèles qui font en sorte que la collecte se fait dans plusieurs zones simultanément. Une autre caractéristique de ces algorithmes est la possibilité d'avoir l'application et le *Garbage Collector* tournant en même temps. On parle alors de *Garbage Collector* concurrent [Jones et Lins, 1996].

Le temps de parcours total sera égal à la somme des temps de parcours des différentes parties si le parcours n'est pas parallèle. Ce temps sera supérieur à celui d'une collecte atomique de tout l'espace mémoire. En effet, la collecte d'une ou deux zones pourrait suffire à trouver l'espace requis pour allouer de nouveaux objets. Le gain de

temps engendré facilite l'implantation de tels algorithmes pour des environnements à temps réel [Jones et Lins, 1996]. Une autre caractéristique de ces algorithmes est la possibilité d'utiliser, pour chaque région mémoire, une technique de *Garbage Collector* différente. On pourrait utiliser un *Mark and Sweep* pour la première région, un algorithme par recopie pour la deuxième, etc.

C'est Knuth [1973] qui proposa pour la première fois l'idée d'un algorithme incrémental en parallélisant le *Garbage Collector* et l'application. Steel [1975] a décrit un algorithme « à la volée », mais il n'est jamais devenu aussi populaire que celui de Dijkstra [1976] étant donné le manque de détails de ses travaux [Jones et Lins, 1996]. Le premier algorithme de *Garbage Collection* incrémental qui se base sur une approche de compteur de références a été mis en œuvre par Rovner [1985] pour Xerox Parc. Baker, quant à lui, [1979] a mis en œuvre le premier collecteur incrémental par recopie appelé collecteur *Treadmill*. Il a amélioré la technique utilisée pour ne plus déplacer les objets en les organisant dans une liste cyclique doublement chaînée utilisant les trois couleurs de Djikstra [1976]. Les études analytiques de l'efficacité et des différents schémas utilisés pour les algorithmes incrémentaux peuvent être trouvées dans les travaux de Zorn [1990].

b- Les collecteurs *Mark and Compact*

Cet algorithme permet de supprimer les problèmes de défragmentation mémoire et d'allocation de l'algorithme de *Mark and Sweep*. Il se décompose en trois phases:

- Une phase de marquage telle que vue précédemment;
- Une phase de compactage qui rassemble tous les objets vivants (marqués) de façon contiguë. Il y a recopie des objets marqués;
- Une phase de mise à jour des pointeurs.

Les algorithmes *Mark and Compact* peuvent être classés en trois catégories selon la technique de compactage :

- Un compactage arbitraire où les cellules mémoires sont copiées sans se préoccuper de leur ordre original avant le passage du *Garbage Collector* ;
- Un compactage linéaire où les cellules mémoires adjacentes avant le passage du *Garbage Collector* sont copiées de manière à ce qu'elles le restent ;
- Un compactage en « *slide* » où les cellules mémoires sont copiées aux limites inférieures de la zone mémoire en maintenant leur ordre original.

Pour assurer le compactage, une partie des techniques utilisées dans les algorithmes de recopie ont été introduites. Certains algorithmes (*Two finger algorithms*) utilisent deux pointeurs, l'un sur le prochain espace libre et l'autre sur l'objet suivant à déplacer [Bartlett, 1989]. D'autres algorithmes (*Table based methods*) utilisent une table pour calculer les nouvelles valeurs des adresses [Knuth, 1973]. Une troisième méthode consiste à utiliser une liste d'objets et à la recopier dans l'ordre de passage (*Threaded methods*) [Fisher, 1975]. La quatrième technique consiste à ajouter un autre champ à chaque objet où sera stockée l'adresse du prochain objet à copier (*Forwarding address algorithms*) [Jonkers, 1975].

Le premier algorithme de compactage a été publié par Hart et Evans [1974] pour LISP 1.5. Cohen et Nicolau [1983] présentent une étude relative à l'efficacité et à la performance des algorithmes incrémentaux. Finalement Baecker [1972] a suggéré une méthode pour marquer, non pas des objets mais des pages virtuelles dans le but de réaliser un compactage. Il est à noter que la plupart des machines virtuelles Java pour systèmes non distribués et plusieurs autres langages de programmation comme C# utilisent un algorithme de *Garbage Collection Mark and Compact.*

c- Le collecteur générationnel

Les deux principes de localités sur lesquels est axée l'informatique sont :

- La localité spatiale : si une information est requise par un programme, il y a de grandes chances que les informations aux alentours soient également lues peu de temps après (très souvent en séquence et dans le sens des adresses croissantes).
- La localité temporelle : si un objet est présent en mémoire depuis longtemps, il y a toutes les chances pour qu'il y reste encore un bon moment. Par contre, un objet qui vient de naître a d'énormes risques de disparaître très rapidement.

En effet, selon plusieurs études [Zorn, 1989] [Appel, 1992] [Wilson, 1994], 80 à 90 % des cellules récemment allouées disparaissent rapidement. Par contre, un petit nombre reste. Si le système d'exploitation joue sur le premier principe, c'est sur le deuxième que joue le *Garbage Collector* pour rentabiliser son travail.

Un collecteur par recopie va passer son temps à essayer de récupérer de la place pour les cellules qui disparaissent tout de suite en recopiant sans arrêt les cellules qui restent plus longtemps en mémoire. En d'autres termes, le collecteur copiera et recopiera les objets à long cycle de vie à chaque collecte. L'idée est de regrouper les objets par âge ou par génération. Par la suite, le collecteur n'aura qu'à collecter les objets des générations « jeunes » en premier lieu avant de collecter les plus « vieilles ».

Dans cette approche, l'espace mémoire est divisé en deux ou plusieurs parties qui contiendront chacune une génération d'où l'appellation de collecteurs générationnels ou même de *l'algorithme des trains* (chaque wagon représente une génération). L'introduction des générations a eu plusieurs répercussions positives sur le déroulement du travail du collecteur puisqu'il collectera les jeunes générations en priorité et n'aura pas à collecter les autres générations que si la collecte n'a pas été assez riche. Dans ce type de collecteur, on stocke dans des zones différentes les objets récemment créés ou « jeunes » et les objets créés antérieurement dits « vieux ». Si après plusieurs collectes un objet jeune survit encore, il sera promu à une génération supérieure et il sera alors copié dans une autre génération. Le nombre d'itérations d'un collecteur qui permet à un objet de passer d'une génération à une autre plus vieille est appelé *seuil de promotion.* Il n'est pas rare aussi de voir des collecteurs qui réservent une zone mémoire particulière pour les objets immortels ou ceux volumineux afin d'éviter des collectes inutiles. Concernant le seuil de promotion, choisir la valeur « 1 » est tentant puisque cela ne nécessite pas de se souvenir de l'âge de chaque objet, ce qui mène à une occupation mémoire moins importante. Cependant, cette valeur amorce un processus de promotion d'objets très « jeunes » dont l'espérance de vie n'est pas encore bien définie. Le seuil de promotion, le nombre de générations et la taille de chaque génération sont tous paramétrables à l'amorçage du collecteur. Nous pouvons passer d'un simple algorithme par recopie à un algorithme générationnel qui utilise un algorithme différent pour chaque génération.

À la Figure 2-6, l'algorithme évolue avec quatre générations. Après le passage du collecteur, les objets en pointillés de la première génération sont promus vers la deuxième génération et le collecteur n'a pas besoin de collecter la deuxième, troisième ou quatrième génération. Il y a mise à jour des références à chaque passage du collecteur. Lors de la promotion des objets de la première à la deuxième génération, il y a

compactage et cela se fait comme si la première et la deuxième génération représentaient les deux demi-espaces d'un collecteur par recopie.

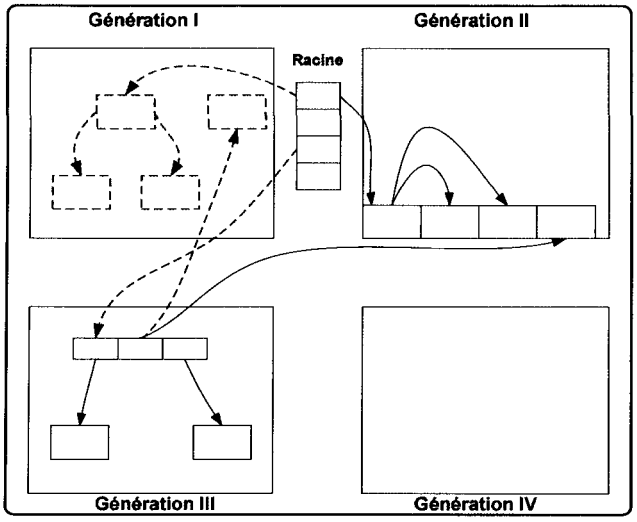

Figure 2-6 : Collecteur générationnel

Le problème principal de ce type de *Garbage Collector* se présente lors de l'allocation des gros objets qui ont une durée de vie importante. En effet, s'il est impossible, malgré leur âge élevé, de placer ces objets en vieille génération, ceux-ci resteront dans la jeune génération. Il en résulte qu'ils seront copiés plusieurs fois. Par ailleurs, les petits objets qui vieillissent pourront se placer dans la vieille génération et, ceci toujours au détriment des grands.

L'autre problème avec cet algorithme est la gestion des pointeurs qui se trouvent dans la « vieille » génération et qui pointent vers des objets qui sont dans la génération « jeune ». Mais une observation permet d'affirmer que ce cas est rare [Zorn, 1989] [Appel, 1992] [Wilson, 1994]. En effet, en général ce sont les nouveaux objets qui pointent vers des objets anciens. Pourtant, de telles références existent et un *Garbage Collector* doit en tenir compte. En pratique, cela se traduit en dénombrant les vieux objets qui pointent vers les plus jeunes et qui seront scannés lors du ramassage de la mémoire jeune.

Finalement, le *Garbage Collector* générationnel peut être combiné avec le *Garbage Collector* incrémental pour donner naissance à un *Garbage Collector* générationnel

temps réel, mais le mariage n'est pas toujours facile, ni concluant [Zorn, 1989] [Jones et Lins, 1996].

La première mise en œuvre d'un algorithme générationnel remonte à 1981 par Libermann et Hewitt [1981] et cet algorithme utilisait une collection incrémentale. Caudill et Wirfs-Brock [1986] ont utilisé cette approche pour assurer la gestion implicite de la mémoire au sein de Smalltalk-80. Avec un nombre de génération différent, Hudson *et al.* [1991] et Hosking [1991] ont proposé un collecteur générationnel pour Toolkit. Il faut noter que la machine virtuelle Hotspot, machine virtuelle Java dédiée aux serveurs, utilise un algorithme générationnel à base de *Mark and Compact* [Sun Microsystems 2006]. C'est aussi un algorithme générationnel *Mark and Compact* qui est utilisé par le Framework .Net de Microsoft [Rammer et Szpuszta, 2005]. L'avantage avec celui de Hotspot est la possibilité de paramétrer la taille des générations, leurs nombres, le seuil de promotion et aussi la fréquence des collectes au sein des générations.

2.1.2.4 Conclusion

Il existe d'autres techniques de *Garbage Collection* qui n'ont pas été détaillées et qui sont dédiées, pour la plupart, à des langages orientés objet. Les algorithmes conservatifs [Bartlett, 1989] [Boehm et Weiser 1988] sont un autre type d'algorithmes qui fonctionnent avec le principe « dans le doute s'abstenir », préférant ne pas avoir d'interactions avec des zones de mémoire ambiguës (on ne peut pas déterminer si ce sont des *Garbage* ou non : on ne peut pas différencier un pointeur et une cellule mémoire contenant des données), le *Garbage Collector* les laisse de côté.

D'autres techniques de *Garbage Collection* ont été aussi utilisées dans d'autres disciplines à savoir les bases de données et les systèmes d'exploitation [Jones et Lins, 1996]. D'un autre côté, les chercheurs ont axé leurs recherches depuis une bonne vingtaine d'années sur les *Garbage Collector* dédiés aux systèmes distribués qu'on verra avec plus de détails dans la section 2.2 de ce mémoire. D'autres travaux apparentés aux *Garbage Collector* ont été menés par Boehm et Weiser [1988] concernant l'intégration d'un *Garbage Collector* aux langages de programmation C et C++ et par Zorn [1989] et Williams [1991] concernant des études de mémoire cache.

Le Tableau 2 présente certains langages de programmation ou plateformes avec leur *Garbage Collector* associé. Il peut exister d'autres collecteurs pour ces mêmes langages comme c'est le cas de Python qui peut évoluer aussi avec un algorithme de compteurs de références.

Tableau 2 : *Garbage Collector* de certains langages

Langages	Techniques de *Garbage Collection*
J2EE Java Hotspot	Générationnel (Mark and Compact par défaut pour toutes les générations)
J2SE	Générationnel Mark and Compact
Java Kilo Virtual Machine KVM	Mark and Compact
C#	Générationnel Mark and Compact
Visual Prolog 6	Conservatif Boehm - Weiser
Modula III	Compteur de référence incrémental
Eiffel	Générationnel Mark and Compact
Lisp 2	Mark and Compact
Smalltalk	Générationnel Mark and Compact
Python	Conservatif Boehm - Weiser
OZ	Générationnel Mark and Compact
Scheme	Recopie
Haskell	Générationnel
Toolkit	Générationnel
Erlang	Incrémental

L'appel des *Garbage Collector* se fait de manière automatique au sein de la majorité des langages de programmation qui l'utilisent, mais il est possible de forcer l'appel du *Garbage Collector* en invoquant, par exemple, la méthode `System.gc()` pour Java ou encore la méthode `System.GC.Collect ()` avec .NET. Il est même possible de choisir la génération à collecter avec .NET en surchargeant la même méthode `System.GC.Collect (n)` où *n* est la génération à collecter. Cependant, il n'est pas conseillé d'invoquer le *Garbage Collector* explicitement trop souvent, car cela pourrait,

d'une part, dégrader les performances des applications et, d'autre part, entraîner les mêmes problèmes qu'une gestion explicite de mémoire [Jones et Lins, 1996].

2.1.3. Gestion mémoire explicite

Nous avons vu, dans la partie précédente, différentes techniques de *Garbage Collection* qui assuraient une gestion implicite de la mémoire, ce qui est fort avantageux pour des applications très complexes [Meyer, 1988] [Jones et Lins, 1996]. Nous avons vu aussi que la libération implicite de la mémoire via le *Garbage Collector* avait de nombreux inconvénients liés à d'importants coûts temporels parfois. Certains langages comme C, C++, Ada, Pascal et bien d'autres utilisent une libération explicite. L'avantage majeur de cette libération par rapport au *Garbage Collector* est sans doute l'emploi des pointeurs, outil très important permettant un accès direct à certaines zones mémoires et une utilisation plus optimale de l'espace mémoire (une utilisation de pointeurs serait plus avantageuse qu'une utilisation de tableaux : on alloue de l'espace à la demande ; il n'y aura donc pas d'espace alloué inutilement). Outre les objets, les pointeurs peuvent pointer des variables de différents types. L'allocation se fait de manière dynamique et c'est au programmeur de libérer l'espace alloué.

En C/C^{++} par exemple, il est possible d'allouer des variables de manière dynamique. L'allocation dynamique permet d'allouer des variables au moment de l'exécution du programme pour tenir compte d'événements non déterministes [Meyer, 1988], puisque nous ne savons pas à l'avance s'il y aura vraiment utilisation de cet espace mémoire à l'avance. Ainsi, pour une application quelconque, on pourra réserver une zone mémoire dont la taille est suffisante pour contenir les variables déclarées dans cette application. Un certain nombre de fonctions permettent d'allouer de la mémoire pour ces besoins. On utilise l'opérateur *sizeof* pour déterminer l'espace mémoire occupé par des variables d'un type donné. La fonction *malloc()* permet d'allouer un bloc de taille donnée. La fonction retourne un pointeur qui désigne le début de la zone. Si le programme ne dispose pas d'assez d'espace mémoire, la fonction retournera la valeur 0 qui est en réalité le pointeur *NULL*. Lorsqu'on a terminé d'utiliser l'objet ou la variable allouée, il ne faut pas oublier de libérer l'espace mémoire qui a été utilisé avec la fonction. On donne la valeur du pointeur retourné par la fonction *malloc()* à la fonction *free()*. Toujours dans l'exemple de

C ou C++, d'autres fonctions traitant de l'allocation de la mémoire existent aussi dans le fichier d'entête *stdlib.h*. On peut citer *calloc()* qui retourne un pointeur sur un espace mémoire ou bien *NULL* si cette demande ne peut pas être satisfaite. La mémoire allouée dans ce cas est initialisée par des zéros. La fonction *realloc()*, quant à elle, change la taille de l'objet pointé.

Concernant la programmation orientée objet pour la gestion des cycles de vie des objets, on utilise ce qu'on appelle des constructeurs et des destructeurs. Ce sont deux méthodes particulières qui sont appelées respectivement à la création et à la destruction d'un objet. Toute classe a un constructeur et un destructeur par défaut fourni par le compilateur. Ces constructeurs et destructeurs appellent les constructeurs par défaut et les destructeurs des classes de base et des données membres de la classe. Il est donc souvent nécessaire de les redéfinir afin de gérer certaines actions qui doivent avoir lieu lors de la création d'un objet et de leur destruction. Par exemple, si l'objet doit contenir des variables allouées dynamiquement, il faut leur réserver de la mémoire à la création de l'objet ou, au moins, mettre les pointeurs correspondants à *NULL*. À la destruction de l'objet, il convient de restituer la mémoire allouée par les fonctions citées plus haut, s'il en a été alloué. On peut trouver bien d'autres situations où une phase d'initialisation et une phase de terminaison sont nécessaires.

Dès qu'un constructeur ou un destructeur a été défini par l'utilisateur, le compilateur ne définit plus automatiquement le constructeur ou le destructeur correspondant par défaut. En particulier, si l'utilisateur définit un constructeur prenant des paramètres, il ne sera plus possible de construire un objet simplement sans fournir les paramètres à ce constructeur, à moins, bien entendu, de définir également un constructeur qui ne prend pas de paramètres.

Le constructeur est appelé après l'allocation de la mémoire de l'objet et le destructeur est appelé avant la libération de cette mémoire. La gestion de l'allocation dynamique de mémoire avec les classes est ainsi simplifiée. Dans le cas des tableaux, l'ordre de construction est celui des adresses croissantes et l'ordre de destruction est celui des adresses décroissantes. C'est dans cet ordre que les constructeurs et destructeurs de chaque élément du tableau sont appelés. Les constructeurs pourront avoir des paramètres. Ils peuvent donc être surchargés, mais pas les destructeurs. Cela est dû au fait qu'en

général, on connaît le contexte dans lequel un objet est créé, mais qu'on ne peut pas connaître le contexte dans lequel il est détruit. Il ne peut donc y avoir qu'un seul destructeur.

2.1.4. Conclusion

L'utilisation d'un algorithme de *Garbage Collection* pour gérer la gestion de la mémoire permettait d'avoir de meilleures performances qu'une gestion explicite [Zorn, 1989]. Nous avons vu que le cycle de vie d'un objet local passe par deux étapes, à savoir la création et la destruction. Si la création ne présente aucune ambiguïté, la destruction de l'objet présente certaines difficultés, notamment avec la finalisation du cycle de vie de l'objet. La gestion explicite, à l'égard de C++ et de bien d'autres langages de programmation usant de cette approche, permet de mettre terme à l'existence d'un objet à n'importe quel moment, qu'il soit bon ou mauvais. La gestion implicite, quant à elle, par le biais du *Garbage Collector* est un moyen très habile pour mettre fin à la vie d'un objet malgré certains défauts. Le développeur, dans des environnements comme Java ou Modula III, n'a pas à se soucier de la libération de la mémoire, c'est le *Garbage Collector* qui s'occupe de cela. Le choix d'une approche explicite ou implicite pour la gestion de la mémoire semble très difficile, car chaque approche a ses avantages et ses inconvénients. Cependant, dans des milieux distribués, avec des applications très complexes et plusieurs clients et serveurs à gérer, une approche implicite est conseillée [Zorn, 1989] [Jones et Lins, 1996].

2.2 Gestion du cycle de vie des objets distribués

2.2.1. Introduction

Nous nous sommes intéressés dans la section précédente au cycle de vie des objets en systèmes non distribués. Nous allons, dans cette section, détailler les principales techniques de gestion de cycle de vie d'objets distribués. Nous allons pour cela étudier le cycle de vie d'objets sur des plateformes différentes, à savoir JINI, CORBA et .Net à titre d'exemples. Avant de détailler le cycle de vie des objets au sein de ces trois plateformes, nous allons faire une description non exhaustive des principaux algorithmes de *Garbage*

Collection dédiés aux systèmes distribués ainsi que de la technique de *leasing* qui permet de remplacer les *Garbage Collector* en milieux distribués [Rammer et Szpuszta, 2005]. L'utilisation d'une gestion de mémoire explicite en milieu distribué étant liée à la complexité de ces milieux [Tanembaum et Van Steen, 2002], il serait donc plus adéquat de laisser le langage ou la plateforme s'occuper de la gestion mémoire.

Un système distribué est un système qui s'exécute sur un ensemble de machines sans mémoires partagées, à l'inverse des systèmes parallèles. Seulement, l'utilisateur voit ce système comme une seule machine au point que la défaillance d'une machine dont l'utilisateur ignore l'existence peut rendre sa propre machine inutilisable. Si un processeur de l'une des machines connectées est en panne, un autre peut prendre en charge l'exécution. On parle alors de machines uniprocesseur virtuelles.

C'est naturellement les applications de types clients/serveur qui sont encouragées, et ce type d'application connaît un nouvel essor avec les systèmes distribués. Un système distribué est souvent appelé *middleware* du fait qu'il joue le rôle d'interface entre les applications et les machines.

2.2.2. *Garbage Collector* distribués

Comme les algorithmes vus dans la section précédente, les *Garbage Collector* distribués doivent assurer une collecte d'objets qui ne sont d'aucune nécessité pour l'application et ne pas collecter les objets qui peuvent être encore utilisés par l'application. Seulement, dans les milieux distribués, il ne s'agit plus d'un seul espace mémoire dédié à une seule machine, mais d'un espace mémoire partagé par toutes les machines du *middleware*. De plus, dans un tel milieu, les racines ne se situent plus sur une seule machine cliente, mais sur plusieurs machines et la constitution du graphe d'objets se fera à partir de plusieurs machines. Ce sont principalement les mêmes techniques de *Garbage Collection* utilisées en milieu local qui sont utilisées en milieu distribué. Cependant, dans les systèmes distribués, il y a une différenciation sur le plan du type d'objet qui peut être passif, actif ou un acteur et c'est la dépendance de l'objet avec le système qui est à la base de cette différenciation [Jones et Lins, 1996]. Un objet passif comporte des données, mais son *thread* est extérieur contrairement à un objet actif. Les

acteurs sont des objets actifs qui possèdent un système de messagerie et qui peuvent communiquer directement avec d'autres objets [Hewitt, 1977].

L'un des premiers *Garbage Collector* distribués est celui de Hudak et Keller [1982]. Ils ont fait la conception d'un algorithme se basant sur la technique de *Garbage Collection* à la volée de Djikstra [1976] et ont permis de paralléliser la collecte et le déroulement de l'application. Ce premier algorithme était un *Mark and Sweep* qui établissait un graphe d'objets à partir de toutes les machines. D'autres *Mark and Sweep* ont suivi, notamment, celui de Mohamed-Ali [1984] qui permettait à chaque processeur de faire la collecte de son propre espace mémoire. L'algorithme de Hughes [1985] est basé sur celui de Mohamed-Ali [1984]. L'algorithme de ce dernier n'était pas vraiment un algorithme temps réel puisque le *Garbage Collector* était bloqué lors de la phase de l'allocation, et possédait en plus de cela une horloge globale qui lui procurait une fiabilité sur le plan de la communication. L'algorithme de Vestal [1987] est, quant à lui, basé sur celui de Djikstra [1976] et assure la parallélisation des deux phases de marquage et de balayage. Les deux plateformes orientées objet Emerald [Hutchinson, 1987] et IK [Sousa, 1993] utilisaient aussi l'algorithme de Djikstra [1976].

Des algorithmes de recopie ont été aussi conçus pour les systèmes distribués. Le premier a été celui de Rudalics [1986] qui a utilisé les deux demi-espaces. Lieberman et Hewitt [1983] d'un côté et Plainfossé et Shapiro [1992] d'un autre, ont utilisé des listes pour assurer la collecte en partant d'un ensemble de racines globales à toutes les machines [Abdullahi et Ringwood, 1998] [Jones et Lins, 1996].

La technique de *Mark and Sweep* et celle de recopie n'ayant pas répondu aux exigences recherchées [Jones et Lins, 1996], la recherche s'est orientée vers l'élaboration d'algorithmes de *Garbage Collection* se basant sur la technique de compteur de références qui est très attrayante pour des architectures faiblement couplées [Tanembaum et Van Steen, 2002]. Le premier algorithme de *Garbage Collection* se basant sur les compteurs de références a été décrit par Nori [1979]. Lermen et Maurer [1986] ont créé un protocole qui permettait de réduire le nombre de messages pour l'incrémentation et la décrémentation des compteurs de références des objets. En 1991, en utilisant l'approche de Djikstra [1976], Piqué [1991] a proposé un *Garbage Collector* basé sur la technique des compteurs de références en ajoutant deux autres champs à chaque objet. La technique

a été appelée compteurs de références indirects. Son algorithme constituait un graphe d'objets et le transformait en arbre, les deux nouveaux champs contenaient les adresses du père et du descendant direct dans l'arbre. Lang *et al.* [1992] ont présenté un algorithme qui utilise une technique similaire pour les objets globaux en construisant un seul arbre pour tous les objets. En 1989, un algorithme générationnel, se basant sur la technique des compteurs de références, a été utilisé par Goldberg [1991] pour un système multiprocesseur. Moreau [1998] a proposé un algorithme similaire à celui de Piqué [1991], mais qui a assuré aussi la collecte des objets mobiles. D'autres techniques ont été utilisées, notamment dans le cadre de l'élaboration d'outils de gestion de mémoire explicite performants et surtout avec l'augmentation de la complexité des systèmes distribués et pour les nécessités technologiques du juste à temps [Abdullahi et Ringwood, 1998] [Tanembaum et Van Steen, 2002]. Ces différentes techniques ont été utilisées par des plateformes dédiées aux systèmes distribués pour libérer l'espace mémoire occupé inutilement et aussi pour gérer l'allocation de la mémoire dans la mémoire partagée.

2.2.3. Le *Leasing*

Le *leasing* permet à des objets présents sur des sites clients distincts de négocier et d'obtenir la réservation de ressources de toutes sortes. Le processus de *leasing* prévoit les conditions du maintien et de la suppression de la réservation de ressource, y compris en cas de blocage du système distribué (panne d'un élément...). Sur un système Java non distribué, un objet instancié est conservé en mémoire tant que subsiste une référence pointant vers lui. À la suppression de la dernière référence, le *Garbage Collector* se charge de libérer l'espace mémoire qu'occupait l'objet devenu inutile. Dans le cadre d'un système distribué, un tel procédé n'est pas applicable. Si une entité réserve, par exemple, une ressource rare sur un système distant, puis subit une panne qui la rend hors service, le système distant conserve la réservation indéfiniment, croyant la ressource employée. La ressource est bloquée sans espoir d'être libérée pour un usage plus productif.

Le *leasing* s'appuie sur le principe suivant : au lieu d'allouer une ressource jusqu'à ce que le requérant déclare explicitement ne plus en avoir besoin, on alloue la ressource pour une période donnée appelée bail. Ce bail peut être négocié entre les deux parties, ou imposé par le gestionnaire de la ressource. Bien entendu, il peut être renouvelé *ad*

libitum, si les circonstances le permettent. Les négociations peuvent porter sur la durée du bail, son renouvellement éventuel ou encore sur son arrêt prématuré. En cas de problème, la ressource n'est bloquée que le temps du bail et redevient disponible à l'expiration de celui-ci. De plus, le modèle de *leasing* permet d'éviter l'accumulation des ressources réservées sur un serveur, mais non utilisées. En effet, dans un système traditionnel, certaines ressources peuvent être réservées puis « oubliées » par des clients; avec le système de *leasing*, la nécessité de renouveler régulièrement le bail impose au client de sélectionner avec plus de discernement les ressources qu'il juge nécessaires.

Pour la compréhension du modèle, un bail (*lease*) présente les caractéristiques essentielles suivantes :

- Un bail est une période durant laquelle le fournisseur du bail garantit au demandeur l'accès à une certaine ressource. La durée peut en être définie par le fournisseur ou négociée par les deux parties.
- Pendant la durée du bail, celui-ci peut être annulé par le demandeur. Les ressources en jeu sont alors libérées.
- Un demandeur peut demander la reconduction du bail pour une durée qui peut différer de la précédente.
- Un bail peut expirer. S'il n'est pas renouvelé, les ressources impliquées sont libérées.

Ainsi, lors du renouvellement du bail, la valeur du nouveau bail peut être calculée par la formule suivante :

```
valeur bail = MAX(bail actuel - temps expiré, renouvellement du bail)
```

La valeur du bail sera alors égale au maximum entre le bail actuel moins le temps expiré de ce bail et la durée du bail demandé. Le renouvellement du bail peut être effectué par un client différent de celui qui détient l'objet. D'un autre côté, la négociation du bail se fait directement entre le client et le détenteur de l'objet distant, mais c'est le détenteur qui impose la durée du bail s'il surpasse une certaine limite. Ces caractéristiques d'attribution et de renouvellement de bail ont permis à la plateforme JINI de s'imposer dans le monde des plateformes distribuées [Tanembaum et Van Steen, 2002].

Soit deux clients, Client1 et Client2. Les deux objets A et B qu'ils demandent sont fournis par un troisième client appelé le Client3. Le Client1 demande l'objet A avec un bail de 10 ms et le Client2 demande un bail de 5 ms tel qu'illustré à la Figure 2-7.

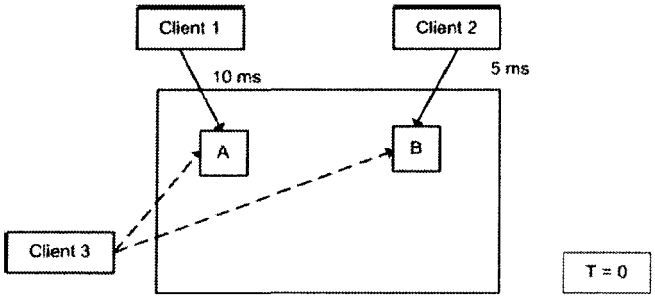

Figure 2-7 : *Leasing* et demande de baux

Dans un second temps, le Client2 va s'apercevoir qu'il a encore besoin de l'objet B et il relance alors une autre demande. Dès que le bail se termine, l'objet est aussitôt libéré et supprimé. C'est le cas de l'objet B à la Figure 2-8 après 7 ms, car il reste encore 3 ms pour Client1. Supposons qu'à T = 8 ms, le Client1 n'a plus besoin de l'objet A. Cet objet persistera encore 2 ms dans l'espace réservé. Revenons maintenant à la Figure 2-8 et imaginons que le Client1 tombe en panne à T = 2ms. L'objet A qu'il a demandé persistera en mémoire inutilement pendant 6 ms et entraînera de sérieux problèmes de mémoire en cas de répétition.

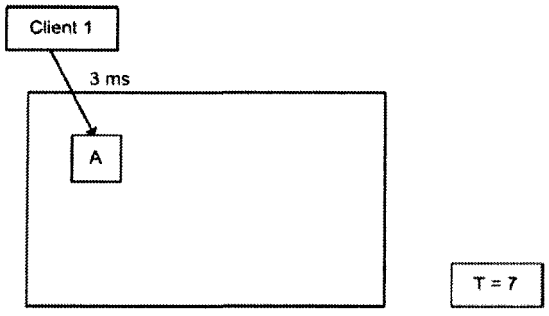

Figure 2-8 : Persistance des objets après utilisation

Le principal problème avec l'utilisation des baux pour la gestion de la mémoire est le fait qu'un client ne peut jamais faire une demande exacte d'une ressource. En effet,

cette demande s'appuie sur des heuristiques probabilistes faites par le client comme le calcul se basant sur les moyennes d'utilisation et de saturation [Ilango, 2001]. Le client ne peut fournir ainsi qu'une estimation du temps dont il a besoin pour l'utilisation de la ressource et il y a surestimation dans la plupart des cas [Ilango, 2001]. Outre l'utilisation non optimale des baux, la gestion des contrats pour tous les clients, le trafic pour les négocier et les renouveler peut s'avérer très coûteux pour le système et le ralentir considérablement [Ilango, 2001]. Nous pouvons dire finalement que puisque les baux ont une durée déterminée, ils ne conviennent pas dans les cas où il faudrait conserver des objets pendant de longues durées dans le système.

Pour la résolution de ces problèmes, plusieurs améliorations ont été effectuées non seulement par rapport au nombre de messages générés, mais aussi pour remédier à la surestimation des baux comme nous allons voir dans la technique de *leasing* utilisée par la plateforme .Net [Rammer et Szpuszta, 2005]. D'autre part, la technique de *leasing* peut être hybridée avec un *Garbage Collector.* En effet, le *Garbage Collector* ne fait que collecter les objets qui ont été libérés après que leur bail respectif soit épuisé et qu'aucun autre client n'ait renouvelé ce bail. L'approche de leasing a aussi été utilisée par RMI [Pitt et McNiff, 2001], JINI [Tanembaum et Van Steen, 2002], DCOM de Microsoft [Frank, 1997] et CORBA [OMG, 1995].

2.2.4. CORBA

CORBA [OMG, 1995], ou le *Common Object Request Architecture*, est une plateforme qui représente un bus d'objets répartis prenant en charge les communications entre les différents objets. En d'autres termes, CORBA propose un modèle client/serveur de coopération entre des objets répartis. Pour la description des objets distribués et leurs interactions, CORBA spécifie deux plateformes indépendantes : un modèle objet (*Object Model*) qui définit et décrit les interfaces des objets distribués sur un environnement distribué et un modèle de référence (*Reference Model*) qui caractérise les interactions entre ces objets. Le modèle d'objet définit un objet comme étant une *entité encapsulée* avec une *identité distincte et immutable*, les services associés à cette entité sont accessibles par ce qu'on appelle des *interfaces*. Le modèle de référence fournit les catégories des différentes interfaces. Toutes ces catégories sont liées conceptuellement

par un *Object Request Broker* (ORB). Généralement, un ORB assure une communication entre les clients et les objets et une transparence lors de l'activation des objets par les clients. Les interfaces sont décrites par l'intermédiaire d'un langage appelé OMG – *Interface Definition Langage* ou plus simplement IDL dans le but est de fournir le nom, les méthodes et les attributs de chaque objet appelé en anglais *servant* [OMG, 1995].

CORBA a donc essentiellement été conçu pour résoudre le problème d'hétérogénéité des langages. Il sait exploiter les principales qualités du langage Java : portabilité, sécurité et facilité d'emploi tout en révolutionnant son système d'invocation à distance Java RMI. D'autre part, CORBA comporte des services objet communs (*CORBAservices*) qui fournissent sous forme d'objets CORBA, spécifiés grâce au langage IDL, les fonctions systèmes nécessaires à la plupart des applications réparties. Ils contribuent à assurer l'interopérabilité entre les diverses implémentations et une compatibilité avec beaucoup de technologies existantes. Parmi les services CORBA, nous pouvons citer le service d'appellation (*Naming Service*), le service d'événement (*Event Service*), le service de sécurité (*Security Service*), le service de contrôle (*Control Service*), le service de transaction (*Transaction Service*), le service de cycle de vie (*Lyfe Cycle Service*), etc. Les services CORBA sont un ensemble de recommandations et non de spécifications. Chaque service assure une tâche, mais tous les services interagissent entre eux via l'ORB [Henning et Vinoski, 2004].

Le service de cycle de vie s'occupe de la création, de la copie, du déplacement et de la destruction des objets CORBA qui peuvent être en local ou sur des sites distants. C'est l'utilisateur qui devra implémenter ces quatre opérations. Au sein de la plateforme CORBA, la création des objets via le service de cycle de vie se fait grâce au modèle des « usines » (*Factory*). Une *factory* est un objet CORBA qui offre une ou plusieurs opérations pour créer d'autres objets [Henning et Vinoski, 2004]. Lors de la création d'un objet, le client lance une requête pour une opération dans une *factory*. L'implémentation de cette opération crée un nouvel objet et retourne sa référence au client qui a lancé la requête. Outre la création, le service de cycle de vie permet aux objets d'être déplacés d'un site à un autre. Seulement, le déplacement des objets va à l'encontre de la philosophie de CORBA : migrer un objet d'un emplacement à une autre n'a aucun sens du point de vue conceptuel puisqu'un client peut accéder à un objet où qu'il soit sur des

sites accessibles. De plus, le service de cycle de vie d'objet a d'autres inconvénients comme ses insuffisances au point de vue de la sécurité des données en cause. Puisqu'il s'agit de l'un des premiers services élaborés pour CORBA, il est donc conseillé d'utiliser ce service lors d'implémentation des applications à petite échelle de distribution [Henning et Vinoski, 2004].

Il existe une autre technique permettant la gestion du cycle de vie des objets sur la plateforme CORBA sans passer par le service de cycle de vie d'objet et c'est un utilitaire CORBA qui permet de remédier aux inconvénients rencontrés avec ce dernier. Il s'agit de la technique de « l'expulseur » (*Evictor Pattern*) qui est une stratégie limitant la consommation en termes de mémoires. À chaque instanciation d'un *servant*, le manager des *servants* vérifie le nombre de *servants* existants et, si ce nombre est inférieur à une limite définie par le système, le nouveau *servant* sera créé. Sinon, le manager en expulse un autre déjà existant pour allouer de l'espace à celui nouvellement créé.

Le choix du *servant* à expulser peut se faire selon plusieurs stratégies comme celle du plus ancien utilisé (*Least Recently Used*), celle du moins utilisé (*Least Frequently Used*), celle avec la plus grosse taille mémoire ou simplement de façon aléatoire. Généralement, la stratégie de choix est celle du plus ancien utilisé [Henning et Vinoski, 2004]. En présence d'un éjecteur, l'instanciation d'un servant se déroule en quatre étapes : d'abord, le client invoque une opération, ensuite, le POA (*Portable Object Adapter*), qui est une interface entre les interfaces et l'ORB assurant la portabilité des applications d'un ORB à un autre [Henning et Vinoski, 2004], vérifie au sein du *locateur des servant*[1] si cette opération est existante ou non. Puis, le manager procède à l'instanciation du *servant*. Enfin le manager vérifie la file de l'expulseur[2] : si elle n'est pas pleine, le servant sera ajouté, sinon l'expulseur appliquera l'une des techniques (LRU, LFU, etc.) pour expulser l'une des interfaces déjà existantes [Henning et Vinoski, 2004]. Le *servant* élu pour la suppression se trouve en tête de file, de même les *servants* sont ordonnés dans la file dans l'ordre de prochaine suppression. À la Figure 2-9, l'ajout

[1] C'est une interface qui assure de retrouver des servant en collaboration avec le POA [OMG, 1997].

[2] Il est aussi possible d'utiliser une liste.

du servant S6 a supprimé le servant qui est à la tête de la file de l'expulseur, le servant S1.

Il existe une troisième alternative pour assurer cette gestion : la *Garbage Collection.* Il s'agit du même principe que tout autre *Garbage Collector* : collecter des ressources utilisées par les objets, mais sans aucune utilité pour les clients. L'utilisation du *Garbage Collector* au sein de la plateforme CORBA est justifiée quand la méthode *remove()* est appelée et ceci peut se réaliser dans les cas suivants :

- L'application néglige l'appel de cette méthode;
- Le client tombe en panne et, alors, cette méthode ne sera jamais appelée;
- Il y a un problème de connexion du réseau, la méthode est envoyée, mais elle n'arrivera jamais à destination;
- Une erreur peut survenir dans l'environnement et peut faire oublier au client d'appeler cette méthode;
- Un crash du serveur peut arriver (un *run out of memory* par exemple);

C'est pour ces principaux problèmes qui peuvent arriver très fréquemment [Tanenbaum et Van Steen, 2002][Henning et Vinoski, 2004] qu'un *Garbage Collector est* conseillé au sein de la plateforme CORBA. À cet effet, il existe plusieurs techniques de *Garbage Collection* qui peuvent être utilisées par un ORB pour gérer le cycle de vie des objets. Nous pouvons citer le *Garbage Collector* par arrêt système ou celui utilisant un expulseur ou encore le *leasing* [Henning et Vinoski, 2004]. Il est aussi possible d'hybrider plusieurs techniques.

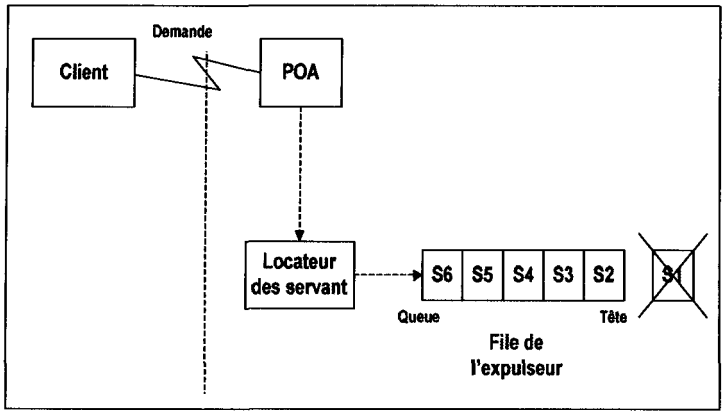

Figure 2-9 : Technique d'expulseur CORBA

Comme nous avons pu le voir, il existe plusieurs méthodes pour gérer le cycle de vie au sein de la plateforme CORBA. Selon Henning et Vinoski [2004], la technique de l'expulseur est la plus prometteuse. Ils attribuent une certaine difficulté associée à l'implémentation des autres méthodes.

2.2.5. .NET

.Net (DotNet en anglais) est un ensemble de technologies de logiciel Microsoft pour connecter de l'information, des personnes, des systèmes et différents appareils mobiles comme les téléphones mobiles ou encore les *palms*. .Net est une technologie récente (mis sur le marché en janvier 2002), adaptée au développement d'Internet et qui s'appuié sur la technologie XML. Elle permet également à Microsoft de concurrencer Sun et le langage Java, notamment EJB ou JINI. .Net se décompose en serveurs, clients et services WEB XML. Elle offre également aux développeurs une infrastructure de développement pour la conception et la réalisation d'application .Net : le Framework .Net. .Net étant la vue d'ensemble des technologies, le Framework .Net est le cœur de .Net permettant aux développeurs de concevoir des applications .Net.

En fait, toutes les applications .Net (ainsi que les pages Web ASP .Net) sont compilées dans un langage intermédiaire, le *Microsoft Intermediate Language* (MSIL), qui ne dépend pas du processeur et qui, par son code binaire, rappelle le fonctionnement de la machine virtuelle Java. La compilation en code binaire est effectuée *just in time* lors de la première exécution du code par le CLR (*Commun Language Runtime*). Le CLR gère l'exécution du code et procure des services qui simplifient le processus de développement comme la gestion automatique de la mémoire, la gestion de la sécurité et les autorisations d'exécution.

Pour la gestion automatique de la mémoire, partie intégrante du service de durée de vie du CLR, .Net utilise un *Garbage Collector* générationnel évoluant avec l'approche *Mark and Compact* pour toutes ses générations. Pour optimiser les performances de ce *Garbage Collector*, la pile est divisée en trois générations numérotées 0, 1 et 2. L'algorithme de *Garbage Collection* du CLR se base sur plusieurs généralisations que l'industrie des logiciels a observées, en essayant divers schémas de *Garbage Collection*. En premier lieu, il est plus rapide de compacter la mémoire pour une partie de la pile que

pour la pile toute entière. Deuxièmement, les nouveaux objets ont des durées de vie plus courtes que les objets plus anciens. Enfin, les nouveaux objets sont fréquemment liés entre eux et l'application y accède à peu près au même moment [Rammer et Szpuszta, 2005].

Seulement, cette technique de gestion automatique de la mémoire se transforme dès qu'on passe à une architecture distribuée. En effet, en milieu distribué, la gestion de mémoire par *Garbage Collector* passe à une gestion de mémoire qui se base sur le *leasing*. .Net ne fait appel au *leasing* que lorsqu'il s'agit de gestion des cycles de vie d'objets à distance. Les deux approches *Mark and Compact* générationnel et *leasing* cohabiteront ensemble : le *Garbage Collector* collectera les objets dans le même domaine d'application (les objets se trouvent au sein du même client ou du même serveur) appelé AppDomain [Rammer et Szpuszta, 2005] et le *leasing* se chargera de libérer l'espace mémoire occupé par les objets à distance, sachant que ces objets à distance ne sont pas les objets originaux demandés par les clients, mais seulement des copies mises en mémoire partagée. Pour gérer l'interaction entre ces différents objets à distance (demande d'objet, objets originaux et objets copies), .Net offre une infrastructure appelée *.Net Remoting* [Rammer et Szpuszta, 2005]. En plus de la gestion de l'interaction des objets, cette dernière offre plusieurs services parmi lesquels on retrouve la prise en charge de l'activation et des durées de vie des objets distants via les baux.

Contrairement à ce qui se passe avec la plateforme JINI, .Net fait intervenir un troisième tiers (Figure 2-10), outre le client et le site détenteur de l'objet demandé, pour la gestion des baux qu'on appelle plus usuellement sponsor. Le sponsor s'occupe du renouvellement des baux lorsqu'ils ont expiré. Généralement, il existe un sponsor pour chaque client ce qui fait qu'un sponsor administre plusieurs objets en même temps. Dans le cas de JINI, lorsque le bail d'un objet arrive à terme et qu'aucun autre bail n'a été effectué, l'objet est amené à être détruit et sa mémoire devra être libérée.

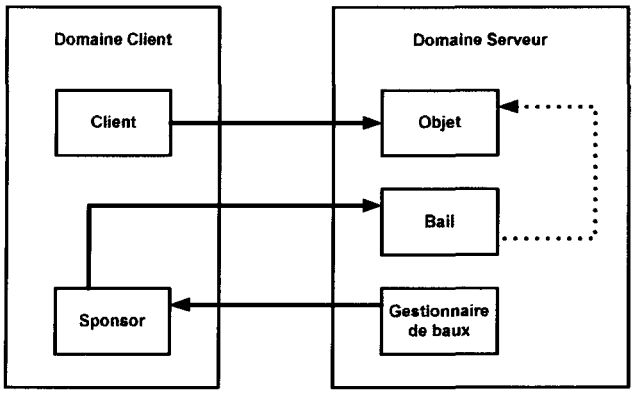

Figure 2-10 : Architecture Leasing de .NET

Sur la plateforme .Net, le sponsor intervient avant la libération de l'espace et envoie des messages aux différents clients leur demandant s'ils auraient besoin de cet objet dans un futur proche (ce futur proche est une date par défaut égale à 15 minutes sachant que cette date est paramétrable au besoin). Si un client « sponsorise » cet objet, c'est-à-dire qu'il envoie une réponse au sponsor lui indiquant qu'il en aura besoin (les clients qui n'ont pas besoin de cet objet ne répondront pas), alors cet objet ne sera pas détruit. Par contre, si aucun client ne « sponsorise » cet objet, il sera alors détruit (on sous-entend par destruction la collecte de cet objet). Concernant la gestion des baux, chaque domaine d'application contient un gestionnaire de bail qui est chargé d'administrer les baux dans ce même domaine. Tous ces baux sont examinés régulièrement afin de détecter les durées des baux expirés. Si un bail a expiré, les clients qui utilisaient cet objet sont contactés afin de leur donner la possibilité de renouveler le bail. Si aucun de ces clients ne décide de renouveler ce bail, le gestionnaire de bail parcourt sa liste de sponsors pour cet objet et interroge la liste pour savoir si l'un des sponsors souhaite renouveler le bail. Si aucun sponsor ne renouvelle le bail, le gestionnaire de bail supprime le bail, l'objet est supprimé et sa mémoire est récupérée par le *Garbage Collector*. De même, si un sponsor ne répond pas dans un intervalle de temps spécifié, il est supprimé de la liste des sponsors de cet objet.

2.2.6. Conclusion

Une gestion de cycle de vie implicite au niveau des systèmes distribués s'avère plus efficace qu'une gestion explicite étant donné la complexité de tels systèmes. L'outil le plus utilisé à ce jour dans les langages de programmation pour assurer une gestion de cycle de vie implicite est sans doute la *Garbage Collection*, qui étend les différents algorithmes et schémas utilisés dans les systèmes non distribués pour une utilisation plus efficace. Les algorithmes générationnels ont cependant démontré de meilleurs résultats que les autres approches [Jones et lins, 1996], surtout dans le cas d'un algorithme *Mark and Compact.* Pour assurer une gestion implicite, la plateforme CORBA a utilisé la *Garbage Collection*, mais aussi d'autres techniques telles que le *leasing* ou encore l'expulseur qui n'a été utilisé qu'avec cette plateforme. .NET, la plateforme de Microsoft, pour assurer la gestion de cycle de vie, a utilisé le *leasing* en attribuant les différentes tâches de gestion des baux et de leur renouvellement à différentes parties comme le gestionnaire des baux et le sponsor.

Nous explorons, dans la section suivante, la gestion de cycle de vie explicite et implicite utilisée au sein du langage C++.

2.3 Gestion du cycle de vie des objets au sein de C++

C++ est l'un des langages de programmation les plus utilisés actuellement. Il est à la fois facile à utiliser et très efficace. Il souffre cependant de la réputation d'être compliqué et illisible. Cette réputation est en partie justifiée [Ellis et Stroustrup 1990]. La complexité du langage est inévitable lorsqu'on cherche à avoir beaucoup de fonctionnalités. En revanche, en ce qui concerne la lisibilité des programmes, tout dépend de la bonne volonté du programmeur.

Les caractéristiques du C++ en font un langage idéal pour certains types de projets. Il est incontournable dans la réalisation des grands programmes. Les optimisations des compilateurs actuels en font également un langage de prédilection pour ceux qui recherchent les performances. Enfin, ce langage est, avec le C, idéal pour ceux qui doivent assurer la portabilité des fichiers sources de leurs programmes (pas des exécutables) [Stroustrup, 2004].

Le langage C++ est une extension d'ANSI-C [Kernighan et Ritchie, 1988]. La structure d'un programme écrit en C++ est évidemment similaire à la structure d'un programme en C. On y retrouve une fonction principale, des fichiers d'en-tête et des fichiers source, et une syntaxe très proche du C. Il est souvent rapporté que le langage C++ tient sa puissance du fait qu'il est la descendance directe du langage C. D'autres affirment que cela constitue son principal handicape [Jones et Lins, 1996]. Comme tout langage de programmation, C++ aura à gérer la création et la destruction de ces variables et objets. Puisqu'il est une extension de l'ANSI-C, C++ utilise une gestion mémoire explicite avec les *delete, new, free* et *malloc*. Notamment certains chercheurs dans [Bartlett, 1989][Boehm et Weiser, 1988][Coplien, 1992][Edelson et Pohl, 1990] [Edelson et Pohl, 1991] [Ellis et Detlefs, 1993] et [Detlefs et al, 1994] ont intégré un *Garbage Collector* à C++. Nous détaillons dans ce qui suit la gestion explicite et implicite via le *Garbage Collector* au sein de C++.

2.3.1. Gestion explicite en C++

Lors de la déclaration d'un objet, il est possible de définir les méthodes *constructeur* et *destructeur*. Le *constructeur* est appelé lors de l'instanciation de l'objet et ce, en lui allouant un espace mémoire et en initialisant les membres de cet objet. Il peut cependant avoir d'autres fonctionnalités : acquérir des ressources comme des routines du système ou ouvrir des fichiers ou des bases de données. Aussi, destructeur est appelé à la fin du programme ou lorsque la fonction *delete* est appelée. Le destructeur est utilisé pour l'action de *finalisation* lorsqu'elle est nécessaire, mais aussi pour désallouer l'espace mémoire consommé par l'objet. Typiquement, l'action de finalisation consiste à libérer les routines ou à fermer un fichier.

Comme son prédécesseur, pour la gestion du cycle de vie de ses variables, le langage C++ a utilisé les fonctions *new (malloc())* et *delete (free())*. Ces deux fonctions ont aussi intégré respectivement les constructeurs et les destructeurs des objets pour pouvoir leur allouer et libérer l'espace mémoire. Ainsi la création d'un objet passe automatiquement par l'allocation d'un espace mémoire qui lui est requis. Le cycle de vie d'un objet se termine par la destruction de cet objet et ce, en libérant l'espace mémoire qu'il occupait via la fonction *delete*.

2.3.2. *Garbage Collector* pour C++

Pour Ellis et Stroustrup [1990], la gestion de mémoire explicite via le *Garbage Collector* n'est pas une composante du langage C++. Dans [Jones et Lins, 1996], les auteurs évoquent que plusieurs programmeurs C++ se sont opposés fermement à l'intégration d'un *Garbage Collector* comme standard au sein de C++. D'autres programmeurs, au contraire, trouvent que l'intégration du *Garbage Collector* pourrait augmenter l'efficacité de C++ comparativement à d'autres outils tels que le système de manipulation des images au sein d'Adobe Photoshop qui est géré par un *Garbage Collector* à base de compteurs de références [Stroustrup, 2004].

L'opposition à l'intégration d'un *Garbage Collector* au sein de C++ s'est reposée sur deux principaux aspects : l'efficacité et la complexité [Jones et Lins, 1996]. Toujours dans [Jones et Lins, 1996], on trouve que plusieurs programmeurs C++, spécialement ceux ayant un background C, étaient très suspicieux de léguer la gestion mémoire au collecteur. L'utilisation du *Garbage Collector* était perçue inutile puisqu'il existait déjà des outils de débogage dédiés à la gestion mémoire, mais surtout qu'un *Garbage Collector* cause des pénalités temporaires trop élevées. De même, il était évident que l'intégration d'un *Garbage Collector* sans compromettre le code existant et sans imposer des contraintes de programmation serait très complexe.

Le *Garbage Collector* représente une composante essentielle dans plusieurs langages de programmation orientée objet. Smaltalk [Goldberg et Robson, 1983], Eiffel [Meyer, 1988] et des variantes de Lisp ont incorporé un *Garbage Collector* depuis le début de leur création. Meyer [1988] a placé la gestion automatique de la mémoire en troisième place dans une liste des « sept recommandations pour la programmation orientée objet ». Plusieurs études, notamment celle de Rovner [1985], ont démontré qu'une considérable proportion du temps de développement était passée à résoudre les problèmes liés à la gestion mémoire. De nos jours, il existe plusieurs outils d'aide à la gestion de la mémoire comme Purify [Pure Software, 1992] et CenterLine [CenterLine Software, 1992]. L'existence de ces outils prouve l'importance accordée à la gestion mémoire et la nécessité de son optimisation [Jones et Lins, 1996]. Cependant, de tels outils ne sont pratiques que pour le débogage puisqu'ils nécessitent un temps d'exécution

exponentiel. En effet, Ellis [1993] a démontré que CenterLine consomme cinq fois plus de temps que le programme même, et que Purify en consomme de deux à quatre fois plus.

Dans leur travail, Ellis et Detlefs [1993] ont identifié cinq contraintes à l'intégration d'un *Garbage Collector* au sein de C++ :

- Ni les programmeurs ni les revendeurs des compilateurs n'accepteront de multiples ou majeurs changements au sein du langage. En effet les programmeurs n'accepteront pas les changements qui peuvent affecter leur méthodologie ou leur style de programmation, les vendeurs, de leur côté, ne voudront pas changer de compilateurs.
- Tout code collecté par un *Garbage Collector* devra coexister avec les composants qui n'en utilisent pas. En effet, il existe au sein du langage C++ plusieurs librairies utilisées qui sont écrites en C. Il serait irréaliste de reprendre toutes ces librairies et de les réécrire de manière à ce qu'elles deviennent compatibles avec des composants utilisant un *Garbage Collector* puisque dans la plupart des cas les programmeurs n'ont jamais accès à ces librairies. Un corollaire à cette contrainte est la nécessité d'une coexistence à la fois d'une gestion explicite et implicite de la mémoire. Selon Ellis et Detlefs, avec l'utilisation d'un *Garbage Collector*, il faudra aussi apporter des modifications dans la sémantique des destructeurs si la finalisation était asynchrone pour l'utilisateur.
- Il faudra définir les règles de sûreté du *Garbage Collector*. La suppression d'un objet prématurément, par exemple, pourrait entraîner des difficultés à retracer les erreurs. L'approche d'un *Garbage Collector* est de parcourir le graphe d'objet et de déterminer l'accessibilité des objets. Cependant, avec la gestion des pointeurs, comme nous l'avons vu avec les collecteurs conservatifs dans la section précédente, la tâche d'un *Garbage Collector* devient plus imposante en terme de temps pour coder les informations.
- Les programmes qui utilisent un *Garbage Collector* doivent être portables, c'est-à-dire que le résultat d'un programme utilisant un *Garbage Collector* devra être le même que celui implémenté en C++.

- La *Garbage Collection* en C++ ne sera acceptée et approuvée que si elle démontre une efficacité. La crainte principale des programmeurs vis-à-vis l'intégration d'un *Garbage Collector* au sein de C++ concerne les ralentissements considérables du temps d'exécution. Cependant, Ellis et Detlefs assurent qu'un petit sacrifice du temps d'exécution garantit la réduction du temps nécessaire au développement et au débogage associé aux problèmes de mémoire.

D'un autre côté, selon Detlefs, dans son travail [Detlefs, 1992], pour qu'un *Garbage Collector* puisse être légitime et reconnu au sein de C++, il devra assurer les cinq priorités suivantes :

- Le *Garbage Collector* ne devra exiger aucun appui ou aucune information de la part du compilateur.
- Le *Garbage Collector* ne doit exiger aucune information de la part du programmeur concernant le format des objets à collecter.
- Le *Garbage Collector* devra se charger des problèmes qui peuvent survenir à la suite à d'optimisations du compilateur.
- Le *Garbage Collector* permet l'utilisation à la fois d'une gestion implicite et explicite de la mémoire dans un même programme.
- Le *Garbage Collector* devra invoquer les destructeurs pour les objets collectés en mémoire.

Plusieurs stratégies de *Garbage Collection* ont été proposées pour être intégrées dans le langage C++. Boehm et Weiser [1988] ont proposé un collecteur conservatif évoluant avec une technique *Mark and Sweep* qui a été élaborée pour C. Boehm [1993] a apporté quelques modifications pour l'intégrer avec C++. Ce collecteur a répondu à toutes les contraintes établies par Ellis et Detlefs. Il ne demande ni changement du langage ni restrictions du code. Seulement, ce collecteur est vulnérable à d'éventuelles optimisations de code. D'un autre côté, ce collecteur n'invoque pas les destructeurs pour les objets en mémoires [Detlefs, 1992]. Le collecteur de Bartlett [Bartlett, 1989], lui aussi, ne demande ni changement du langage ni restriction du code. Cependant, le programmeur devra implémenter une méthode pour différencier les cellules mémoires contenant des pointeurs des autres cellules. Detlefs [1991] a apporté certaines

modifications au collecteur de Bartlett en étendant ce dernier et ce, en ajoutant un pré-compilateur capable d'ajouter de l'information dans l'entête de chaque objet en mémoire. Les collecteurs de Bartlett et de Detlefs ont plusieurs propriétés communes avec celui de Boehm et Weiser. Il n'y a ni changement de syntaxe du langage ni contraintes pour le code. Seulement, ces trois collecteurs sont très vulnérables à l'optimisation du compilateur. Le mouvement des objets d'un emplacement à un autre peut compromettre la coexistence avec les librairies existantes dans le cas du collecteur du Detlefs [Jones et Lins, 1996].

Ferreira [1991] a proposé un collecteur qui assure à la fois une gestion explicite de la mémoire et un appel aux destructeurs. Le collecteur est un algorithme conservatif *Mark and Sweep* comme celui de Boehm-Weiser [Boehm et Weiser 1988]. Seulement, le programmeur devra invoquer une macro à chaque définition d'un nouvel objet, ce qui implique donc une participation du programmeur. On viole ainsi la deuxième propriété de Detlefs [Detlefs, 1992]. Ferreira a, quant à lui, violé la première règle de Detlefs en ajoutant des modifications au compilateur de C++ pour la mise en œuvre de règles qui améliorent la performance des programmes. Dans [Seliger, 1990], l'auteur décrit une extension de C++ qui définit un nouveau langage implémenté par un translateur qui retourne du C++ comme résultat. Cela implique un changement sur le plan du compilateur d'où une violation de la première propriété de Detlefs. Kuse et Kamimura [1991] ont utilisé la même approche et ont créé leur propre langage C++ qui inclut un *Garbage Collector*. Plus récemment encore, dans son article, Boehm [2002] décrit un collecteur conservatif qui assure une collection en minimisant le temps d'exécution, mais le compilateur devra alors subir quelques modifications.

Edelson et Pohl dans [1991] ont présenté un collecteur de recopie qui se base sur les *smart pointers* [Coplien, 1992]. Ces derniers peuvent être implémentés comme classes *template* ou comme *pré-processeur*. Généralement, ils surchargent les opérateurs -> et * pour imiter le comportement des pointeurs sur des objets. Les constructeurs des *smart pointers* dans le collecteur d'Edelson et Pohl insèrent à partir des racines les adresses d'un pointeur dans une structure de données appropriée et le destructeur le détruira ensuite. Ainsi, le collecteur n'aura qu'à parcourir cette structure pour retrouver tous les pointeurs et accomplir sa tâche de gestion de mémoire. Parce qu'il ne peut récupérer

l'adresse de *this*, Edelson est amené à rejeter le collecteur par recopie se basant sur les *smart pointers* au détriment d'un collecteur *Mark and Sweep* basé aussi sur les *smart pointers* [Edelson, 1992]. Pour utiliser ce dernier collecteur, il faut que le programmeur implémente une fonction *mark (Object O)* pour chaque classe utilisée. Les *smart pointers* ont aussi été utilisés par Detlefs [1992] pour implémenter une interface qui représente un *Garbage Collector* appliqué à un collecteur par compteur de références. Dans [Edelson, 1992], l'auteur décrit en détail, d'une part, les avantages et les inconvénients des *smart pointers* et, d'autre part, l'impact de leur utilisation pour implémenter un *Garbage Collector*. Les *smart pointers* possèdent des constructeurs qui leur permettent d'être initialisés à chaque appel de la fonction *new ()*. Ils ont principalement deux inconvénients majeurs : ils ne supportent pas les pointeurs vers des objets constants et ils ne supportent pas les conversions standards des pointeurs telle que la conversion d'un tableau vers un pointeur sur le premier élément de ce tableau (*int T[10]* vers *int *T* ou encore *0* vers *NULL*) [Detlefs 1992]. Ces inconvénients peuvent bien sûr être surmontés par l'apport de quelques modifications au compilateur.

2.3.3. Conclusion

Le langage C++ a été créé sans *Garbage Collector*. Plusieurs tentatives d'incorporations d'outils pour la gestion implicite des cycles de vie des objets ont connu l'échec et ce, pour de nombreuses raisons que nous avons détaillées dans la section précédente (section 2.3.2). Les recherches ont montré que l'algorithme conservatif de Boehm et Weiser [1988] est le plus approprié à être la clé de la réussite pour l'incorporation d'une telle composante [Boehm, 2002]. Si Jones et Lins [1996] croient peu probable l'incorporation d'un *Garbage Collector* au sein de C++, Ellis et Detlefs [1993] ou encore Boehm [2002] croient que l'incorporation pourrait avoir lieu un jour et répondre à toutes les contraintes citées plus haut.

2.4 Conclusion

Nous avons survolé, dans ce chapitre, la gestion mémoire implicite et explicite. Nous avons vu qu'il existe plusieurs méthodes pour gérer les ressources mémoires. Les méthodes de gestion implicite de la mémoire figurent comme les plus prometteuses pour

un bon nombre de langages de programmation et les techniques varient du plus simple, comme la technique des compteurs de références, au plus compliqué comme les collecteurs générationnels ou encore le *leasing*. Nous avons vu aussi que le *Garbage Collector* a évolué sous plusieurs formes dans les langages à gestion implicite de la mémoire, mais qu'il a aussi été ajouté à des langages à gestion explicite de la mémoire comme le C++. Ce dernier est un langage reconnu pour sa puissance et sa robustesse et l'ajout d'un *Garbage Collector* ne devrait en aucun cas l'en diminuer. Plusieurs tentatives ont été faites pour l'ajout avec succès d'un *Garbage Collector*, mais aucune d'entre elles n'a eu une approbation de la part de la communauté de C++.

Dans le chapitre suivant, nous détaillerons la programmation par aspects qui nous permettra de développer l'outil de gestion de mémoire implicite pour le langage C++.

Chapitre 3 : Préoccupations et aspects

3.1 Introduction

La séparation des préoccupations (*separation of concerns*, ou SOC) consiste à identifier plusieurs domaines d'intérêts pour un même ensemble d'objets et ce, relativement aux différentes apparences qu'ils peuvent avoir. Il s'agit de séparer les exigences pour pouvoir les traiter de façon plus ou moins indépendante. La séparation peut se faire à différentes phases du développement d'une application. L'objectif est de garder une concordance entre les différentes étapes et de retarder le plus possible la fusion des préoccupations pour un même ensemble d'objets.

Le concept de modularisation peut être perçu comme un cas particulier de l'idée de «séparation de préoccupations». Toutefois, toutes les exigences ne se traduisent pas forcément par des livrables distincts. Il existe cependant certaines exigences que les techniques de développement traditionnelles (procédurales, objet) ne permettent pas de modulariser, mais que les nouveaux paradigmes, communément regroupés sous le nom de développement de logiciels par aspects ou *Aspect Oriented Software Development (AOSD)*, permettent de faire. Dans le contexte des méthodes de développement dites orientées aspects, une préoccupation est définie comme un ensemble d'exigences reliées d'un logiciel, et un aspect est, lui, défini comme la réalisation de ces exigences.

Par ces séparations, le logiciel n'est plus abordé dans sa globalité, mais dans ses parties. Cette approche réduit non seulement la complexité de conception et de réalisation, mais aussi la maintenance d'un logiciel et en améliore la compréhension, la réutilisation et l'évolution. Cette séparation est la motivation d'approches telle que la programmation par aspects (*Aspect Oriented Programming* en anglais AOP) [Kicsales *et al.*, 1997] qui permet de modulariser des exigences non fonctionnelles telles que la sécurité, la persistance, l'archivage, le traçage, etc. La programmation par sujets (*Subject Oriented Programming* en anglais SOP) [Harrison et Ossher, 1993] ou la programmation par vues (*View Oriented Programming* en anglais VOP) [Mili *et al,* 1999], quant à elles, permettent de modulariser des exigences fonctionnelles touchant aux mêmes entités.

La modularisation par préoccupations est un outil conceptuel qui permet de gérer la complexité d'un système. Les chercheurs ont exploré de nouveaux moyens pour la modularisation qui peuvent mieux capturer et encapsuler différentes préoccupations, tant fonctionnelles que non fonctionnelles. Par exemple, il a été démontré que le contrôle d'accès et la synchronisation entre les objets ne peuvent être exprimés dans un langage orienté objet courant sous forme de modules séparés [Bergmans et Atkis, 2001]. Par contre, ces exigences peuvent être modularisées grâce à la programmation par aspects.

Les exemples suivants peuvent illustrer quelques préoccupations :

- préoccupations concernant les données comme la persistance des données ou le contrôle des accès,
- préoccupations concernant les fonctions métiers comme les fonctions comptables ou les fonctions de gestion du personnel d'une entreprise,
- préoccupations concernant les règles de gestion comme des règles organisationnelles dans une application de gestion du personnel,
- préoccupations architecturales comme l'interopérabilité et la communication entre applications hétérogènes,
- préoccupations systèmes comme l'intégration d'un logiciel dans sa plateforme d'exécution, la sécurité, la persistance, etc.

La séparation des préoccupations fournit ainsi un support méthodologique de modélisation et de programmation. Elle doit bien sûr être accompagnée d'un processus d'intégration des différentes composantes générées pour les différentes préoccupations. Ce processus est appelé processus de *tissage* (*weaving* en anglais). On pourra gérer, par la suite, plus naturellement et modulairement des intégrations très complexes.

Les facettes fonctionnelles (*separation of concerns*) peuvent être utilisées comme une solution permettant i) d'identifier, ii) d'encapsuler et iii) de manipuler les parties d'un logiciel, qui sont pertinentes pour un concept particulier. La séparation des facettes ou des préoccupations consiste à séparer les différents aspects (ou *concerns*) d'un problème ainsi que leur implémentation [Tarr *et al.*, 1999]. Cette approche propose de se concentrer sur le développement des entités de base et de concevoir séparément, par la suite, les fonctionnalités connexes. Ces différentes préoccupations sont ensuite composées grâce à des points de jointure selon l'approche utilisée : AOP, SOP ou VOP.

3.2 Programmation par aspects (AOP)

3.2.1. Introduction

L'une des techniques de séparation des préoccupations les plus étudiées à l'heure actuelle correspond à la programmation par aspects (AOP, Aspect-Oriented Programming). Celle-ci a été introduite par des chercheurs du XEROX PARC, en 1997 [Kicsales *et al.*, 1997]. Il s'agit d'une technique novatrice pour l'ingénierie des applications complexes, telles que les applications distribuées, les Entreprise Ressource Planning (ERP) et bien d'autres.

Typiquement, plusieurs programmes ont besoin de partager des ressources (imprimante, mémoire, processeur...), de traiter des erreurs, d'optimiser les performances, etc. Ces composantes partagées peuvent être classifiées en deux parties : composantes fonctionnelles et aspects. La programmation par aspect se fond sur une séparation claire entre les préoccupations « métiers » (ou « fonctionnelles ») et « non fonctionnelles » présentes dans les applications. Ce point de vue est similaire à celui présent dans les serveurs d'applications (.Net, EJB, CCM) où les composants fournissent des services métiers et où les serveurs d'applications sont une structure d'accueil proposant aux composants des services système.

Néanmoins, l'AOP ne s'arrête pas à ce premier niveau de découpage et vise à appliquer la séparation à toutes les préoccupations, qu'elles soient fonctionnelles ou non. Chaque aspect est destiné à être développé de façon indépendante puis intégré à une application par un processus dit de tissage d'aspects (*aspect weaving*).

La composition entre les préoccupations fonctionnelles et non fonctionnelles, dans un même programme, crée des systèmes difficiles à développer, à comprendre et à maintenir [Kizales *et al.* 1997]. Ipso facto, on devra faire la distinction entre une composante fonctionnelle et un aspect. Une composante fonctionnelle est une propriété qui peut être encapsulée dans une procédure généralisée tels un objet, une méthode, une procédure ou une fonction. Un aspect, lui, est un module indépendant spécifiant des propriétés et des contraintes sur son application à une classe donnée; l'ajout d'un aspect à

une classe lui accorde ses propriétés. Cet ajout se fait automatiquement grâce à un outil pour la programmation par aspect comme AspectC++ [Spinczyk O. *et al.*, 2002], AspectJ [AspectJ, 99], ou encore AspectWerkz [AspectWerkz, 2006]. Les aspects les plus étudiés jusqu'à ce jour sont le traitement d'exceptions, le traitement des erreurs, la synchronisation des objets concurrents, l'archivage, le tracing, etc. [Spinczyk O. *et al.*, 2002][Kizales *et al.* 1999][AspectWerkz, 2005][JAC, 2005].

3.2.2. Motivation et fonctionnement

Afin de comprendre la différence entre une composante fonctionnelle et un aspect, nous considérons une classe «Etudiant», une classe « Cours » et une classe «Commande». Les classes «Etudiant» et « Cours » appartiennent au même domaine, tandis que la classe « Commande » appartient à un domaine différent. Ces classes contiennent, en même temps, des aspects et des composantes fonctionnelles tel qu'illustré à la Figure 3-1. Dans ces classes, il peut y avoir une confusion entre le code décrivant l'objet (Etudiant, Commande ou Cours) et le code concernant le traitement des exceptions. Le code décrivant l'objet traite les comportements de celui-ci. Par contre, le code concernant les exceptions traite les aspects liés au système et à plusieurs applications et non pas à un objet particulier. Ainsi, il est difficile de comprendre ce que ces classes sont supposées faire. L'exception « Etudiant introuvable » (en pointillés) concerne les deux classes « Etudiant » et « Cours » qui appartiennent au même domaine, tandis que l'exception «Impression interrompue» (en grisé) concerne les deux classes « Etudiant » et « Commande » qui appartiennent à deux domaines complètement différents. La programmation par aspects permet de faire une séparation entre tout ce qui concerne les composantes fonctionnelles et non fonctionnelles des classes.

Avec cet exemple, nous pouvons aussi comprendre qu'il est inutile de réécrire à chaque fois le code concernant ces exceptions puisqu'il est identique. Outre la séparation entre les composantes fonctionnelles et non fonctionnelles, la programmation par aspect permet aussi de modulariser le code de ces aspects non fonctionnels et de le propager sur des domaines différents. Dans l'exemple, les domaines des deux classes « Etudiants » et « Commande » sont distincts.

La programmation par aspect (AOP) fournit des mécanismes pour abstraire et composer les aspects. Le tisseur (*weaver*), qui est le compilateur d'aspects, permet d'appliquer les aspects aux composantes. En sortie, les composantes associées aux aspects, par le tisseur, génèreront un programme complet tel que montré à la Figure 3-2.

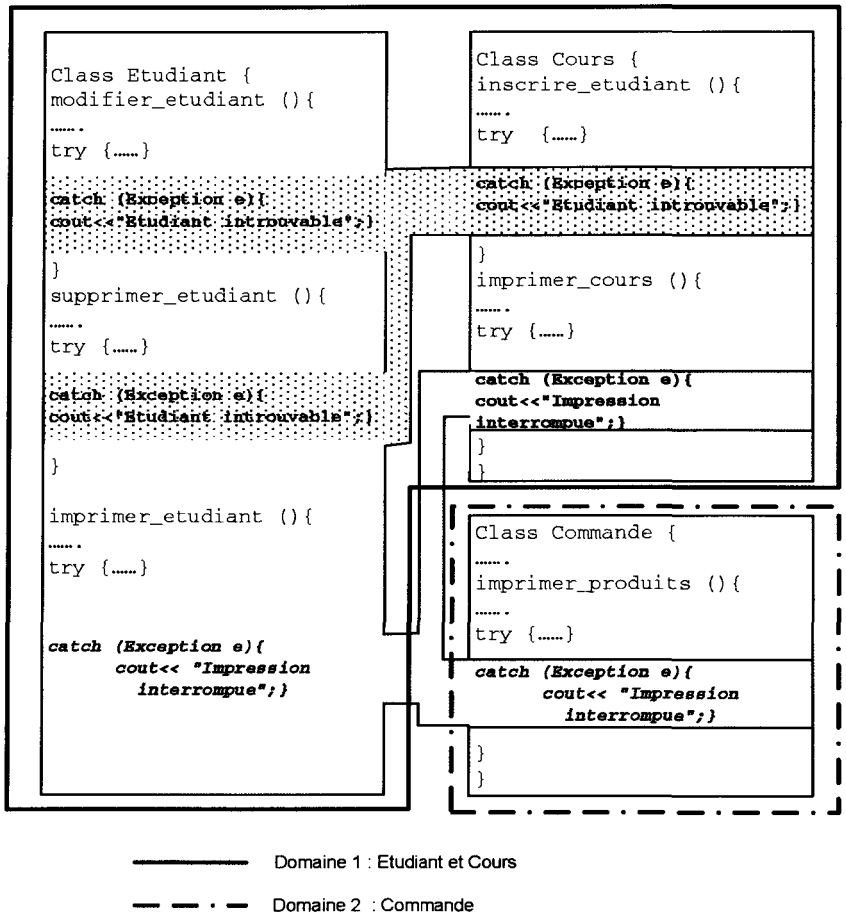

Figure 3-1 : Composantes fonctionnelles

Les composantes et les programmes de sortie sont écrits dans le langage de programmation C++ dans le cas d'AspectC++. Le langage d'aspect est un langage procédural simple fournissant des opérations sur des méthodes et des classes développées dans les composantes fonctionnelles. Techniquement, un aspect est une entité regroupant un ensemble de *conseils* (*advice*), de *points de jonction* (*joinpoint*), ou encore des *coupes transverses* (*pointcut*). Un aspect permet de fournir une implémentation modulaire d'une *préoccupation transverse* en définissant des *méthodes d'aspects*. Une *préoccupation*

transverse est une préoccupation liée à une implémentation qui affecte de nombreuses parties différentes d'un même programme. Un *conseil* ou encore *greffon* est un fragment de code à insérer dans les *points de jonction.* Le conseil implante une *préoccupation transverse* qui regroupe un ou plusieurs domaines. Un *point de jonction* est un point donné dans l'exécution de programme (par exemple, un appel à une méthode ou la fermeture d'un port). Il est soit un évènement présent dans le flux de contrôle (on parle alors de *points de jonction dynamiques*), soit un élément présent dans une structure statique du programme (on parle alors de *points de jonction statiques*). Les méthodes d'aspects affectent le programme sur ces *points de jonction.* Il est à noter qu'il n'est pas possible d'insérer un conseil au milieu du code d'une fonction. Par contre, il est possible de le faire avant, autour de, à la place de ou après l'appel de la fonction. Une *coupe transverse* est un ensemble de *points de jonction* et elle est exprimée en fonction des expressions régulières [Broberg *et al.*, 2004] qu'on appelle *expressions de coupes transverses.* On compose les *expressions de coupes transverses* en appariant les expressions et les fonctions de *coupes transverses.* Elles définissent l'emplacement où la méthode d'aspect est censée affecter le programme. Une *méthode d'aspect* définit la façon dont un aspect affecte le programme sur une *coupe transverse* donnée. Ces concepts sont implantés dans les langages de programmation tel que C++ (AspectC++ [Spinczyk O. *et al.*, 2002] et TACO [Vaysse, 2005]), Java (AspectJ [AspectJ, 2005] et [Lesiecki, 2002]), AspectWerkz [AspectWerkz, 2005] et JAC [JAC, 2005].

Ces tisseurs sont classés en deux catégories selon la stratégie de tissage :

- un tissage statique par instrumentation du code source ou du pseudo-code machine intermédiaire (bytecode java, IL) (dans le cas d'AspectJ [AspectJ, 2005] et AspectWerkz [AspectWerkz, 2005]);
- un tissage dynamique lors de l'exécution du logiciel (comme dans le cas de AspectC++ [Spinczyk O. *et al.*, 2002], TACO [Vaysse, 2005] et JAC [JAC, 2005]).

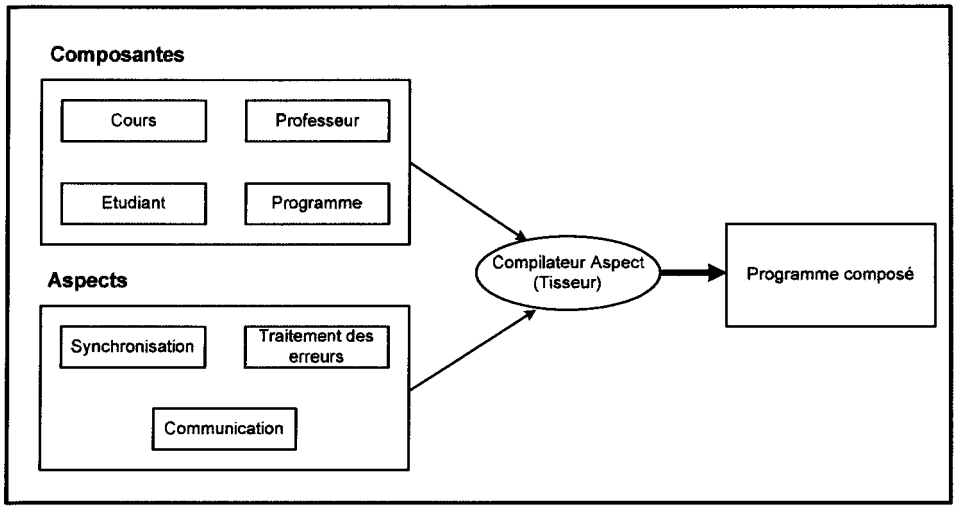

Figure 3-2 : Composition des composantes et aspects

Tel qu'illustré à la Figure 3-2, le compilateur d'aspects (*weaver*) procède en trois phases :

- Il utilise le déploiement (*unfolding*) comme technique pour générer le diagramme de flux de données.
- Le programme d'aspects est exécuté pour éditer le graphe en identifiant et fusionnant les nœuds ensemble et en ajustant leurs corps en conséquence.
- Un générateur du code parcourt le graphe fusionné pour générer le programme en un langage cible tel que C++ ou Java.

Ce compilateur applique les aspects (Synchronisation, Communication et Traitement des erreurs) sur les composantes (Cours, Professeur, Etudiant et Programme). Les composantes sont développées indépendamment des aspects. L'intégration est réalisée grâce aux règles décrites dans les aspects sous forme de conseils, de points de jonction et de coupes.

Dans ce travail de recherche, nous utilisons essentiellement le tisseur AspectC++. La majeure partie de nos exemples sont écrits avec ce langage que nous verrons de façon plus détaillée dans la section suivante.

3.2.3. Programmation par aspects avec AspectC++

Considérons la classe « Cours » contenant les fonctionnalités de cet objet et l'aspect «*Affichage_nomf*» qui va afficher le nom d'une des méthodes de la classe « Cours » à chaque fois qu'il y a un accès à l'une d'entre elles. D'un point de vue syntaxique, un aspect sous AspectC++ est très proche d'une classe sous C++. Toutefois, en plus des fonctions membres et des éléments de données, un aspect peut définir une méthode d'aspects. Après le mot clé *advice*, une expression de coupe transverse définit l'emplacement où la méthode d'aspect est censée affecter le programme (autrement dit, les points de jonction), alors que la partie qui suit le mot clé *advice* définit la façon dont le programme est censé être affecté sur ces points. Il s'agit de la règle générale pour l'ensemble des différents types de méthodes d'aspect sous AspectC++ [Spinczyk O. *et al.*, 2002].

L'expression de la coupe transverse donnée dans l'exemple de la Figure 3-3 est `execution("% ...::%(...)")`. Autrement dit, cette méthode d'aspect devrait affecter l'exécution de l'ensemble des fonctions qui ne satisfont pas l'expression `"% ...::%(...)"`. Dans les « expressions de correspondances », le signe « % » et les caractères « ... » sont utilisés en tant que caractères de remplacement. Le signe pourcentage « % » correspond à n'importe quel type. Par exemple, `"% *"` correspond à tous types pointeurs ainsi qu'à n'importe quelle séquence de caractères dans les identifiants. De même, `"trilia_%"` correspond à l'ensemble des classes dotées d'un nom commençant par `trilia_`. Les points de suspension « ... » équivalent à n'importe quelle séquence de types ou à des espaces de nommage. Par exemple, `"int toto(...)"` équivaut à n'importe quelle fonction générale chargée de retourner un entier et nommée toto. Enfin, l'expression de correspondance `"% ...::%(...)"` équivaut à n'importe quelle classe ou espace de nommage [Spinczyk O. *et al.*, 2002].

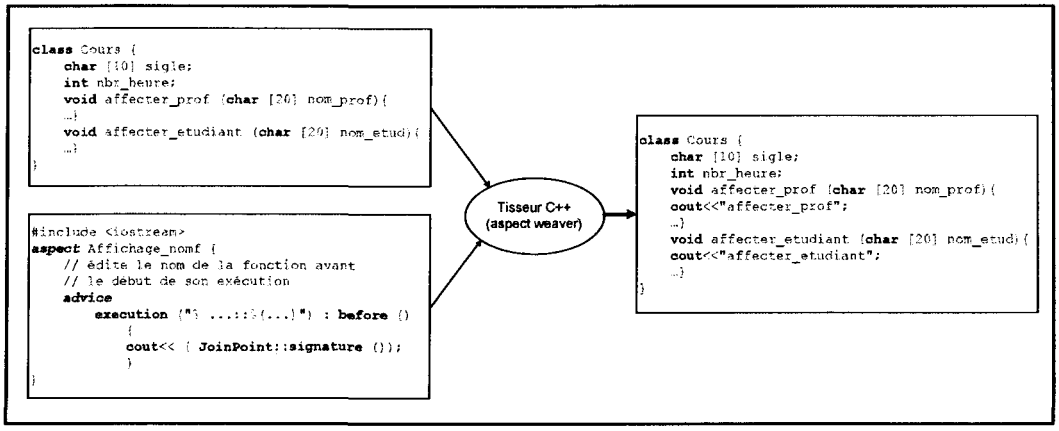

Figure 3-3 : Fonctionnement du tisseur AspectC++

Les expressions de correspondance représentent des ensembles d'entités de programme nommés comme les fonctions ou les classes. Ainsi, les expressions de correspondance sont toujours des expressions de coupes transverses primitives chargées de décrire un ensemble de points de jonction dans la structure statique du programme et sont nommées « points de jonction statiques ». Toutefois, dans l'exemple de la Figure 3-3, nous utilisons une méthode d'aspect pour décrire un évènement dans le flux dynamique de contrôle du programme, à savoir l'exécution des fonctions. La fonction de coupe transverse *execution()* est, par conséquent, utilisée. Celle-ci a pour objectif de produire l'ensemble des points de jonction d'exécution des fonctions données en argument. L'exemple de la Figure 3-3 représente l'aspect de *tracing*. Il a pour but de modulariser l'implémentation des opérations de données de sorties utilisées afin de pister le flux de contrôle du programme. L'aspect de *tracing* est aspect dit de développement classique. Par opposition aux aspects dits de *production*, ces aspects ne sont utilisés qu'au cours du développement d'une application et ont pour but de déboguer et d'obtenir une assurance qualité de l'application. Les aspects dits de *production* font partie du produit logiciel final livrable.

En ce qui concerne les points de jonction dits dynamiques, AspectC++ supporte trois types de méthodes d'aspect dans le code, intitulées *before(), after(),* et *around().* Ces méthodes implémentent toutes une partie supplémentaire au comportement du programme. Dans l'aspect «Affichage_nomf » de l'exemple de la Figure 3-3, ce

comportement est implémenté au moyen de la déclaration `cout>>` à la suite de *before()*. D'un point de vue syntaxique, cette technique est identique au corps d'une fonction et, de ce fait, il est d'ores et déjà possible de considérer *advice body* (ou corps de la méthode d'aspect) comme une fonction membre anonyme de l'aspect.

À la place de *before()*, il était aussi bien possible d'utiliser la méthode d'aspect *after()* (ou les deux) dans l'exemple. Auquel cas, le corps de la méthode d'aspect s'exécuterait une fois l'exécution d'une fonction achevée. Un corps de méthode d'aspect *around()* est exécuté à la place du flux de contrôle et est censé suivre normalement le *point de jonction dynamique* [Spinczyk O. *et al.*, 02]. On définit alors *les méthodes d'aspects* qui caractérisent la façon dont un aspect affecte le programme sur une coupe transverse donnée. Concernant les points de jonction dynamiques intitulés *before(), after(),* ou *around(),* ces méthodes d'aspect peuvent être utilisées afin d'implémenter des comportements supplémentaires.

Dans Spinczyk O. *et al.* [2002], on note que AspectC++ peut supporter les *points de jonction statiques et dynamiques* où les *points de jonction statiques* sont des entités statiques que l'on retrouve habituellement dans les programmes et les *points de jonction dynamiques* sont des évènements qui se produisent durant l'exécution d'un programme. Les entités de C++ telles que les classes, les structures, les unions et les fonctions (membre et non, operateur, etc.) sont considérées comme *points de jonction statiques.* Ces *joinpoint* sont décrits par les *expressions de correspondance.* L'expression `"% ...::Trilia(...)"` fait correspondre toutes les fonctions intitulées Trilia avec n'importe quel type de retour ou n'importe quel argument. Les évènements de C++ comme les appels de fonctions, l'exécution des fonctions, les constructeurs d'objets et les destructeurs d'objets sont considérés comme des *points de jonction dynamiques.* La description d'un *point de jonction dynamique* est basée sur le mécanisme de description des *points de jonction statiques* en adjonction avec les *expressions de coupes transverses* telles que :

- `call("% ...::trilia(...)")`
- `execution("float MathFuncs::%(float)")`
- `construction("trilia")`
- `destruction("X"||"Y")`

Les aspects sont automatiquement intégrés à la compilation. Mais le code de la classe «Cours» n'a pas besoin de connaître le contenu et la finalité des aspects intégrés. Les avantages d'une telle approche sont la simplicité ainsi que la facilité de comprendre et de maintenir le code. La simplicité est due au fait que les aspects sont intégrés facilement sans préoccupation des conflits des caractéristiques et sans besoin d'informations sur l'environnement où évolue l'application. La facilité de compréhension et de maintenance est due au fait que AOP sépare les composantes fonctionnelles des aspects. Cette approche pourrait supporter le changement dynamique des comportements d'objets. Il serait ainsi possible d'ajouter ou de retirer des aspects durant la phase d'exécution, et ce, par l'ajout d'autres aspects. On utilise alors les considérations entrecroisées ou appelées encore préoccupations transverses (*cross-cutting concers*) qui sont des mélanges, au sein du même programme, de sous-programmes couvrant des aspects techniques séparés [Spinczyk O. *et al.*, 02]. Ainsi, dans l'exemple de la Figure 14, il serait possible d'ajouter un aspect qui dirigera le flux vers une imprimante et non vers l'écran.

Il est aussi possible de combiner les coupes transverses au moyen des opérateurs « *&&* » (intersection), « || » (union), et « ! » (inversion). Par exemple, l'expression `"%toto(int,...)" || "int titi(...)"` équivaut à n'importe quelle fonction générale intitulée `toto` prenant un `int` (entier) en premier paramètre et n'importe quelle fonction générale appelée `titi`, chargée de retourner un `int`. En combinant des fonctions de coupes transverses entre elles, on obtient des expressions plutôt puissantes afin de décrire l'emplacement où la méthode d'aspect est censée affecter le programme. Par exemple, nous pourrions changer l'expression de la coupe transverse par l'aspect « Affichage » comme suit :

```
advice call ("% ...::%(...)")
&& within ("Cours") : before () {
cout>>"calling ">> (JoinPoint::signature ());
}
```

La fonction `call()` produit l'ensemble des points de jonction des appels pour des fonctions données. Par opposition aux points de jonction d'exécution, les points de jonction d'appels prennent effet du côté de la routine appelante, c'est-à-dire avant, après

ou en même temps que l'appel de la fonction concernée. La fonction de la coupe transverse `within()` se contente de retourner l'ensemble des points de jonction dans les classes ou fonctions indiquées. En introduisant la méthode d'appel à l'intersection de `call("% ...::%(...)")` (soit n'importe quel appel de fonction) et de `within("Cours")` (n'importe quel point de jonction dans la classe Cours), l'aspect ne pistera dès lors que les appels de fonction lancés à partir d'une méthode de la classe « Cours ».

3.2.4. Conclusion

La séparation des préoccupations est un paradigme consistant à déterminer différents domaines d'application pour un ensemble d'objets. La séparation a lieu en différenciant les aspects fonctionnels et non fonctionnels. Les aspects non fonctionnels comme le *tracing*, ne pouvant être exprimés par le biais des techniques de développement traditionnelles, sont décrits par de nouvelles méthodes regroupées sous ce qu'on appelle *développement de logiciels par aspect* ou encore *Aspect Oriented Software Development (AOSD)*. Parmi ces méthodes, nous pouvons citer la programmation orientée sujet (SOP), la programmation orientée vue (VOP) ou encore la programmation par aspects (AOP). Nous nous intéressons, dans ce travail de recherche, à la programmation par aspects qui est un paradigme qui ne consiste pas à décrire le fonctionnement du programme, ou bien encore sa structure comme en programmation orientée objet. Il consiste plutôt à décrire des modifications à lui apporter. Pour apporter ces modifications aux programmes, on utilise des *tisseurs d'aspects* (*aspect weaver*) qui prennent en entrée le code du programme et les fonctionnalités non fonctionnelles codées en un langage de programmation aspect. La sortie est un seul code composé du code principal et des aspects. Nous utilisons dans ce travail de recherche l'outil AspectC++ [AspectC++, 2006] qui est un tisseur d'aspect adapté au langage C++. Actuellement, il existe un autre tisseur adapté à C++, Taco [Vaysse, 2005], mais contrairement à AspectC++ qui permet de modifier les méthodes ou les membres d'une classe donnée, Taco permet de modifier les structures de contrôle du langage cible [Vaysse, 2005]. En plus de cela, AspectC++ ayant subit plus de tests, il serait plus stable que le tisseur Taco [Vaysse, 2005].

Après avoir décrit la gestion des cycles de vie en général et celle en C++ dans le chapitre précédent, et après avoir décrit la séparation des préoccupations en général et la

programmation par aspects en particulier, nous établissons, dans la prochaine section, l'objectif de la recherche ainsi que les différentes étapes de la méthodologie proposée.

3.3 L'objectif de la recherche

Comme nous l'avons vu au sein du deuxième chapitre, la gestion de la mémoire est un élément majeur à considérer lors du développement d'un logiciel. Certains langages offrent une gestion implicite qui facilite la tâche des programmeurs lors de la phase de développement. D'autres langages, comme le C++, n'offrent pas une gestion mémoire implicite qui pourrait faciliter le travail des développeurs. Plusieurs tentatives ont été effectuées pour permettre au C++ d'offrir une gestion de mémoire implicite [Boehm et Weiser, 1988] [Bartlett, 1989][Detlefs, 1991]. Des *Garbage Collector* [Boehm et Weiser, 1988] aux librairies [Detlefs, 1991], passant par les *smart pointers* [Detlefs 1992] et les translateurs de C++ [Seliger, 1990], plusieurs stratégies ont été utilisées pour incorporer une gestion mémoire implicite au sein de C++.

L'objectif principal de cette recherche vise à proposer un outil basé sur la programmation par aspects afin d'assurer une gestion mémoire implicite pour C++. L'idée est d'utiliser la programmation par aspects pour simuler les compteurs de références. En effet, pour chaque objet x créé, un compteur de références lui sera attribué. Chaque fois qu'un autre objet y référencera vers x, le compteur de références associé à x sera incrémenté de 1. De même, chaque fois qu'un objet z supprimera sa référence vers x, le compteur de références associé à x sera décrémenté de 1. Finalement, pour chaque objet ayant un compteur de références égal à 0, cet objet n'a plus aucune utilité pour l'application et pourra ainsi être supprimé. Ainsi, l'adjonction des aspects se fera dans le programme à quatre endroits précis :

- au sein du constructeur : dans ce cas, un objet est créé, le compteur de références associé à cet objet y est créé aussi;
- à chaque invocation de la fonction `'new'` : dans ce cas, un objet est créé sans passer par un constructeur directement et il y a création de compteur de références;

- à chaque affectation entre objet `(x = y)` : l'objet y est créé. On lui crée un compteur de références, mais en plus de cela, une référence est ajoutée à l'objet x et on met à jour son compteur de références;
- à chaque invocation de la fonction `'delete'` : dans ce cas, l'objet est supprimé, ses références le seront aussi. Il faudra mettre à jour les compteurs de références et il faudra aussi annihiler l'invocation de cette fonction pour les objets qui ont déjà été détruits à la suite des décrémentations des compteurs.

Nous utilisons à cet effet AspectC++ [AspectC++, 2006] pour élaborer les différents aspects qui gèrent la création des objets et leurs compteurs de références associés (création, incrémentation, décrémentation, suppression) et la suppression d'objet.

Dans une première étape, nous réaliserons cet outil ainsi qu'un autre outil de gestion implicite de la mémoire basé sur les *smart pointers* [Alexandrescu, 2001]. Dans une deuxième étape, nous effectuerons une étude comparative entre les deux méthodes en utilisant les critères suivants :

- Facilité d'utilisation : l'utilisation de l'outil ne doit pas demander une intervention de l'utilisateur.
- Participation de l'utilisateur : on ne doit demander aucune information de la part de l'utilisateur concernant les objets.
- Étendue : l'outil devra gérer tous les types d'objets.
- Coexistence d'une gestion explicite et implicite de la mémoire : l'outil permettra à la fois l'utilisation de la gestion explicite et implicite de la mémoire.

Le chapitre suivant porte sur la description des différentes étapes pour l'élaboration des deux méthodes et leurs comparaisons.

Chapitre 4 : Conception d'un outil de gestion mémoire implicite basé sur la programmation par aspects

4.1 Introduction

Nous avons vu dans le chapitre 2 que plusieurs tentatives d'intégration de gestion implicite au sein de C++ ont été réalisées, notamment dans les travaux de Boehm [1993] qui a utilisé un *Garbage Collector* conservatif, ceux de Detlefs [1991] avec son pré-compilateur associé au *Garbage Collector* de Bartlett [1989], ou encore dans ceux d'Edelson et Pohl [1991] qui ont utilisé les *smart pointers* comme Detlefs [1992]. Detlefs [1992] puis Ellis et Detlefs [1993] ont listé un ensemble de propriétés que doit posséder l'outil de gestion implicite de la mémoire pour C++. Nous présentons dans ce chapitre un outil de gestion mémoire au sein de C++ basé sur la programmation par aspects et plus particulièrement AspectC++ [AspectC++, 2006]. Nous élaborerons aussi, dans un deuxième temps, un autre outil basé sur les *smart pointers* similaire au premier, puisque les deux outils utilisent une approche basée sur les compteurs de références. La troisième partie de ce chapitre portera sur une comparaison entre les deux outils développés.

4.2 Gestion mémoire par aspects

Le but essentiel d'une gestion implicite ou *Garbage Collection* de la mémoire est d'alléger la tâche de programmation et de débogage pour les programmeurs. En effet, avec un tel outil, les programmeurs n'auront pas à se soucier de libérer de l'espace mémoire occupé. De même, ils ne passeront pas de longues heures à déboguer des erreurs liées à la gestion mémoire. Les techniques de *Garbage Collection* sont multiples (compteurs de références, *Mark and Sweep*, recopie...) et les approches diverses (précompilateur, routines, *smart pointers*...). Nous proposons, dans ce travail de recherche, une technique de compteur de références implémenté par la programmation par aspects, et plus précisément avec le langage AspectC++ [AspectC++, 2006].

Contrairement à la programmation orientée objet, où l'objet est l'unique unité de la composition d'un programme, la programmation par aspects permet de regrouper une même préoccupation transverse en un seul aspect et de ne pas la répartir sur plusieurs objets comme la libération de la mémoire. Ainsi en programmation par aspects, les objets ne contiennent que les fonctions essentielles d'un objet donné et cela assure donc une meilleure lisibilité, ainsi qu'une maintenance et une réutilisation plus faciles. Cependant, la programmation par aspects n'est pas un remplacement de la programmation orientée objet, mais plutôt une extension.

4.2.1. AspectC++

AspectC++ [AspectC++, 2006] est un langage orienté aspect en incubation[3] et une extension du langage C++ ; c'est un préprocesseur pour un compilateur C++ usuel. À la suite du tissage *(weaving)*, AspectC++ donne en sortie un code C++ standard. La Figure 4-1 illustre un simple exemple utilisant une classe. Dans cet exemple, il y aura création de deux objets et leur destruction. Nous voulons écrire un aspect qui permet de calculer le nombre de fois que la classe a été instanciée. Ainsi, l'aspect devra calculer le nombre de fois que *new* instanciera un objet, et incrémenter un compteur à chaque fois. Nous allons avant tout survoler quelques concepts fondamentaux de la programmation par aspects avec AspectC++.

```
class Point
  {
    int X; int Y; //les données
    int GetX(void) {return X;}
    int GetY(void) {return X;}
    Point (int NewX=0, int NewY=0) {X=NewX;Y=NewY;} //déclaration
        //interne du constructeur, avec initialisation par défaut
     void Affiche {cout<<X<<'\n'<<Y;}
  };
void Salut()
  {
    cout<<'Salut à vous' <<'\n'
  }
int main ()
```

[3] Plusieurs chercheurs ont davantage contribué au développement d'AspectJ qu'à celui d'AspectC++. Les exemples et les champs de recherches abordés avec AspectJ sont plus nombreux, ce qui le rend plus mature qu'AspectC++.

```
{
   Point *p1 = new Point(10,10); //Création d'un objet
   Salut(); // Appel de la fonction Salut()
   delete (p1); // Libération mémoire objet 1
Point *p2 = new Point(200,200); //Création d'un
                                    //deuxième objet
   Salut(); // Appel de la fonction Salut()
delete (p2); // Libération mémoire objet 2

}
```

Figure 4-1 : Exemple avant tissage.

4.2.1.1 Les points de jointure (Join point)

Les points de jointures sont les endroits dans le code source où les aspects devront être greffés. Un point de jointure peut référer vers des classes, des structures, des unions, des objets ou des méthodes. Plus généralement, un point de jointure est un motif (*pattern*) qui sera reconnu lors de l'exécution du tisseur d'aspect. Dans l'exemple de la Figure 4-1, le motif recherché est `new`. C'est une fonction qui retourne un pointeur sur un objet.

4.2.1.2 Les coupes transverses (Pointcut)

Une coupe transverse est utilisée pour identifier un groupe de points de jointure. Généralement, les coupes transverses sont décrites via les expressions de correspondances (*match expressions*) vues dans la section 3.2.2. Selon la nécessité, les coupes transverses peuvent être :

- un ensemble d'appels `call (point de jointure)`; par exemple `call ("void trilia ()")` collecte l'ensemble des appels à la fonction `trilia`,
- un ensemble d'exécutions `execution (point de jointure)`; par exemple `execution ("void trilia ()")` collecte l'ensemble des implémentations de la fonction `trilia`,
- un ensemble de classe dérivée d'une classe quelconque `derived (point de jointure)`; par exemple `derived ("maclasse ")` collecte toutes les classes dérivées de `maclasse`,
- un ensemble identifiant la portée d'un point de jointure `within (point de jointure)`; par exemple `call ("void trilia ()") && within`

`("maclasse")` collecte toutes les fonctions *trilia* qui ne sont pas des méthodes de la classe dérivée de `maclasse`.

Dans notre exemple, pour avoir le nombre d'appels de la fonction `new`, nous utiliserons la coupe transverse suivante : `call ("new")`.

4.2.1.3 Composition de coupes transverses (Pointcut composition)

Il est possible de combiner différentes coupes transverses en utilisant les opérateurs logiques utilisés usuellement en C++. Supposons que nous voulons étendre la comptabilisation des appels à `new` avec ceux de la fonction `malloc`, sachant que la coupe transverse associée à la fonction `malloc` est `call ("void* malloc (size_t)")`. Nous aurons alors à utiliser la coupe transverse suivante : `call ("new") || call ("void* malloc (size_t)")`. Le « ou » logique (||) combine les deux coupes transverses de *new* et *malloc*. Le résultat de plusieurs coupes transverses combinées est une coupe transverse.

4.2.1.4 Les coupes transverses nommées (Named Pointcut)

Une coupe transverse nommée peut être déclarée pour faciliter l'utilisation de la composition des coupes transverses qui peut s'avérer dans certains cas très complexe. Ainsi la coupe transverse caractérisant la fonction `new` et `malloc` peut être nommée et remplacée par une autre coupe transverse allocation :

```
Pointcut allocation () = call ("new") || call ("void* malloc (size_t)");
```

4.2.1.5 Les expressions de correspondance (Match expressions)

Les expressions de correspondance sont utilisées pour exécuter des filtres spécifiques sur les points de jointure. Elles sont très semblables aux méthodes de signature de C++. Le symbole ("`%`") est utilisé comme un joker pour augmenter la flexibilité des filtres sur les points de jointure. Par exemple le point de jointure `call ("new %")` retournera l'ensemble des appels à `new`, alors que `call ("new Point")` retournera tous les appels pour la création d'un objet `Point`.

4.2.1.6 Les conseils (Advice)

Un conseil est une action activée par un aspect lorsqu'un point de jointure correspondant est atteint dans le programme. L'activation du conseil peut avoir lieu avant, après ou avant et après avoir atteint le code du point de jointure. Le langage AspectC++ permet de spécifier les conseils grâce aux mots clés `before ()` pour avant, `after ()` pour après et `around ()` qui permet de le faire avant et après.

Après avoir survolé les différents concepts de la programmation aspect avec AspectC++, snous pouvons définir l'aspect `new_counter` qui comptabilise le nombre d'occurrences d'appel de la fonction `new`, ce qui nous donnera :

```
aspect new_counter
{
        int x = 0 ;
        public : // liste des coupes transverses
        advice call ("new"): void before ()
        {
              cout<<x++<<endl; // incrémentation de x après avoir
                               // trouvé new
        }
}
```

Ainsi, comme illustré à la

Figure 4-2, nous pouvons voir comment l'aspect `new_counter` est greffé dans l'exemple de la Figure 4-1. Dans cet aspect, le point de jointure `call ("new")` s'effectue à chaque appel de la fonction `new`. Le conseil `before ()` contient la portion de code qui sera exécutée à chaque appel de la fonction `new`. C'est ainsi que nous pouvons comptabiliser le nombre d'occurrences des appels de cette fonction.

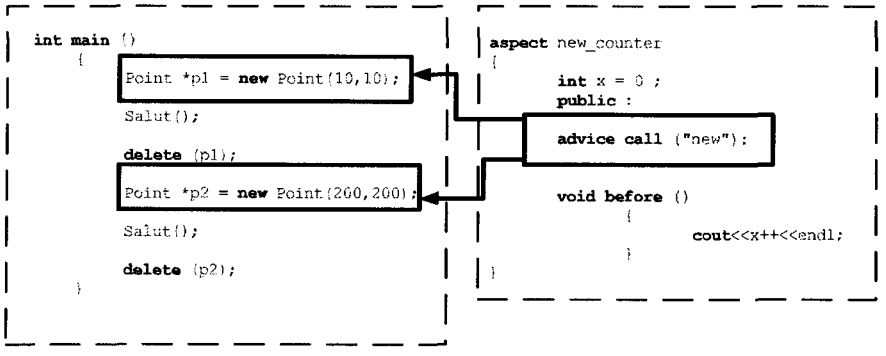

Figure 4-2 : Exemple après tissage.

D'une manière analogue au calcul du nombre d'occurrence de la fonction `new` dans l'exemple de la Figure 4-2, nous procèderons aux calculs des occurrences des appels des constructeurs et des destructeurs des objets pour implémenter nos compteurs de références sur les objets.

4.2.2. Extraction des données

La majeure partie du travail se fera en utilisant les expressions de correspondance (*match expressions*). Ainsi, que ce soit pour extraire toutes les classes présentes, tous les constructeurs, toutes les instanciations, tous les destructeurs ou toutes les affectations, nous utilisons les expressions de correspondance. Dans certains cas, nous utilisons les fonctions membres de la classe *JoinPoint* [AspectC++, 2006] qui est une classe propre à AspectC++ renfermant plusieurs utilitaires et fonctionnalités. La liste suivante énumère les fonctions auxquelles on a eu recours pour l'extraction des données.

- `Classes ("%")`

Retourne tous les constructeurs de classes.

- `const char* signature ()`

Retourne le nom d'une fonction ou d'un attribut dans le cas d'une surcharge d'un opérateur (dans notre cas le " = ").

- `args (type pattern, ...)`

Retourne une liste des types filtrés pour des méthodes ou des attributs avec une expression de correspondance dans le cas d'un constructeur par copie.

- `args (ptr_object)`

Sélectionne tous les appels de tous les points de jointures `delete(...)` (de même pour `free`, `new` et) avec l'argument `ptr_object` dans le cas d'un constructeur par copie ou d'une création d'objet statique.

- `void proceed ()`

Exécute le point de jointure `around ()` pour la coupe transverse dans le cas où on annihile la destruction d'un objet que nous avons déjà détruit via les compteurs de références.

- `thisJointPoint->toString()`

Retourne les noms d'un objet d'une classe quelconque comme dans l'exemple ci-dessous où `classes("%")` retourne toutes les classes.

```
pointcut all_classes()=classes("%");
advice all_classes():void print()
  {
        cout <<"Address of"<<thisJoinPoint->toString()
        <<"object:"<<(void*)this<<endl;
  }
```

Nous utiliserons ces différentes fonctions pour extraire les données relatives à l'implémentation des compteurs de références que nous abordons dans la prochaine section.

4.2.3. Outil de gestion mémoire par aspects

Les compteurs de références ont été utilisés à maintes reprises pour la gestion mémoire des objets que ce soit en systèmes distribués ou non [Lins et Jones, 1996]. Ils ont aussi été utilisés pour d'autres applications comme Adobe Photoshop [Lins et Jones, 1996] ou la distribution de vues en systèmes distribués [Mili *et al.*, 2002] [Mcheick *et al.*, 2006]. L'utilisation répétée des compteurs de références pour la gestion de la mémoire est due essentiellement à la simplicité du procédé. En effet, il s'agit d'attribuer un compteur de références à une ressource (dans notre cas un objet), d'incrémenter ce compteur chaque fois qu'une autre ressource se lie de relation avec (dans notre cas une référence) et de le décrémenter chaque fois qu'une relation est supprimée.

Il s'agit donc de comptabiliser toutes les créations d'objet et leur destruction et de gérer tous les compteurs de références relatifs. Le procédé est le suivant : tout d'abord, on parcourt le code à la recherche de la déclaration de toutes les classes créées et utilisées par l'utilisateur. Ensuite, dans une deuxième phase, on maintient une table qu'on appelle table d'objets et qui contient toutes les classes et les instanciations d'objets. À chaque instanciation d'un objet statiquement, par affectation ou par les constructeurs qui seront maintenus dans la table, le compteur sera incrémenté. Après, si on trouve un appel à un destructeur ou un `delete` (ou un `free`), on met à jour tous les compteurs associés à la suppression de cet objet. Enfin, on supprime les objets pour lesquels les compteurs sont à

zéro et on annihile les destructeurs pour lesquels les objets ont déjà été détruits. Les classes représentent les racines à partir desquelles on parcourt tous nos objets instanciés.

Le Tableau 3 représente la structure de la table objet à maintenir. Il contient les huit champs suivants :

- Le champ *Nom_obj* représente le nom de l'objet ou de la classe. Il est retourné par un conseil comportant la fonction membre de la classe JointPoint `thisJointPoint->toString()`, qui est retourné par la coupe transverse `classes("%")` qui lui retourne toutes les classes.
- Le champ *Cls* est une variable booléenne qui détermine si c'est une classe ou non.
- Le champ *Cpt_obj* représente le compteur de références associé à cet objet.
- Le champ *List_ref_a* représente la liste des objets qui réfèrent à cet objet ou à la classe. *List_ref_de* représente la liste des objets pour lesquels cet objet maintient une référence. La mise à jour de ces deux champs se fait à chaque instanciation d'un objet référant à cet objet ou pour lequel cet objet y réfère.
- Le champ *Pro_const* est une liste contenant les prototypes de tous les constructeurs existants associés à cet objet retournés par la fonction `Classes ("%")` où `Classes` représente le nom de toutes les classes trouvées. On utilisera aussi la fonction membre de la classe `JointPoint : Args (type pattern, …)` pour retourner toutes les possibilités des constructeurs.
- Le champ *Pro_dest* est le prototype du destructeur associé à cet objet retourné par la fonction `Args (ptr_object)` qui est aussi une fonction membre de la classe `JointPoint`. *Sur_op* est une variable booléenne indiquant s'il y a lieu ou non de la surcharge de l'opérateur " = ", retourné par la fonction `const char* signature ()` où signature est le nom de l'objet.

Tableau 3 : Structure de la table d'objets

Nom_obj	*Cls*	*Cpt_obj*	*List_ref_a*	*List_ref_de*	*Pro_const*	*Pro_dest*	*Sur_op*

La liste des objets référencés *List_ref_a* a pour fonction de maintenir la liste des objets référant à cet objet ou à une classe. Ainsi chaque fois qu'un objet ajoute une référence vers notre objet, il sera ajouté à la liste tout en incrémentant le compteur. Cette liste est maintenue pour la gestion des cycles et, associée avec les objets, les deux structures forment un graphe d'objets. L'ensemble des nœuds est représenté par les noms des objets, tandis que *List_ref_a* représente la liste d'adjacence associée à chaque nœud. Comme *List_ref_a*, *List_ref_de* sert à mettre à jour les compteurs de référence. Si la première est utilisée pour un chaînage avant c'est-à-dire pour déterminer les objets référant à cet objet, la deuxième est utilisée pour un chaînage arrière et une remontée à la source dans le graphe d'objet. Si un objet venait à être supprimé, les compteurs des objets lui maintenant une référence seront tous décrémentés. Dans la Figure 4-3, la suppression de l'objet A, qui est une instanciation de la classe Cl entraînera la décrémentation du compteur de B, qui à son tour entraînera la décrémentation des compteurs respectifs de C et D puisque celui de B est devenu à 0. Dans une deuxième étape, les compteurs respectifs de E et F seront décrémentés.

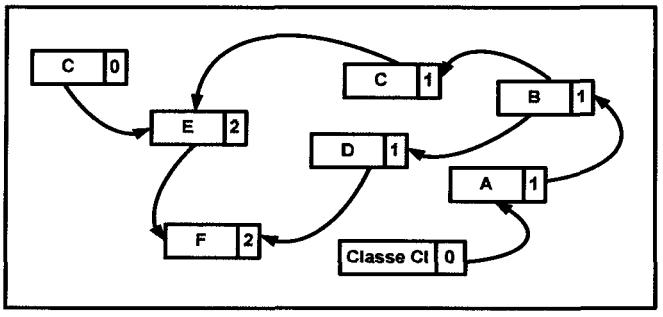

Figure 4-3 : Exemple de compteurs de références

4.2.4. Exemple

Pour illustrer le fonctionnement de l'outil de gestion mémoire par aspects, nous appliquons notre technique sur l'exemple de la Figure 4-4. Dans un premier temps, on parcourt la classe `Point` (L1 à L9). Le champ *Cls* est à vrai, *Sur_op* est à vrai et nous avons un prototype de constructeur *Pro_const* : `Point(int,int)`. Après le parcours de la

classe `Pixel` (L10 à L18), on aura dans la table d'objet *Cls* à vrai, *Sur_op* est à faux et nous avons un prototype de constructeur *Pro_const* : `Point(int,int,int)`. Ces deux classes ne contiennent pas de destructeurs. Nous obtiendrons alors la table d'objet du Tableau 4.

```
L1      class Point {
L2              protected: int X ; intY;
L3              public :
L4              int GetX(void) {return X;}
L5              int GetY(void) {return Y;}
L6              Point (int NewX=0, int NewY=0) {X = NewX; Y = NewY;}
L7              Point operator+ (Point &P1, Point &P2)
              {Point res(P1.GetX()+P2.GetX(),P1.GetY()+P2.GetY(),P1);
                return res;}
L8              Point operator = (Point &P);
                {Point res; res.X = this.X; res.Y=this.Y; return
        res;}
L9      };
L10     class Pixel : public Point {
L11             protected:    int couleur;
L12             public :
L13             Pixel (int nx,int ny,int coul=0):Point(nx,ny)
L14             Pixel    (int    nx,int    ny,int    coul):Point(nx,ny)
        {couleur=coul;}
L15             void allume(void); {g_pixel(X,Y,couleur);} //g_pixel
        : //une fonction qui allume un pixel à l'écran
L16             void              allume(int              couleur);
        {g_pixel(X,Y,couleur=coul);}
L17             void eteind(void); {allume(0);}
L18     };
L19     void main(void){
L20     Point P1(25,25);
L21     Point P2(20,20);
L22     Point P3=P1+P2;
L23     int coordX = P1.GetX();
L24     Pixel *ptrPixel1 = new Pixel(100,100,1);
L25     Pixel *ptrPixel2 = new Pixel(200,200,2);
L26     Pixel *ptrPixel3 = new Pixel(300,300,3);
L27     Pixel *ptrPixel4 = new Pixel(400,400);
L28     …
L29     ptrPixel1->allume();
L30     ptrPixel1->eteind();
L31     …
L32     delete P1;
L33     delete ptr1Pixel;
L34     delete ptr2Pixel;
L35     }
```

Figure 4-4 : Exemple C++ pour compteurs de références

Tableau 4 : Table d'objets après parcours des classes

Nom_obj	*Cls*	*Cpt_obj*	*List_ref_a*	*List_ref_de*	*Pro_const*	*Pro_dest*	*Sur_op*
Point	vrai	-	-	-	Point(int,int)	-	vrai
Pixel	vrai	-	-	-	Pixel(int,int,int)	-	faux

Les lignes L20 et L21 représentent une instanciation statique de deux objets `Point`. Les compteurs de références *Cpt_obj* associés à `P1` et `P2` sont initialisés et la liste *List_ref_a* de la classe `Point` sera mise à jour en y ajoutant les objets `P1` et `P2`. Nous obtenons la table d'objets du Tableau 5.

Tableau 5 : Table d'objets après instanciation de P1 et P2

Nom_obj	*Cls*	*Cpt_obj*	*List_ref_a*	*List_ref_de*	*Pro_const*	*Pro_dest*	*Sur_op*
Point	vrai	-	P1, P2	-	Point(int,int)	-	vrai
Pixel	vrai	-	-	-	Pixel(int,int,int)	-	faux
P1	faux	0	-	-	-	-	-
P2	faux	0	-	-	-	-	-

La ligne L22 représente une instanciation avec affectation d'un objet de type `Point`. Le compteur de `P3` est initialisé et ceux de `P1` et `P2` sont incrémentés. De même, `P3` est ajouté à la liste *List_ref_a* de la classe `Point`. Les lignes L24 à L27 représentent la déclaration de pointeurs sur des objets `Pixel`. À la ligne L32, l'objet `P1` est supprimé ainsi que les pointeurs sur les objets `ptrPixel1` et `ptrPixel2`. Le Tableau 6 illustre la table d'objets à la fin du programme après la suppression de ces trois objets.

À la fin du programme, les compteurs de références des objets `ptrPixel3` et `ptrPixel4` sont à 0 et ils sont alors supprimés. De même, le compteur associé à `P3` est à 0 et `P3` est ainsi supprimé. Avant la suppression de `P3`, on parcourt sa liste *List_ref_de* pour déterminer les objets qui réfèrent vers `P3`. `P2` fait partie de la liste *List_ref_de* de `P3`, le compteur de références associé à `P2` est alors décrémenté .Ce dernier tombe à 0 et l'objet `P2` est à son tour supprimé. Dans le cas où il y aurait une suppression de l'objet `P3` avec *delete*, puis une autre suppression de `P2`, l'appel à *delete* pour l'objet `P2` serait annihilé par l'intermédiaire de la fonction `void proceed ()` qui exécutera le conseil

`around()`, pour la simple raison que son compteur de références serait à 0 et que l'objet serait détruit.

Tableau 6 : Table d'objets à la fin du programme

Nom_obj	*Cls*	*Cpt_obj*	*List_ref_a*	*List_ref_de*	*Pro_const*	*Pro_dets*	*Sur_op*
Point	vrai	-	P2, P3	-	Point(int,int)	-	vrai
Pixel	vrai	-	ptrPixel2, ptrPixel4	-	Pixel(int,int,int)	-	faux
P2	faux	1	P3	-	-	-	-
P3	faux	0	-	P2	-	-	-
ptrPixel3	faux	0	-	-	-	-	-
ptrPixel4	faux	0	-	-	-	-	-

Dans la prochaine section, nous abordons la gestion des cycles qui peuvent survenir avec la gestion des compteurs de références.

4.2.5. Gestion des cycles

L'inconvénient principal des compteurs de références est l'incapacité d'une telle approche à déterminer les cycles [Lins et Jones, 1996]. À la Figure 4-5, les objets A et B ont tous deux leurs compteurs de références respectifs à 1. Cependant, la seule référence de A est une référence vers B, et la seule référence de B est une référence vers A. On dit alors que nous avons un cycle. Les objets A et B ne sont d'aucune utilité pour l'application, mais la technique de compteurs de références ne peut les supprimer puisque leurs compteurs ne sont pas à 0, comme dans le cas d'une *Forward declaration* [Stroustrup, 2004] où deux classes se référencent mutuellement. Nous utilisons l'algorithme de recherche en profondeur *Depth First Search* (DFS) [Cormen *et al.*, 2002], qui, à partir de la table d'objets créés et en parcourant les différentes listes d'adjacence formées par les listes des objets référés *List_ref_a* dans la table d'objet, pourra déterminer les cycles. Tous les objets appartenant à un cycle pour lesquels les compteurs de références sont à 1 seront supprimés.

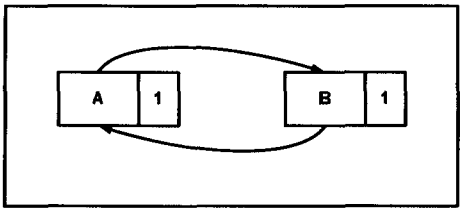

Figure 4-5 : Compteurs de références et cycle

Pour alléger l'exécution de l'outil, la recherche des cycles se fera une seule fois à la fin du parcours du graphe d'objet formé par la table. L'algorithme de la Figure 4-6 est un pseudo-code permettant de récupérer les nœuds d'un cycle trouvé. Après avoir traité séquentiellement un cycle trouvé, l'algorithme est réitéré pour trouver les autres cycles. La liste critique de nœuds représente le cycle en cours de recherche. La complexité de cet algorithme est : $\Theta(|E|+|V|)$, où $|V|$ représente la cardinalité des sommets et $|E|$ la cardinalité des arcs de notre graphe formé à partir de la table des objets.

```
Tant que cycle existant faire
 {
  void DFS(int v) // v étant le sommet de départ
        {
              pour i allant de 1 à n faire
              initialiser marquer [i] = 0;
              marquer [v] = 1
              Pour chaque sommet w adjacent à v faire
              si marquer [w] = 0 alors DFS(w)
              finpour
        }


  Pour tout sommet s
        {
              Si s n'a pas encore été visité
              alors
              visiter s
              finsi
              finpour
        }
  retourner pas de cycle
  visiter s :
  si s est dans la liste critique
  alors
  retourner le cycle
  sinon
  ajouter s à la liste critique;
  visiter tous les successeurs de s;
  enlever s de la liste critique
  finsi

Pour tout sommet s appartenant à un cycle
```

```
{
        Si compteur (s)= 1
        alors
        supprimer s
        sinon
        décrémenter (s.compteur)
        finsi
        finpour
  }

}
```

Figure 4-6 : Algorithme de recherche et résolution des cycles

Pour la résolution de ce problème, l'idée est la suivante : on vérifie les compteurs de références des objets appartenant à un cycle donné. Si le compteur d'un objet donné est égal à 1, cet objet est alors supprimé et écarté du cycle, sinon le compteur sera décrémenté et l'objet sera gardé.

4.2.6. Problèmes et limites rencontrés

Plusieurs problèmes ont été rencontrés pour l'élaboration de cet outil de gestion mémoire implicite pour C++ basé sur la programmation par aspects. Le premier problème rencontré est lié au tisseur d'AspectC++ : certains fichiers entête (*header files*) ne sont pas chargés à partir de C++ dû à la configuration de PUMA [AspectC++, 2006], un utilitaire qui accompagne le tisseur AspectC++ et qui ne prend pas en charge tous ces fichiers entête. Le deuxième problème rencontré est l'incapacité, dans certains cas, de prendre en charge les classes *templates* de la part du tisseur. Comme nous l'avons mentionné auparavant, le tisseur est encore en phase d'incubation et la stabilité n'est pas totalement assurée. Ainsi certaines fonctions membres de la bibliothèque `iostream` (comme `cin`, `cout`, etc.) ne sont pas reconnues lors de l'utilisation des classes *templates* et nous ne pouvons donc pas retourner certaines données ou certains résultats directement par des flux. Le troisième problème consiste en l'inexistence des points de jointure liés aux fonctions `new` et `delete`. Il nous a donc fallu regrouper plusieurs autres points de jointures pour remédier à ce manque[4]. L'idée était de concaténer `new` (ou `delete`) avec le nom de l'objet pour obtenir la chaîne de caractères à rechercher. Le dernier problème est

[4] La version 0.9 d'AspectC++ avec laquelle nous avons effectué nos travaux ne gère pas les points de jointure liés à *new* et *delete*. Cependant, la dernière version 1.0.3 les prend en charge.

l'incapacité du tisseur AspectC++ à reconnaître directement les constructeurs et les destructeurs d'une classe donnée. Pour palier à ce problème, nous avons aussi hybridé certains points de jointure pour pouvoir déterminer les différents constructeurs et destructeurs.

L'outil de gestion implicite de la mémoire par aspects pour C++ présente essentiellement trois principales limites. La première limite est liée au compilateur : notre outil ne prend pas en compte les fichiers entête vu l'incapacité d'AspectC++ de charger certains fichiers entête. La deuxième limite est le fait que notre outil ne gère que les objets créés par l'utilisateur ou l'application légataire. Par la suite, tous les objets créés par le système ou par le compilateur ne sont pas pris en charge par notre outil. La troisième limite est liée à l'adjonction de l'algorithme traitant les cycles. Ce dernier augmente les consommations temporelles et spatiales en appliquant des algorithmes qui sont reconnus d'être gourmands en consommation mémoire [Cormen *et al.*, 2002].
Nous explorons dans la prochaine section le deuxième outil basé sur les *smart pointers*.

4.3 Outil à base des *smart pointers*

Les *smart pointers* [Alexandrescu, 2001] sont des objets C++ qui simulent les pointeurs en surchargeant les opérateurs « * » et « —> ». Les *smart pointers* ont été utilisés pour plusieurs tâches, mais la gestion mémoire et le blocage d'accès restent les deux applications les plus répandues et ce, en libérant l'espace mémoire consommé par les objets d'une application donnée [Alexandrescu, 2001]. Les *smart pointers* ont été utilisés par Edelson et Pohl [1991] pour implémenter un collecteur par recopie. Detlefs [1992] les utilisa pour implémenter une interface qui représente un *Garbage Collector* appliqué à un collecteur par compteur de références. Edelson [1992] implémenta aussi un *Garbage Collector Mark and Sweep* se basant sur les *smart pointers* pour alléger son précédent algorithme [Edelson et Pohl, 1991]. Les compteurs de références restent cependant la stratégie de gestion mémoire utilisant les *smart pointers* la plus populaire [Alexandrescu, 2001]. Le principe des compteurs de références est le même : on associe

un compteur à chaque objet puis on l'incrémente et on le décrémente à chaque création de référence ou de suppression tant qu'il est différent de 0.

La Figure 4-7 présente deux techniques différentes pour implémenter les *smart pointers*. Dans la première technique, on associe un compteur de références à chaque référence alors que dans le deuxième cas on associe un compteur global à chaque objet pour comptabiliser toutes les références qui y pointent. La deuxième technique demande moins d'espace mémoire, mais elle est plus compliquée à gérer. Nous utilisons pour notre outil la première méthode où chaque objet possède son propre compteur de références par besoin de similarité entre les deux outils d'aspects et de *smart pointers*. D'un autre côté, pour implémenter les *smart pointers*, il est préférable d'utiliser des classes *template*, car ces derniers peuvent simuler plusieurs types de pointeurs, mais aussi parce que le code est générique pour toutes les classes [Coplien, 1992]. Pour la gestion des *smart pointers*, il faudra non seulement prévoir des constructeurs bruts et par recopie pour chaque *smart pointer*, mais aussi des destructeurs [Alexandrescu, 2001] pour gérer les appels de fonctions [Stroustroup, 2001]. Il faudra également prévoir une méthode pour l'accès aux méthodes d'un *smart pointer* pour assurer l'encapsulation des fonctionnalités [Coplien, 1992].

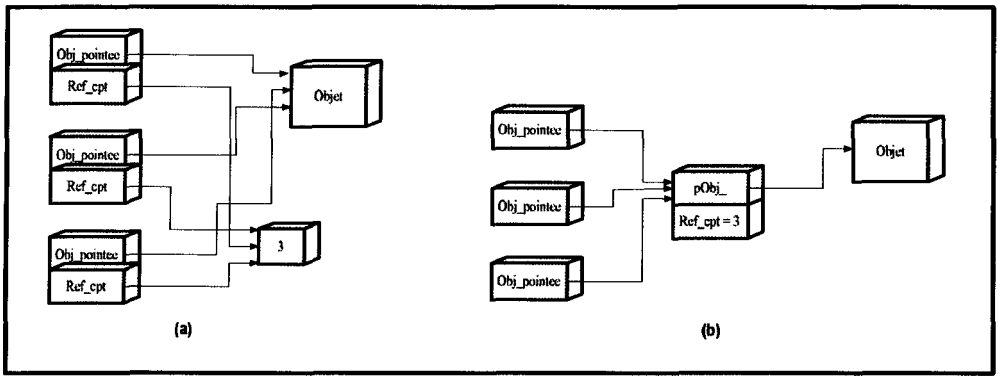

Figure 4-7 : Compteurs de références et *smart pointers*

Nous allons donc créer des objets qui se comportent comme des pointeurs ordinaires mais qui, grâce à la redéfinition d'opérateurs, gèrent automatiquement leur destruction. L'idée est d'utiliser une super-classe `SmartPointable` pour toutes les classes

d'objets dont l'allocation et la destruction seront gérées par notre outil. Cette classe contient un compteur de références `refcount` et trois méthodes. La première méthode `void inc()`, incrémente le compteur de références et est appelée quand un autre objet crée une référence sur celui-ci. La deuxième méthode, `void dec()`, décrémente le compteur de références et si ce compteur est égal à 0, l'objet est détruit. Cette méthode est appelée lorsqu'une référence sur notre objet est supprimée. La troisième méthode `cptref()`, retourne la valeur du compteur de références. Il faudra aussi déclarer un destructeur pour cette classe pour que les *dynamic_cast* fonctionnent sur des pointeurs à cette classe. Une deuxième classe `SmartPointer` sera utilisée pour représenter un pointeur dit *smart*, c'est-à-dire sur un objet de type `SmartPointable`. Les opérations habituelles comme l'affectation, vont automatiquement faire des appels aux fonctions d'incrémentation et décrémentation des compteurs de références. Cette classe sera implantée sous forme de classe *template* qui redéfinit tous les constructeurs et les opérateurs appropriés pour assurer une bonne gestion des compteurs de références. Nous aurons besoin de constructeurs vides (`SmartPointer<T>()`) qui servent à initialiser un *smart pointers*, et de constructeurs par copie, l'un brut (`SmartPointer<T>(T*)`) qui crée un pointeur sur un *smart pointers* et le deuxième pour la même classe qui crée un autre *smart pointers* (`SmartPointer<T>(SmartPointer<T>&)`). Nous aurons aussi besoin de surcharger l'opérateur d'affectation « = » avec `SmartPointer<T>& operator=(T*)` et l'opérateur « * » avec la fonction `T& operator*()` pour simuler les pointeurs. L'accès aux méthodes de cette classe se fait par la fonction `T* operator->()`et `T& operator*()` pour obtenir l'objet lui-même. Finalement, pour vérifier si un pointeur est nul ou non, il ne faut pas faire « `if (!smartpointer)` » ou « `if (smartpointer==NULL)` » [Alexandrescu, 2001], mais plutôt via un accès direct à la fonction membre `bool isNull()` en faisant « `if (smartpointer.isNull()` » puisqu'il ne s'agit plus de pointeur.

Ainsi, pour toute application, l'utilisateur devra dériver toutes ces classes de la classe `SmartPointer` qui hérite elle-même de la classe `SmartPointable` et les *smart pointers* se chargeront de la gestion de la libération de la mémoire. On parle alors de technique de *wrapping* [Alexandrescu, 2001] comme illustré à la Figure 4-8. Chaque

objet créé par l'utilisateur sera emboîté dans un autre objet qui contient un compteur de références et les méthodes utilisées pour la gestion de ces compteurs.

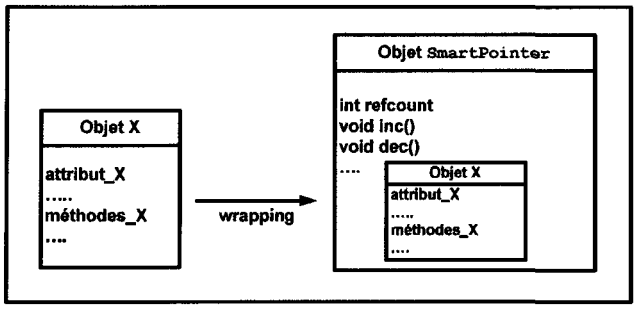

Figure 4-8 : Technique de *wrapping*

Dans la section suivante, nous entamons une comparaison entre les deux outils.

4.4 Comparaison et discussion

Nous procédons, dans cette section, à la comparaison des deux outils de gestion mémoire implicite proposés. Le premier, décrit à la section 4.2, est basé sur la programmation par aspects et le deuxième, décrit dans la section 4.3, est basé sur les *smart pointers* et la technique de *wrapping*. Pour effectuer cette comparaison, nous nous basons sur les propriétés énumérées par Detlefs [Detlefs, 1992] pour l'adjonction d'un outil de gestion mémoire implicite pour le langage C++.

4.4.1. Facilité d'utilisation et participation de l'utilisateur

La première propriété à vérifier est la facilité de l'utilisation de l'outil et la participation de l'utilisateur. Concernant l'outil de gestion par aspects, il ne demande aucune intervention de l'utilisateur pour avoir des informations sur les objets traités et les variables systèmes. Il n'y a aucune modification de code pour les applications légataires, et les futures applications n'y procéderont pas. Contrairement à l'outil par aspects, l'outil basé sur les *smart pointers* ne peut traiter des applications légataires sans passer par des modifications de code. Pour une utilisation de cet outil, l'utilisateur devra modifier tout son code, puisque tous les objets créés devront hériter de la superclasse `SmartPointable`.

4.4.2. Coexistence d'une gestion implicite et explicite de la mémoire

Afin d'assurer une bonne utilisation du langage C++, l'outil de gestion mémoire implicite devra assurer à la fois la gestion explicite et implicite de la mémoire. Ainsi, l'outil devra permettre à l'utilisateur d'user des différentes fonctions pour récupérer la mémoire au sein de C++ comme `delete` ou `free`. L'outil de gestion mémoire par aspects permet une telle coexistence. L'utilisateur pourra en tout temps récupérer la mémoire explicitement, mais l'outil permet aussi d'annihiler les appels à la libération de la mémoire pour lesquels la récupération a déjà été faite via notre outil. L'outil basé sur les *smart pointers* ne libère l'espace mémoire que lorsqu'un compteur de références tombe à 0. Ainsi, l'utilisateur ne peut libérer de la mémoire en tout temps. Cet outil ne peut donc assurer une gestion explicite de la mémoire en tout temps. De ce fait, l'outil par aspects permet de faire appel aux destructeurs d'objets en tout temps et non celui à base des *smart pointers*.

4.4.3. Étendue

Un outil de gestion mémoire implicite pour C++ devra traiter tous les types d'objets. Si l'outil par aspects traite tous les types d'objets[5], l'outil basé sur les *smart pointers* ne traite pas certains types d'objets comme les pointeurs constants ou tableaux constants. En effet, les *smart pointers* ne peuvent différencier `const T*` et `T* const` et les traitent de la même manière [Edelson, 1992]. De même les *smart pointers* souffrent d'une incapacité à supporter l'héritage multiple [Edelson, 1992].

4.4.4. Appui compilateur

Les deux outils ne nécessitent aucun appui ou aucune information de la part du compilateur. AspectC++ interagit directement avec le compilateur pour produire le code après tissage[6] sans intervention de l'utilisateur. L'outil à base des *smart pointers* est du C++ qui peut être pris en charge par tous les compilateurs C++.

---

[5] AspectC++ ne traite pas encore correctement les classes *templates*, ensuite notre outil ne pourra pas les traiter.

[6] AspectC++ est compatible avec la plupart des compilateurs C++ [AspectC++, 2006].

4.4.5. Discussion

Notre outil de gestion implicite de la mémoire pour C++ vérifie la totalité des contraintes émises par Detleft [Detleft, 1992]. Cependant, notre outil ainsi que celui basé sur les *smart pointers* n'ont pas été testés pour des optimisations de compilateur. Le deuxième problème est évoqué à la section 4.2.6 : l'outil basé sur les aspects ne gère que les objets créés par l'utilisateur. Ainsi, il ne prend en charge aucun type de base du compilateur comme les *int, char, string*, etc. De même, l'outil ne prend pas en charge les objets systèmes créés par le compilateur, comme ceux créés pour les *namespace*.

Nous ne pouvons pas établir une comparaison entre les deux outils du point de vue de la consommation temporelle et spatiale. En effet, l'exécution de l'outil basé sur les aspects requiert à la fois, un temps d'exécution et un espace mémoire supplémentaire pour effectuer le tissage[7].

4.5 Conclusion

Dans ce chapitre, nous avons détaillé, dans une première étape, la réalisation d'un outil de gestion de mémoire implicite des objets pour C++ par aspects. Nous avons utilisé pour cela le tisseur AspectC++ [AspectC++, 2006]. Nous avons utilisé la technique des compteurs de références pour gérer le cycle de vie des objets créés par l'utilisateur. Ainsi à chaque création d'un objet, l'outil lui affecte un compteur de références et gère l'incrémentation et la décrémentation d'une part et la destruction de l'objet d'une autre part. Il suffit de parcourir le code source de l'utilisateur et de détecter toutes les déclarations des classes et les instanciations des objets respectifs et ce, avec des techniques d'extraction de données du code. Une table d'objet est utilisée pour la gestion des compteurs de références relatifs à chaque objet créé par l'utilisateur.

Dans une deuxième étape, nous avons décrit une technique de gestion implicite de la mémoire pour C++ se basant sur les *smart pointers*. Pour cette technique, il suffisait de faire hériter toutes les classes de l'utilisateur d'une classe `SmartPointable` qui permet de

[7] Les dernières versions d'AspectC++ tendent à optimiser les consommations spatiale et temporelle [AspectC++, 2006]

gérer des compteurs de références associés à chaque objet. Nous avons procédé dans la dernière partie de ce chapitre, à une comparaison des deux outils. Si le premier répond à la totalité des contraintes de Detlefs [1992], le deuxième, par contre, n'en répond qu'à deux. En plus, l'outil basé sur les *smart pointers* ne gère pas les applications légataires et pour pouvoir l'utiliser, il faudrait réécrire une bonne partie du code de l'utilisateur.

Chapitre 5 : Conclusion

Les langages tels Java, Lisp, Eiffel, Modula III sont des langages orientés objet qui ont gagné leur popularité grâce à la gestion automatique de la mémoire assurée par l'entité *Garbage Collector* ou ramasse-miettes. Contrairement à ces langages, le langage C++ utilise une gestion mémoire explicite avec les *delete, new, free* et *malloc* puisqu'il est une extension de l'ANSI-C. Le langage C++ est l'un des langages de programmation les plus utilisés actuellement vu sa facilité d'utilisation et son efficacité. De plus, ses caractéristiques en font un langage idéal pour certains types de projets. Cependant, une gestion mémoire explicite pour d'amples projets peut s'avérer très coûteuse du point de vue du débogage des problèmes liés à la gestion mémoire et plus précisément à la libération de la mémoire.

Plusieurs tentatives ont été effectuées pour ajouter un outil de gestion mémoire implicite au langage C++. Bartlett [1989], Detlefs [1991], Ferreira [1991], Edelson et Pohl [1991], Boehm [1993] et Boehm [2002] sont les principaux chercheurs qui ont implémenté des *Garbage Collector* pour C++. Si certains ont utilisé les techniques de *Garbage Collection* usuelles (*Mark and Sweep* et algorithmes conservatifs) comme Bartlett et Boehm, d'autres comme Edelson et Pohl ont utilisé les *smart pointers* pour implémenter leur *Garbage Collector*. D'un autre côté, Detlefs [1992] puis Ellis et Detlefs [1993] ont énuméré certaines conditions qui devaient être vérifiées pour tout outil de gestion mémoire implicite dédié au langage C++. Malheureusement, pour plusieurs de ces travaux, ces conditions n'étaient pas totalement vérifiées.

Dans ce travail de recherche, nous avons proposé un outil de gestion mémoire implicite pour le langage C++ basé sur la programmation par aspects. Cette dernière est un paradigme de programmation qui permet de réduire fortement les couplages entre les différents aspects techniques d'un logiciel. On parle alors de préoccupations transverses. Ainsi, ce paradigme permet de différencier les aspects fonctionnels et non fonctionnels d'une application, ce qui permettrait de contourner certaines limites du modèle de programmation par objet. L'approche principale de ce paradigme est de regrouper une même préoccupation transverse en un seul aspect

et de ne pas la répartir sur plusieurs objets comme la libération de la mémoire. Nous avons utilisé cette propriété pour élaborer un outil de gestion implicite de la mémoire en regroupant l'aspect mémoire en un seul aspect. Pour la gestion du cycle de vie des objets, notre outil simule les compteurs de références. En effet, nous attribuons un compteur de références à chaque objet créé par l'utilisateur. Ce dernier sera incrémenté à chaque création d'un objet référant à l'objet associé au compteur. De même, il sera décrémenté à chaque suppression d'une référence à cet objet. Tout objet dont le compteur de référence associé tombe à 0 sera supprimé et une mise à jour des différentes références pointant vers cet objet sera effectuée. L'outil permet aussi d'annihiler les *delete* introduits par l'utilisateur, mais qui ne sont d'aucune nécessité lorsqu'un compteur tombe à 0. De même, un algorithme DFS a été incorporé à l'outil pour surmonter le problème des cycles lié aux compteurs de références [Jones et Lins, 1996]. Nous avons élaboré aussi un outil de gestion de cycles de vie des objets pour C++ basé sur les *smart pointers* [Alexandrescu, 2001].

Ce présent travail est composé en cinq chapitres. Le premier chapitre a présenté une introduction. Dans le deuxième chapitre, nous avons survolé les différentes techniques de gestion mémoire implicite et explicite, ainsi que celles utilisées pour le langage C++. Dans le troisième chapitre, nous avons détaillé les principes de la programmation par aspects, en particulier ceux de la programmation par aspects avec le tisseur d'aspects AspectC++ [AspectC++, 2006]. Le quatrième chapitre décrit les différentes étapes de l'élaboration de l'outil basé sur les aspects ainsi qu'une description de l'outil basé sur les *smart pointers*. De même, nous avons effectué une comparaison des deux outils selon les différentes propriétés énumérées par Detlefs [1992]. Le cinquième chapitre présente une conclusion à ce travail.

L'outil basé sur les aspects a répondu à tous les critères émis par Detlefs, sauf pour celui de l'optimisation qui n'a pas été essayé. L'outil basé sur les *smart pointers* a failli à plusieurs de ces critères. L'inconvénient principal de cet outil est sans doute la nécessité de réécriture de la majorité du code pour les applications légataires, contrairement à l'outil par aspects pour lequel on n'a nul besoin de réécrire le code puisqu'il est destiné aux applications légataires. Ce dernier possède aussi un inconvénient : il ne traite que les objets créés par l'utilisateur. En effet l'outil par aspects ne peut traiter les objets systèmes créés par le compilateur et les objets des types de

base comme les *int, char, long*, etc. D'autre part, cet outil ne peut pas non plus traiter les classes *templates* et ce, dû à l'incapacité du tisseur AspectC++ à les traiter.

Concernant les travaux futurs, il est possible, dans un premier temps, d'améliorer l'outil basé sur les aspects et d'augmenter son étendue pour traiter tous les types d'objets, ceux créés par l'utilisateur, mais aussi ceux créés par le compilateur et les différents types de base. Dans un deuxième temps, annihiler toutes les suppressions d'objets et de rendre ainsi l'outil le seul responsable de la mise à terme du cycle de vie des objets pour une application donnée. De ce fait, l'utilisateur n'aura plus à se soucier de la libération de l'espace mémoire. Pour ce qui est des *smart pointers*, il est possible de créer un outil qui génère les *smart pointers* associés à chaque objet automatiquement sans la nécessité de réécrire le code de l'application. Dans ce cas, on parle de *decorator* qui représente une technique avancée de *wrapping* passant par des délégations et des agrégations comme celles utilisées dans la plateforme SPRING [Spring, 2006].

Bibliographie

[Abdullahi et Ringwood, 1998] S. E. Abdullahi and G. A. Ringwood. Garbage collecting the Internet: a survey of distributed garbage collection. *ACM Computing Surveys*, 30(3):330-373, September 1998.

[Aho *et al.*, 1988] A. V. Aho, B. W. Kernighan, and P. J. Weinberger, *The AWK Programming Language*. Addison-Wesley, 1988.

[Alexandrescu, 2001] A. Alexandrescu, Modern C++ Design, Addison Wesley, 2001.

[Appel, 1992] A. W. Appel, Compilers and runtime systems for languages with garbage collection, *Proceedings of SIGPLAN'92 Conference on Programming Languages Design and Implementation*, volume 27 of *ACM SIGPLAN Notices*, San Francisco, CA, June 1992.

[AspectC++, 2006] AspectC++ Project, http://www.aspectc.org/fileadmin/documentation/ac-languageref.pdf, Mars 2006.

[AspectJ, 2001] The AspectJ Organization, "Aspect-Oriented Programming for Java." http://www.aspectj.org , 2001.

[AspectWerkz, 2006] BEA Systems, http://aspectwerkz.codehaus.org/, Chicago, Illinois, USA, 2005.

[Baecker, 1970] H. D. Baecker, Implementing the Algol-68 heap. *BIT*, 10(4):405-414, 1970.

[Baecker, 1972] H. D. Baecker, Garbage collection for virtual memory computer systems. *Communications of the ACM*, 15(11):981-986, November 1972.

[Baker, 1979] H. G. Baker, Optimizing allocation and garbage collection of spaces in MaclLisp. In Winston and Brown, editors, *Artificial Intelligence: An MIT Perspective*. MIT Press, 1979.

[Bartlett, 1989] J. F. Bartlett, Mostly-Copying garbage collection picks up generations and C++. Technical note, DEC Western Research Laboratory, Palo Alto, CA, October 1989.

[Bergmans et Aksit, 2001] L. Bergmans and M. Aksit, Composing Crosscutting Concerns Using Composition Filters, Communications of the ACM, Vol. 44, No. 10, pp. 51-57, October 2001.

[Bobrow, 1980] D. G. Bobrow, Managing re-entrant structures using reference counts. *ACM Transactions on Programming Languages and Systems*, 2(3):269-273, July 1980.

[Boehm et Weiser, 1988] H. J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807-820, 1988.

[Boehm, 1993] H.J. Boeh, Space efficient conservative garbage collection. *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, volume 28(6) of *ACM SIGPLAN Notices*, Albuquerque,

NM, ACM Press, June 1993.

[Boehm, 2002] H. Boehm, "Bounding Space Usage of Conservative Garbage Collectors", *Proceeedings of the 2002 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, , pp. 93-100 Jan. 2002.

[Bray, 1977] O. H. Bray, Data management requirements: The similarity of memory management, database systems, and message processing, Proceedings of the 3rd workshop on Computer architecture : Non-numeric processing, ACM Press New York, NY, USA, Pages: 68 – 76, 1977.

[Broberg *et al.*, 2004] Niklas Broberg , Andreas Farre , Josef Svenningsson, Regular expression patterns, ACM SIGPLAN Notices, v.39 n.9, September 2004.

[Caudill et Wirfs-Brock, 1986] P. J. Caudill and A. Wirfs-Brock, A third-generation Smalltalk-80 implementation, *OOPSLA'86 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 21(11) of *ACM SIGPLAN Notices*. ACM Press, October 1986.

[CenterLine Software, 1992] CenterLine Software, Cambridge, MA. *CodeCenter, The Programming Environment*, 1992.

[Cheney, 1970] C. J. Cheney, A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677-8, November 1970.

[Christopher, 1984] T. W. Christopher, Reference count garbage collection. *Software Practice and Experience*, 14(6):503-507, June 1984.

[Cohen et Nicolau, 1983] J. Cohen and A. Nicolau, Comparison of compacting algorithms for garbage collection. *ACM Transactions on Programming Languages and Systems*, 5(4):532-553, 1983.

[Collins, 1960] G. E. Collins, A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655-657, December 1960.

[Coplien, 1992] James Coplien, *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.

[OMG, 1997] Object Management Group, Specification of the Portable ObjectAdapter (POA), OMG Document orbos/97-05-15 ed., June 1997.

[Cormen *et al.*, 2002] T. H. Cormen, C. E. Leiserson, R. L. Rivest et C. Stein, Introduction à l'algorithmique. Dunod, Deuxième édition, 2002.

[Courts, 1988] R. Courts. Improving locality of reference in a garbage-collecting memory management-system. *Communications of the ACM*, 31(9):1128-1138, 1988.

[Detlefs, 1992] D. L. Detlefs, Garbage collection and runtime typing as a C++ library. In *USENIX C++ Conference*, Portland, Oregon, USENIX Association. August 1992.

[Detlefs, 1994] David L. Detlefs, Concurrent garbage collection for C++. In Peter Lee,

editor, *Topics in Advanced Language Implementation*. MIT Press, 1994.

[DeTrevill, 1990] John DeTrevill, Experience with concurrent garbage collectors for Modula-2+. Technical Report 64, DEC Systems Research Center, Palo Alto, CA, August 1990.

[Deutsch et Bobrow, 1976] L. P. Deutsch and D. G. Bobrow, An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522-526, September 1976.

[Dijkstra *et al.*, 1976] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens, On-the-fly garbage collection: An exercise in cooperation. In *Lecture Notes in Computer Science, No. 46*. Springer-Verlag, New York, 1976.

[Dijkstra *et al.*, 1978] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens, On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965-975, November 1978.

[Edelson et Pohl, 1990] D. R. Edelson and I. Pohl, The case for garbage collection in C++, Eric Jul and Niels-Christian Juul, editors. *OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems*, Ottawa, October 1990.

[Edelson et Pohl, 1991] D. R. Edelson and I. Pohl, A copying collector for C++. In *Usenix C++ Conference Proceedings*, pages 85-102. USENIX Association, 1991.

[Edelson, 1992] D. R. Edelson, A mark-and-sweep collector for C++.*Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, Albuquerque, NM, ACM Press, January 1992.

[Ellis et Detlefs, 1993] J. R. Ellis and D. L. Detlefs, Safe, efficient garbage collection for C++. Technical report, Xerox PARC, Palo Alto, CA, 1993.

[Ellis et Stroustrup, 1990] M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[Ellis, 1993] J. R. Ellis, Put up or shut up. In Moss *et al. OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, October 1993.

[Fenichel et Yochelson, 1969] R. R. Fenichel and J. C. Yochelson, A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM*, 12(11):611-612, November 1969.

[Ferreira, 1991] Paulo Ferreira. Garbage collection in C++. In *Workshop on Extensions to C++*, Lisbon, June 1991.

[Fisher, 1975] D. A. Fisher. Copying cyclic list structure in linear time using bounded workspace. *Communications of the ACM*, 18(5):251-252, May 1975.

[Frank, 1997] E. Frank, DCOM: Microsoft Distributed Component Object Model, Redmond III November, 1997.

[Gelernter *et al.*, 1960] H. Gelernter, J. R. Hansen, and C. L. Gerberich, A Fortran-compiled list processing language. *Journal of the ACM*, 7(2):87-101, April 1960.

[Goldberg et Robson, 1983] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[Goldberg, 1989] B. Goldberg, Generational reference counting: A reduced-communication distributed storage reclamation scheme, *Proceedings of SIGPLAN'89 Conference on Programming Languages Design and Implementation*, volume 24(7) of *ACM SIGPLAN Notices*, Portland, Oregon, ACM Press, June 1989.

[Harrison et Ossher, 1993] W. Harrison and H. Ossher, "Subject-oriented programming: a critique of pure objects," *in Proceedings of OOPSLA'93*, September 26-Octobre 1, 1993, pp. 411-428, Washington D.C, USA.

[Hart, 2002] James Hart, Java J2Se 1.4 Core Platform Update, Wrox, 2002.

[Hart et Evans, 1974] T. P. Hart and T. G. Evans, Notes on implementing LISP for the M-460 computer. In Berkeley and Bobrow, *The Programming Language LISP: Its Operation and Applications*. Information International, Inc., Cambridge, MA, fourth edition, pages 191-203, 1974.

[Henning et Vinoski. 2004] M. Henning et S. Vinoski. "Advanced CORBA Programming with C++", Addison-Wesley Series, 1120 pp, 1999.

[Hewitt, 1977] Carl Hewitt, Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323-364, June 1977.

[Hirzel et al., 2002] M. Hirzel , J. Henkel , A. Diwan , M. Hind, Understanding the connectivity of heap objects, Proceedings of the third international symposium on Memory management, Berlin, Germany, June 20-21, 2002.

[Hosking, 1991] A. L. Hosking, Main memory management for persistence, *OOPSLA/ECOOP '91 Workshop on Garbage Collection in Object-Oriented Systems, Addendum to OOPSLA'91 Proceedings*, October 1991.

[Hudack et Keller, 1982] P. R. Hudak and R. M. Keller. Garbage collection and task deletion in distributed applicative processing systems. In LFP *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, Pittsburgh, PA, ACM Press, pages 168-178, August 1982.

[Hudson *et al.*, 1991] R. L. Hudson, J. E. B. Moss, A. Diwan, and C. F. Weight, A language-independent garbage collector toolkit. Technical Report COINS 91-47, University of Massachusetts at Amherst, Department of Computer and Information Science, September 1991.

[Hughes, 1982] R. J. M. Hughes. A semi-incremental garbage collection algorithm. *Software Practice and Experience*, 12(11):1081-1084, November 1982.

[Hughes, 1985] R. J. M. Hughes, A distributed garbage collection algorithm. *Record of the*

1985 Conference on Functional Programming and Computer Architecture, volume 201 of *Lecture Notes in Computer Science*, Nancy, France, Springer-Verlag, pages 256-272, September 1985.

[Hutchinson, 1987] N. Hutchinson, *Emerald: An Object-Based Language for Distributed Programming*. PhD thesis, University of Washington, January 1987.

[Ilango, 2001] S. K. Ilango, Jini Technology: An Overview , Prentice Hall PTR 2001.

[IONA, 2006] IONA Technologies, http://www.iona.com/products/orbacus.htm,2006.

[JAC, 2006] http://jac.objectweb.org/, AOPSYS compagnie, Paris, France, 2006.

[Jones et Lins, 1996] R. E. Jones and R. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[Jonkers, 1979] H. B. M. Jonkers. A fast garbage compaction algorithm. *Information Processing Letters*, 9(1):25-30, July 1979.

[Kernighan et Ritchie, 1988] B.W. Kernighan et D.M. Ritchie, The C Programming Language (Ansi C), Edition Prentice Hall, 1988.

[Kicsales *et al.*, 1997] G. Kicsales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. Technical Report SPL97-008 P9710042, XEROX PARC, February 1997.

[Knuth, 1973] D. E. Knuth, *The Art of Computer Programming*, volume I: Fundamental Algorithms, chapter 2. Addison-Wesley, second edition, 1973.

[Kuse et Kamimura, 1991] K. Kuse and T. Kamimura, Generational garbage collection for C-based object-oriented languages. In Paul R. Wilson and Barry Hayes, editors. *OOPSLA/ECOOP '91 Workshop on Garbage Collection in Object-Oriented Systems, Addendum to OOPSLA'91 Proceedings*, October 1991.

[Lang *et al.*, 1992] B. Lang, C. Quenniac, and J. Piquer, Garbage collecting the world, *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, Albuquerque, NM, ACM Press, January 1992.

[Lermen et D. Maurer, 1986] C.-W. Lermen and D. Maurer, A protocol for distributed reference counting, *Conference Record of the 1986 ACM Symposium on Lisp and Functional Programming*, ACM SIGPLAN Notices, Cambridge, MA, ACM Press, August 1986.

[Lesiecki, 2002] http://www-128.ibm.com/developerworks/library/j-aspectj/, 2002.

[Levanoni et Petrank, 2001] Y. Levanoni et E. Petrank, An on-the-fly Reference Counting Garbage Collector for Java, Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, Volume 36 Issue 11 October 2001.

[Lieberman et Hewitt, H. Lieberman and C. E. Hewitt, A real-time garbage collector based on the

1981] lifetimes of objects. *Communications of the ACM*, 26(6):419-429, 1983. Also report TM-184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569, 1981.

[Lins et Jones, 1993] R. D. Lins and R. E. Jones. Cyclic weighted reference counting. In K. Boyanov, editor, *Procedings of WP & DP'93 Workshop on Parallel and Distributed Processing*, pages 369-382, Sofia, Bulgaria, May 1993.

[Malenfant, 1995] Jacques Malenfant: On the Semantic Diversity of Delegation-Based Programming Languages. OOPSLA :215-230, 1995.

[McCarthy, 1960] J. McCarthy, Recursive functions of symbolic expressions and their computation by machine. Communications of the ACM, 3:184-195, 1960.

[Mcheick *et al.*, 2006] H. Mcheick, H. Mili, H.Msheik, and A. Sioud. Views Distributed Life Cycle Management approaches Using Aspect Oriented Progamming. *IADIS International Conference Applied Computing*. San Sebastian, Spain, 25-28 février, 2006.

[Metropolis *et al.*, 1980] N. Metropolis, J. Howlett, and Gian-Carlo Rota, editors. *A History of Computing in the Twentieth Century*. Academic Press, 1980.

[Meyer, 1988] Bertrand Meyer, *Object-oriented Software Construction*. Prentice-Hall, 1988.

[Mili *et al.*, 1999] Hafedh Mili, Ali Mili, Joumana Dargham, Omar Cherkaoui, and Robert Godin, "View Programming: Towards a Framework for Decentralized Development and Execution of OO Programs," *Proceedings of TOOLS USA '99*, Prentice-Hall, pp. 211-221, Aug. 1-5, 1999.

[Mili *et al.*, 2002] H. Mili, H. Mcheick, and S. Sadou, "CORBAViews: Distributing objects that support several functional aspects," *Journal of Object Technology*, August, 2002, Zurich, Suisse.

[Mohamed-Ali, 1984] K. A. Mohamed-Ali, *Object Oriented Storage Management and Garbage Collection in Distributed Processing Systems*. PhD thesis, Royal Institute of Technology, Stockholm, December 1984.

[Moreau, 1998] L. Moreau. A distributed garbage collector with diffusion tree reorganisation and mobile objects. *Proceedings of International Conference on Functional Programming*, Baltimore, MA, ACM Press. pages 204-215, September 1998.

[Nori, 1979] A. K. Nori, A storage reclamation system for an applicative multiprocessor system. Master's thesis, University of Utah, Salt Lake City, Utah, 1979.

[OMG, 1995] Object Management group. "The Common Object Request Broker: Architecture and Specification," 2.0 December, 1995.

[Piquer, 1991] J. M. Piquer, Indirect reference counting: A distributed garbage collection algorithm, *PARLE'91 Parallel Architectures and Languages Europe*, volume 505 of *Lecture Notes in Computer Science*, Eindhoven, The Netherlands,

Springer-Verlag, , June 1991.

[Pitt et McNiff, 2001] E. Pitt et K. McNiff, Java.Rmi the Remote Method Invocation Guide: Programmer's Guide and Class Reference, Juin 2001.

[Plainfossé and M Shapiro, 1992] D Plainfossé and M Shapiro, Experience with fault-tolerant garbage collection in a distributed Lisp system, *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, St Malo, France, 16-18 Springer-Verlag, September 1992.

[Pure Software, 1992] Pure Software, Los Altos, CA. *Purify*, 1992.

[Rammer et Szpuszta, 2005] Ingo Rammer, Mario Szpuszta, Advanced .Net Remoting, APRESS, 2005.

[Rovner, 1985] P. Rovner, On adding garbage collection and runtime types to a strongly-typed, statically-checked, concurrent language. Technical Report CSL-84-7, Xerox PARC, Palo Alto, CA, July 1985.

[Rudaclis, 1986] M. Rudalics, Distributed copying garbage collection. In LFP *Conference Record of the 1986 ACM Symposium on Lisp and Functional Programming*, ACM SIGPLAN Notices, Cambridge, MA, ACM Press., pages 364-372, August 1986.

[Seliger, 1990] R. Seliger, Extending C++ to support remote procedure call, concurrency, exception handling and garbage collection. In *Usenix C++ Conference Proceedings*, pages 241-264. USENIX Association, 1990.

[Shapiro *et al.*, 1992] M. Shapiro, P. Dickman, and D. Plainfossé, Robust, distributed references and acyclic garbage collection. In *Symposium on Principles of Distributed Computing*, pages 135-146, Vancouver, Canada, ACM Press, August 1992.

[Sousa, 1993] P. Sousa, Garbage collection of persistent objects in a distributed object-oriented platform, *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, October 1993.

[Spinczyk O. *et al.*, 2002] Olaf Spinczyk, Andreas Gal et Wolfgang Schroder-Preikschat, "AspectC++: An Aspect-Oriented Extension ro the C++ Programming Language", Australian Computer Society, Inc. Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems. Sydney, Australia, 2002.

[Spring, 2006] Spring Framework, http://www.springframework.org/, 2006.

[Stamos, 1984] J. W. Stamos, Static grouping of small objects to enhance performance of a paged virtual memory. *ACM Transactions on Computer Systems*, 2(3):155-180, May 1984.

[Steele, 1975] G. L. Steele, Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495-508, September 1975.

[Stroustrup, 2004] B. Stroustrup, The C++ Programming Language. Addison-Wesley Third Edition and Special Edition, 2004.

[SUN Microsystems, 2006] The Source for Sun Developer Solutions, http://java.sun.com/products/hotspot/, 2006.

[Tanenbaum et Steen, 2002] A. S Tanenbaum. and M.Van Steen, Distributed System , Principle and Paradigm Prentice Hall Upper Saddle River, 2002.

[Tarr *et al.*, 1999] P. Tarr, H. Ossher, W. Harrison, and S.M. Sutton, Jr., N Degrees of Separation: Multi-Dimensional Separation of Concerns. ICSE-21, pp. 107-119, 1999.

[Thorelli, 1972] L. Thorelli. Marking algorithms. *BIT*, 12(4):555-568, 1972.

[Vaysse, 2005] G. Vaysse, http://afpac.gforge.inria.fr/docs/taco/utilisateur.pdf,2005.

[Vestal, 1987] S. C. Vestal, *Garbage Collection: An Exercise in Distributed, Fault-Tolerant Programming*. PhD thesis, University of Washington, Seattle, WA, 1987.

[Weizenbaum, 1963] J. Weizenbaum, Symmetric list processor. *Communications of the ACM*, 6(9):524-544, September 1963.

[Wilson, 1990] P. R. Wilson, Some issues and strategies in heap management and memory hierarchies. *OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems*, Ottawa, October 1990.

[Wilson, 1994] Paul R. Wilson, Uniprocessor garbage collection techniques. Technical report, University of Texas, January 1994.

[Zorn, 1989] B. G. Zorn, *Comparative Performance Evaluation of Garbage Collection Algorithms*. PhD thesis, University of California at Berkeley, Technical Report UCB/CSD 89/544, March 1989.

[Zorn, 1990] B. Zorn, Barrier methods for garbage collection. Technical Report CU-CS-494-90, University of Colorado, Boulder, November 1990.